

The Pyramid Web Framework

Version 1.7.6

Chris McDonough

Contents

Front Matter	1
0.1 Tutorials	38
0.2 Narrative Documentation	295
0.3 API Documentation	672
0.4 p* Scripts Documentation	851
Change History	857
Glossary and Index	1157

Front Matter

Copyright, Trademarks, and Attributions

“The Pyramid Web Framework, Version 1.7.6”

by Chris McDonough


Copyright © 2008-2011, Agendaless Consulting.

ISBN-10: 0615445675

ISBN-13: 978-0615445670

First print publishing: February, 2011

All rights reserved. This material may be copied or distributed only subject to the terms and conditions set forth in the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. You must give the original author credit. You may not use this work for commercial purposes. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

 While the Pyramid documentation is offered under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License, the Pyramid *software* is offered under a less restrictive (BSD-like) license .

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. However, use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as-is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book. No patent liability is assumed with respect to the use of the information contained herein.

Attributions

Editor: Casey Duncan

Contributors: Ben Bangert, Blaise Laflamme, Rob Miller, Mike Orr, Carlos de la Guardia, Paul Everitt, Tres Seaver, John Shipman, Marius Gedminas, Chris Rossi, Joachim Krebs, Xavier Spriet, Reed O'Brien, William Chambers, Charlie Choiniere, Jamaludin Ahmad, Graham Higgins, Patricio Paez, Michael Merickel, Eric Ongerth, Niall O'Higgins, Christoph Zwerschke, John Anderson, Atsushi Odagiri, Kirk Strauser, JD Navarro, Joe Dallago, Savoir-Faire Linux, Łukasz Fidosz, Christopher Lambacher, Claus Conrad, Chris Beelby, Phil Jenvey and a number of people with only pseudonyms on GitHub.

Cover Designer: Hugues Laflamme of Kemeneur.

Used with permission:

The *Request and Response Objects* chapter is adapted, with permission, from documentation originally written by Ian Bicking.

The *Much Ado About Traversal* chapter is adapted, with permission, from an article written by Rob Miller.

The *Logging* is adapted, with permission, from the Pylons documentation logging chapter, originally written by Phil Jenvey.

Print Production

The print version of this book was produced using the Sphinx documentation generation system and the LaTeX typesetting system.

Contacting The Publisher

Please send documentation licensing inquiries, translation inquiries, and other business communications to Agendaless Consulting. Please send software and other technical queries to the Pylons-devel mailing list.

HTML Version and Source Code

An HTML version of this book is freely available via <http://docs.pylonsproject.org/projects/pyramid/en/latest/>

The source code for the examples used in this book are available within the Pyramid software distribution, always available via <https://github.com/Pylons/pyramid>

Typographical Conventions

Literals, filenames, and function arguments are presented using the following style:

`argument1`

Warnings which represent limitations and need-to-know information related to a topic or concept are presented in the following style:



This is a warning.

Notes which represent additional information related to a topic or concept are presented in the following style:



This is a note.

We present Python method names using the following style:

`pyramid.config.Configurator.add_view()`

We present Python class names, module names, attributes, and global variables using the following style:

`pyramid.config.Configurator.registry`

References to glossary terms are presented using the following style:

Pylons

URLs are presented using the following style:

Pylons

References to sections and chapters are presented using the following style:

Traversal

Code and configuration file blocks are presented in the following style:

```
1 def foo(abc):  
2     pass
```

Example blocks representing UNIX shell commands are prefixed with a \$ character, e.g.:

```
$ $VENV/bin/py.test -q
```

See *venv* for the meaning of \$VENV.

Example blocks representing Windows commands are prefixed with a drive letter with an optional directory name, e.g.:

```
c:\examples> %VENV%\Scripts\py.test -q
```

See *venv* for the meaning of %VENV%.

When a command that should be typed on one line is too long to fit on a page, the backslash \ is used to indicate that the following printed line should be part of the command:

```
$VENV/bin/py.test tutorial/tests.py --cov-report term-missing \  
--cov=tutorial -q
```

A sidebar, which presents a concept tangentially related to content discussed on a page, is rendered like so:

This is a sidebar

Sidebar information.

When multiple objects are imported from the same package, the following convention is used:

```
from foo import (  
    bar,  
    baz,  
)
```

It may look unusual, but it has advantages:

- It allows one to swap out the higher-level package `foo` for something else that provides the similar API. An example would be swapping out one database for another (e.g., graduating from SQLite to PostgreSQL).
- Looks more neat in cases where a large number of objects get imported from that package.
- Adding or removing imported objects from the package is quicker and results in simpler diffs.

Author Introduction

Welcome to “The Pyramid Web Framework”. In this introduction, I’ll describe the audience for this book, I’ll describe the book content, I’ll provide some context regarding the genesis of Pyramid, and I’ll thank some important people.

I hope you enjoy both this book and the software it documents. I’ve had a blast writing both.

Audience

This book is aimed primarily at a reader that has the following attributes:

- At least a moderate amount of *Python* experience.
- A familiarity with web protocols such as HTTP and CGI.

If you fit into both of these categories, you’re in the direct target audience for this book. But don’t worry, even if you have no experience with Python or the web, both are easy to pick up “on the fly”.

Python is an *excellent* language in which to write applications; becoming productive in Python is almost mind-blowingly easy. If you already have experience in another language such as Java, Visual Basic, Perl, Ruby, or even C/C++, learning Python will be a snap; it should take you no longer than a couple of days to become modestly productive. If you don’t have previous programming experience, it will be slightly harder, and it will take a little longer, but you’d be hard-pressed to find a better “first language.”

Web technology familiarity is assumed in various places within the book. For example, the book doesn’t try to define common web-related concepts like “URL” or “query string.” Likewise, the book describes various interactions in terms of the HTTP protocol, but it does not describe how the HTTP protocol works in detail. Like any good web framework, though, Pyramid shields you from needing to know most of the gory details of web protocols and low-level data structures. As a result, you can usually avoid becoming “blocked” while you read this book even if you don’t yet deeply understand web technologies.

Book Content

This book is divided into four major parts:

Tutorials

Each tutorial builds a sample application or implements a set of concepts with a sample; it then describes the application or concepts in terms of the sample. You should read the tutorials if you want a guided tour of Pyramid.

Narrative Documentation

This is documentation which describes Pyramid concepts in narrative form, written in a largely conversational tone. Each narrative documentation chapter describes an isolated Pyramid concept. You should be able to get useful information out of the narrative chapters if you read them out-of-order, or when you need only a reminder about a particular topic while you're developing an application.

API Documentation

Comprehensive reference material for every public API exposed by Pyramid. The API documentation is organized alphabetically by module name.

p Scripts Documentation*

p* scripts included with Pyramid.

The Genesis of `repoze.bfg`

Before the end of 2010, Pyramid was known as `repoze.bfg`.

I wrote `repoze.bfg` after many years of writing applications using *Zope*. Zope provided me with a lot of mileage: it wasn't until almost a decade of successfully creating applications using it that I decided to write a different web framework. Although `repoze.bfg` takes inspiration from a variety of web frameworks, it owes more of its core design to Zope than any other.

The Repoze "brand" existed before `repoze.bfg` was created. One of the first packages developed as part of the Repoze brand was a package named `repoze.zope2`. This was a package that allowed Zope 2 applications to run under a *WSGI* server without modification. Zope 2 did not have reasonable WSGI support at the time.

During the development of the `repoze.zope2` package, I found that replicating the Zope 2 “publisher” – the machinery that maps URLs to code – was time-consuming and fiddly. Zope 2 had evolved over many years, and emulating all of its edge cases was extremely difficult. I finished the `repoze.zope2` package, and it emulates the normal Zope 2 publisher pretty well. But during its development, it became clear that Zope 2 had simply begun to exceed my tolerance for complexity, and I began to look around for simpler options.

I considered using the Zope 3 application server machinery, but it turned out that it had become more indirect than the Zope 2 machinery it aimed to replace, which didn’t fulfill the goal of simplification. I also considered using Django and Pylons, but neither of those frameworks offer much along the axes of traversal, contextual declarative security, or application extensibility; these were features I had become accustomed to as a Zope developer.

I decided that in the long term, creating a simpler framework that retained features I had become accustomed to when developing Zope applications was a more reasonable idea than continuing to use any Zope publisher or living with the limitations and unfamiliarities of a different framework. The result is what is now Pyramid.

The Genesis of Pyramid

What was `repoze.bfg` has become Pyramid as the result of a coalition built between the *Repoze* and *Pylons* community throughout the year 2010. By merging technology, we’re able to reduce duplication of effort, and take advantage of more of each others’ technology.

Thanks

This book is dedicated to my grandmother, who gave me my first typewriter (a Royal), and my mother, who bought me my first computer (a VIC-20).

Thanks to the following people for providing expertise, resources, and software. Without the help of these folks, neither this book nor the software which it details would exist: Paul Everitt, Tres Seaver, Andrew Sawyers, Malthe Borch, Carlos de la Guardia, Chris Rossi, Shane Hathaway, Daniel Holth, Wichert Akkerman, Georg Brandl, Blaise Laflamme, Ben Bangert, Casey Duncan, Hugues Laflamme, Mike Orr, John Shipman, Chris Beelby, Patricio Paez, Simon Oram, Nat Hardwick, Ian Bicking, Jim Fulton, Michael Merickel, Tom Moroz of the Open Society Institute, and Todd Koym of Environmental Health Sciences.

Thanks to Guido van Rossum and Tim Peters for Python.

Special thanks to Tricia for putting up with me.

Defending Pyramid's Design

From time to time, challenges to various aspects of Pyramid design are lodged. To give context to discussions that follow, we detail some of the design decisions and trade-offs here. In some cases, we acknowledge that the framework can be made better and we describe future steps which will be taken to improve it. In others we just file the challenge as noted, as obviously you can't please everyone all of the time.

Pyramid Provides More Than One Way to Do It

A canon of Python popular culture is “TIOOWTDI” (“there is only one way to do it”, a slighting, tongue-in-cheek reference to Perl’s “TIMTOWTDI”, which is an acronym for “there is more than one way to do it”).

Pyramid is, for better or worse, a “TIMTOWTDI” system. For example, it includes more than one way to resolve a URL to a *view callable*: via *url dispatch* or *traversal*. Multiple methods of configuration exist: *imperative configuration*, *configuration decoration*, and *ZCML* (optionally via *pyramid_zcml*). It works with multiple different kinds of persistence and templating systems. And so on. However, the existence of most of these overlapping ways to do things are not without reason and purpose: we have a number of audiences to serve, and we believe that TIMTOWTDI at the web framework level actually *prevents* a much more insidious and harmful set of duplication at higher levels in the Python web community.

Pyramid began its life as `repoze.bfg`, written by a team of people with many years of prior *Zope* experience. The idea of *traversal* and the way *view lookup* works was stolen entirely from Zope. The authorization subsystem provided by Pyramid is a derivative of Zope’s. The idea that an application can be *extended* without forking is also a Zope derivative.

Implementations of these features were *required* to allow the Pyramid authors to build the bread-and-butter CMS-type systems for customers in the way in which they were accustomed. No other system, save for Zope itself, had such features, and Zope itself was beginning to show signs of its age. We were becoming hampered by consequences of its early design mistakes. Zope’s lack of documentation was also difficult to work around. It was hard to hire smart people to work on Zope applications because there was no comprehensive documentation set which explained “it all” in one consumable place, and it was too large and self-inconsistent to document properly. Before `repoze.bfg` went under development, its authors obviously looked around for other frameworks that fit the bill. But no non-Zope framework did. So we embarked on building `repoze.bfg`.

As the result of our research, however, it became apparent that, despite the fact that no *one* framework had all the features we required, lots of existing frameworks had good, and sometimes very compelling ideas. In particular, *URL dispatch* is a more direct mechanism to map URLs to code.

So, although we couldn't find a framework, save for Zope, that fit our needs, and while we incorporated a lot of Zope ideas into BFG, we also emulated the features we found compelling in other frameworks (such as *url dispatch*). After the initial public release of BFG, as time went on, features were added to support people allergic to various Zope-isms in the system, such as the ability to configure the application using *imperative configuration* and *configuration decoration*, rather than solely using *ZCML*, and the elimination of the required use of *interface* objects. It soon became clear that we had a system that was very generic, and was beginning to appeal to non-Zope users as well as ex-Zope users.

As the result of this generalization, it became obvious BFG shared 90% of its feature set with the feature set of Pylons 1, and thus had a very similar target market. Because they were so similar, choosing between the two systems was an exercise in frustration for an otherwise non-partisan developer. It was also strange for the Pylons and BFG development communities to be in competition for the same set of users, given how similar the two frameworks were. So the Pylons and BFG teams began to work together to form a plan to merge. The features missing from BFG (notably *view handler* classes, flash messaging, and other minor missing bits), were added to provide familiarity to ex-Pylons users. The result is Pyramid.

The Python web framework space is currently notoriously balkanized. We're truly hoping that the amalgamation of components in Pyramid will appeal to at least two currently very distinct sets of users: Pylons and BFG users. By unifying the best concepts from Pylons and BFG into a single codebase, and leaving the bad concepts from their ancestors behind, we'll be able to consolidate our efforts better, share more code, and promote our efforts as a unit rather than competing pointlessly. We hope to be able to shortcut the pack mentality which results in a *much larger* duplication of effort, represented by competing but incredibly similar applications and libraries, each built upon a specific low level stack that is incompatible with the other. We'll also shrink the choice of credible Python web frameworks down by at least one. We're also hoping to attract users from other communities (such as Zope's and TurboGears') by providing the features they require, while allowing enough flexibility to do things in a familiar fashion. Some overlap of functionality to achieve these goals is expected and unavoidable, at least if we aim to prevent pointless duplication at higher levels. If we've done our job well enough, the various audiences will be able to coexist and cooperate rather than firing at each other across some imaginary web framework DMZ.

Pyramid Uses a Zope Component Architecture ("ZCA") Registry

Pyramid uses a *Zope Component Architecture* (ZCA) "component registry" as its *application registry* under the hood. This is a point of some contention. Pyramid is of a *Zope* pedigree, so it was natural for its developers to use a ZCA registry at its inception. However, we understand that using a ZCA registry has issues and consequences, which we've attempted to address as best we can. Here's an introspection about Pyramid use of a ZCA registry, and the trade-offs its usage involves.

Problems

The global API that may be used to access data in a ZCA component registry is not particularly pretty or intuitive, and sometimes it's just plain obtuse. Likewise, the conceptual load on a casual source code reader of code that uses the ZCA global API is somewhat high. Consider a ZCA neophyte reading the code that performs a typical “unnamed utility” lookup using the `zope.component.getUtility()` global API:

```
1 from pyramid.interfaces import ISettings
2 from zope.component import getUtility
3 settings = getUtility(ISettings)
```

After this code runs, `settings` will be a Python dictionary. But it's unlikely that any civilian would know that just by reading the code. There are a number of comprehension issues with the bit of code above that are obvious.

First, what's a “utility”? Well, for the purposes of this discussion, and for the purpose of the code above, it's just not very important. If you really want to know, you can read this. However, still, readers of such code need to understand the concept in order to parse it. This is problem number one.

Second, what's this `ISettings` thing? It's an *interface*. Is that important here? Not really, we're just using it as a key for some lookup based on its identity as a marker: it represents an object that has the dictionary API, but that's not very important in this context. That's problem number two.

Third of all, what does the `getUtility` function do? It's performing a lookup for the `ISettings` “utility” that should return... well, a utility. Note how we've already built up a dependency on the understanding of an *interface* and the concept of “utility” to answer this question: a bad sign so far. Note also that the answer is circular, a *really* bad sign.

Fourth, where does `getUtility` look to get the data? Well, the “component registry” of course. What's a component registry? Problem number four.

Fifth, assuming you buy that there's some magical registry hanging around, where *is* this registry? *Homina homina*... “around”? That's sort of the best answer in this context (a more specific answer would require knowledge of internals). Can there be more than one registry? Yes. So in *which* registry does it find the registration? Well, the “current” registry of course. In terms of Pyramid, the current registry is a thread local variable. Using an API that consults a thread local makes understanding how it works non-local.

You've now bought in to the fact that there's a registry that is just hanging around. But how does the registry get populated? Why, via code that calls directives like `config.add_view`. In this particular case, however, the registration of `ISettings` is made by the framework itself under the hood: it's not present in any user configuration. This is extremely hard to comprehend. Problem number six.

Clearly there's some amount of cognitive load here that needs to be borne by a reader of code that extends the Pyramid framework due to its use of the ZCA, even if they are already an expert Python programmer and an expert in the domain of web applications. This is suboptimal.

Ameliorations

First, the primary amelioration: Pyramid *does not expect application developers to understand ZCA concepts or any of its APIs*. If an *application* developer needs to understand a ZCA concept or API during the creation of a Pyramid application, we've failed on some axis.

Instead the framework hides the presence of the ZCA registry behind special-purpose API functions that *do* use ZCA APIs. Take for example the `pyramid.security.authenticated_userid` function, which returns the userid present in the current request or `None` if no userid is present in the current request. The application developer calls it like so:

```
1 from pyramid.security import authenticated_userid
2 userid = authenticated_userid(request)
```

They now have the current user id.

Under its hood however, the implementation of `authenticated_userid` is this:

```
1 def authenticated_userid(request):
2     """ Return the userid of the currently authenticated user or
3     ``None`` if there is no authentication policy in effect or there
4     is no currently authenticated user. """
5
6     registry = request.registry # the ZCA component registry
7     policy = registry.queryUtility(IAAuthenticationPolicy)
8     if policy is None:
9         return None
10    return policy.authenticated_userid(request)
```

Using such wrappers, we strive to always hide the ZCA API from application developers. Application developers should just never know about the ZCA API; they should call a Python function with some object germane to the domain as an argument, and it should return a result. A corollary that follows is that any reader of an application that has been written using Pyramid needn't understand the ZCA API either.

Hiding the ZCA API from application developers and code readers is a form of enhancing domain specificity. No application developer wants to need to understand the small, detailed mechanics of how a web framework does its thing. People want to deal in concepts that are closer to the domain they're working in. For example, web developers want to know about *users*, not *utilities*. Pyramid uses the ZCA as an implementation detail, not as a feature which is exposed to end users.

However, unlike application developers, *framework developers*, including people who want to override Pyramid functionality via preordained framework plugpoints like traversal or view lookup, *must* understand the ZCA registry API.

Pyramid framework developers were so concerned about conceptual load issues of the ZCA registry API that a replacement registry implementation named `repoze.component` was actually developed. Though this package has a registry implementation which is fully functional and well-tested, and its API is much nicer than the ZCA registry API, work on it was largely abandoned, and it is not used in Pyramid. We continued to use a ZCA registry within Pyramid because it ultimately proved a better fit.

i We continued using ZCA registry rather than disusing it in favor of using the registry implementation in `repoze.component` largely because the ZCA concept of interfaces provides for use of an interface hierarchy, which is useful in a lot of scenarios (such as context type inheritance). Coming up with a marker type that was something like an interface that allowed for this functionality seemed like it was just reinventing the wheel.

Making framework developers and extenders understand the ZCA registry API is a trade-off. We (the Pyramid developers) like the features that the ZCA registry gives us, and we have long-ago borne the weight of understanding what it does and how it works. The authors of Pyramid understand the ZCA deeply and can read code that uses it as easily as any other code.

But we recognize that developers who might want to extend the framework are not as comfortable with the ZCA registry API as the original developers. So for the purpose of being kind to third-party Pyramid framework developers, we've drawn some lines in the sand.

In all core code, we've made use of ZCA global API functions, such as `zope.component.getUtility` and `zope.component.getAdapter`, the exception instead of the rule. So instead of:

```
1 from pyramid.interfaces import IAuthenticationPolicy
2 from zope.component import getUtility
3 policy = getUtility(IAuthenticationPolicy)
```

Pyramid code will usually do:

```
1 from pyramid.interfaces import IAuthenticationPolicy
2 from pyramid.threadlocal import get_current_registry
3 registry = get_current_registry()
4 policy = registry.getUtility(IAuthenticationPolicy)
```

While the latter is more verbose, it also arguably makes it more obvious what's going on. All of the Pyramid core code uses this pattern rather than the ZCA global API.

Rationale

Here are the main rationales involved in the Pyramid decision to use the ZCA registry:

- **History.** A nontrivial part of the answer to this question is “history”. Much of the design of Pyramid is stolen directly from *Zope*. Zope uses the ZCA registry to do a number of tricks. Pyramid mimics these tricks, and, because the ZCA registry works well for that set of tricks, Pyramid uses it for the same purposes. For example, the way that Pyramid maps a *request* to a *view callable* using *traversal* is lifted almost entirely from Zope. The ZCA registry plays an important role in the particulars of how this request to view mapping is done.
- **Features.** The ZCA component registry essentially provides what can be considered something like a superdictionary, which allows for more complex lookups than retrieving a value based on a single key. Some of this lookup capability is very useful for end users, such as being able to register a view that is only found when the context is some class of object, or when the context implements some *interface*.
- **Singularity.** There’s only one place where “application configuration” lives in a Pyramid application: in a component registry. The component registry answers questions made to it by the framework at runtime based on the configuration of *an application*. Note: “an application” is not the same as “a process”; multiple independently configured copies of the same Pyramid application are capable of running in the same process space.
- **Composability.** A ZCA component registry can be populated imperatively, or there’s an existing mechanism to populate a registry via the use of a configuration file (ZCML, via the optional *pyramid_zcml* package). We didn’t need to write a frontend from scratch to make use of configuration-file-driven registry population.
- **Pluggability.** Use of the ZCA registry allows for framework extensibility via a well-defined and widely understood plugin architecture. As long as framework developers and extenders understand the ZCA registry, it’s possible to extend Pyramid almost arbitrarily. For example, it’s relatively easy to build a directive that registers several views all at once, allowing app developers to use that directive as a “macro” in code that they write. This is somewhat of a differentiating feature from other (non-Zope) frameworks.
- **Testability.** Judicious use of the ZCA registry in framework code makes testing that code slightly easier. Instead of using monkeypatching or other facilities to register mock objects for testing, we inject dependencies via ZCA registrations, then use lookups in the code to find our mock objects.
- **Speed.** The ZCA registry is very fast for a specific set of complex lookup scenarios that Pyramid uses, having been optimized through the years for just these purposes. The ZCA registry contains optional C code for this purpose which demonstrably has no (or very few) bugs.
- **Ecosystem.** Many existing Zope packages can be used in Pyramid with few (or no) changes due to our use of the ZCA registry.

Conclusion

If you only *develop applications* using Pyramid, there's not much to complain about here. You just should never need to understand the ZCA registry API; use documented Pyramid APIs instead. However, you may be an application developer who doesn't read API documentation. Instead you read the raw source code, and because you haven't read the API documentation, you don't know what functions, classes, and methods even *form* the Pyramid API. As a result, you've now written code that uses internals, and you've painted yourself into a conceptual corner, needing to wrestle with some ZCA-using implementation detail. If this is you, it's extremely hard to have a lot of sympathy for you. You'll either need to get familiar with how we're using the ZCA registry or you'll need to use only the documented APIs; that's why we document them as APIs.

If you *extend* or *develop* Pyramid (create new directives, use some of the more obscure hooks as described in *Using Hooks*, or work on the Pyramid core code), you will be faced with needing to understand at least some ZCA concepts. In some places it's used unabashedly, and will be forever. We know it's quirky, but it's also useful and fundamentally understandable if you take the time to do some reading about it.

Pyramid “Encourages Use of ZCML”

ZCML is a configuration language that can be used to configure the *Zope Component Architecture* registry that Pyramid uses for application configuration. Often people claim that Pyramid “needs ZCML”.

It doesn't. In Pyramid 1.0, ZCML doesn't ship as part of the core; instead it ships in the *pyramid_zcml* add-on package, which is completely optional. No ZCML is required at all to use Pyramid, nor any other sort of frameworky declarative frontend to application configuration.

Pyramid Does Traversal, and I Don't Like Traversal

In Pyramid, *traversal* is the act of resolving a URL path to a *resource* object in a resource tree. Some people are uncomfortable with this notion, and believe it is wrong. Thankfully if you use Pyramid and you don't want to model your application in terms of a resource tree, you needn't use it at all. Instead use *URL dispatch* to map URL paths to views.

The idea that some folks believe traversal is unilaterally wrong is understandable. The people who believe it is wrong almost invariably have all of their data in a relational database. Relational databases aren't naturally hierarchical, so traversing one like a tree is not possible.

However, folks who deem traversal unilaterally wrong are neglecting to take into account that many persistence mechanisms *are* hierarchical. Examples include a filesystem, an LDAP database, a *ZODB* (or another type of graph) database, an XML document, and the Python module namespace. It is often

convenient to model the frontend to a hierarchical data store as a graph, using traversal to apply views to objects that either *are* the resources in the tree being traversed (such as in the case of ZODB) or at least ones which stand in for them (such as in the case of wrappers for files from the filesystem).

Also, many website structures are naturally hierarchical, even if the data which drives them isn't. For example, newspaper websites are often extremely hierarchical: sections within sections within sections, ad infinitum. If you want your URLs to indicate this structure, and the structure is indefinite (the number of nested sections can be "N" instead of some fixed number), a resource tree is an excellent way to model this, even if the backend is a relational database. In this situation, the resource tree is just a site structure.

Traversal also offers better composability of applications than URL dispatch, because it doesn't rely on a fixed ordering of URL matching. You can compose a set of disparate functionality (and add to it later) around a mapping of view to resource more predictably than trying to get the right ordering of URL pattern matching.

But the point is ultimately moot. If you don't want to use traversal, you needn't. Use URL dispatch instead.

Pyramid Does URL Dispatch, and I Don't Like URL Dispatch

In Pyramid, *url dispatch* is the act of resolving a URL path to a *view* callable by performing pattern matching against some set of ordered route definitions. The route definitions are examined in order: the first pattern which matches is used to associate the URL with a view callable.

Some people are uncomfortable with this notion, and believe it is wrong. These are usually people who are steeped deeply in *Zope*. Zope does not provide any mechanism except *traversal* to map code to URLs. This is mainly because Zope effectively requires use of *ZODB*, which is a hierarchical object store. Zope also supports relational databases, but typically the code that calls into the database lives somewhere in the ZODB object graph (or at least is a *view* related to a node in the object graph), and traversal is required to reach this code.

I'll argue that URL dispatch is ultimately useful, even if you want to use traversal as well. You can actually *combine* URL dispatch and traversal in Pyramid (see *Combining Traversal and URL Dispatch*). One example of such a usage: if you want to emulate something like Zope 2's "Zope Management Interface" UI on top of your object graph (or any administrative interface), you can register a route like `config.add_route('manage', '/manage/*traverse')` and then associate "management" views in your code by using the `route_name` argument to a view configuration, e.g., `config.add_view('.some.callable', context=".some.Resource", route_name='manage')`. If you wire things up this way, someone then walks up to, for example, `/manage/ob1/ob2`, they might be presented with a management interface, but walking up to `/ob1/ob2` would present them with the default object view. There are other tricks you can pull in these hybrid configurations if you're clever (and maybe masochistic) too.

Also, if you are a URL dispatch hater, if you should ever be asked to write an application that must use some legacy relational database structure, you might find that using URL dispatch comes in handy for one-off associations between views and URL paths. Sometimes it's just pointless to add a node to the object graph that effectively represents the entry point for some bit of code. You can just use a route and be done with it. If a route matches, a view associated with the route will be called. If no route matches, Pyramid falls back to using traversal.

But the point is ultimately moot. If you use Pyramid, and you really don't want to use URL dispatch, you needn't use it at all. Instead, use *traversal* exclusively to map URL paths to views, just like you do in *Zope*.

Pyramid Views Do Not Accept Arbitrary Keyword Arguments

Many web frameworks (Zope, TurboGears, Pylons 1.X, Django) allow for their variant of a *view callable* to accept arbitrary keyword or positional arguments, which are filled in using values present in the `request.POST`, `request.GET`, or route match dictionaries. For example, a Django view will accept positional arguments which match information in an associated “urlconf” such as `r'^polls/(?P<poll_id>\d+)/$`:

```
1 def aview(request, poll_id):
2     return HttpResponse(poll_id)
```

Zope likewise allows you to add arbitrary keyword and positional arguments to any method of a resource object found via traversal:

```
1 from persistent import Persistent
2
3 class MyZopeObject(Persistent):
4     def aview(self, a, b, c=None):
5         return '%s %s %c' % (a, b, c)
```

When this method is called as the result of being the published callable, the Zope request object's GET and POST namespaces are searched for keys which match the names of the positional and keyword arguments in the request, and the method is called (if possible) with its argument list filled with values mentioned therein. TurboGears and Pylons 1.X operate similarly.

Out of the box, Pyramid is configured to have none of these features. By default Pyramid view callables always accept only `request` and no other arguments. The rationale is, this argument specification matching when done aggressively can be costly, and Pyramid has performance as one of its main goals. Therefore we've decided to make people, by default, obtain information by interrogating the request

object within the view callable body instead of providing magic to do unpacking into the view argument list.

However, as of Pyramid 1.0a9, user code can influence the way view callables are expected to be called, making it possible to compose a system out of view callables which are called with arbitrary arguments. See *Using a View Mapper*.

Pyramid Provides Too Few “Rails”

By design, Pyramid is not a particularly opinionated web framework. It has a relatively parsimonious feature set. It contains no built in ORM nor any particular database bindings. It contains no form generation framework. It has no administrative web user interface. It has no built in text indexing. It does not dictate how you arrange your code.

Such opinionated functionality exists in applications and frameworks built *on top* of Pyramid. It’s intended that higher-level systems emerge built using Pyramid as a base.

See also:

See also *Pyramid Applications Are Extensible; I Don’t Believe in Application Extensibility*.

Pyramid Provides Too Many “Rails”

Pyramid provides some features that other web frameworks do not. These are features meant for use cases that might not make sense to you if you’re building a simple bespoke web application:

- An optional way to map URLs to code using *traversal* which implies a walk of a *resource tree*.
- The ability to aggregate Pyramid application configuration from multiple sources using `pyramid.config.Configurator.include()`.
- View and subscriber registrations made using *interface* objects instead of class objects (e.g., *Using Resource Interfaces in View Configuration*).
- A declarative *authorization* system.
- Multiple separate I18N *translation string* factories, each of which can name its own domain.

These features are important to the authors of Pyramid. The Pyramid authors are often commissioned to build CMS-style applications. Such applications are often frameworky because they have more than one deployment. Each deployment requires a slightly different composition of sub-applications, and the framework and sub-applications often need to be *extensible*. Because the application has more than one deployment, pluggability and extensibility is important, as maintaining multiple forks of the application, one per deployment, is extremely undesirable. Because it's easier to extend a system that uses *traversal* from the outside than it is to do the same in a system that uses *URL dispatch*, each deployment uses a *resource tree* composed of a persistent tree of domain model objects, and uses *traversal* to map *view callable* code to resources in the tree. The resource tree contains very granular security declarations, as resources are owned and accessible by different sets of users. Interfaces are used to make unit testing and implementation substitutability easier.

In a bespoke web application, usually there's a single canonical deployment, and therefore no possibility of multiple code forks. Extensibility is not required; the code is just changed in place. Security requirements are often less granular. Using the features listed above will often be overkill for such an application.

If you don't like these features, it doesn't mean you can't or shouldn't use Pyramid. They are all optional, and a lot of time has been spent making sure you don't need to know about them up front. You can build "Pylons 1.X style" applications using Pyramid that are purely bespoke by ignoring the features above. You may find these features handy later after building a bespoke web application that suddenly becomes popular and requires extensibility because it must be deployed in multiple locations.

Pyramid Is Too Big

"The Pyramid compressed tarball is larger than 2MB. It must be enormous!"

No. We just ship it with docs, test code, and scaffolding. Here's a breakdown of what's included in subdirectories of the package tree:

docs/

3.6MB

pyramid/tests/

1.3MB

pyramid/scaffolds/

133KB

pyramid/ (except for pyramid/tests and pyramid/scaffolds)

812KB

Of the approximately 34K lines of Python code in the package, the code that actually has a chance of executing during normal operation, excluding tests and scaffolding Python files, accounts for approximately 10K lines.

Pyramid Has Too Many Dependencies

Over time, we’ve made lots of progress on reducing the number of packaging dependencies Pyramid has had. Pyramid 1.2 had 15 of them. Pyramid 1.3 and 1.4 had 12 of them. The current release as of this writing, Pyramid 1.5, has only 7. This number is unlikely to become any smaller.

A port to Python 3 completed in Pyramid 1.3 helped us shed a good number of dependencies by forcing us to make better packaging decisions. Removing Chameleon and Mako templating system dependencies in the Pyramid core in 1.5 let us shed most of the remainder of them.

Pyramid “Cheats” to Obtain Speed

Complaints have been lodged by other web framework authors at various times that Pyramid “cheats” to gain performance. One claimed cheating mechanism is our use (transitively) of the C extensions provided by `zope.interface` to do fast lookups. Another claimed cheating mechanism is the religious avoidance of extraneous function calls.

If there’s such a thing as cheating to get better performance, we want to cheat as much as possible. We optimize Pyramid aggressively. This comes at a cost. The core code has sections that could be expressed with more readability. As an amelioration, we’ve commented these sections liberally.

Pyramid Gets Its Terminology Wrong (“MVC”)

“I’m a MVC web framework user, and I’m confused. Pyramid calls the controller a view! And it doesn’t have any controllers.”

If you are in this camp, you might have come to expect things about how your existing “MVC” framework uses its terminology. For example, you probably expect that models are ORM models, controllers are classes that have methods that map to URLs, and views are templates. Pyramid indeed has each of these concepts, and each probably *works* almost exactly like your existing “MVC” web framework. We just don’t use the MVC terminology, as we can’t square its usage in the web framework space with historical reality.

People very much want to give web applications the same properties as common desktop GUI platforms by using similar terminology, and to provide some frame of reference for how various components in the common web framework might hang together. But in the opinion of the author, “MVC” doesn’t match the web very well in general. Quoting from the Model-View-Controller Wikipedia entry:

Though MVC comes in different flavors, control flow is generally as follows:

The user interacts with the user interface in some way (for example, presses a mouse button).

The controller handles the input event from the user interface, often via a registered handler or callback and converts the event into appropriate user action, understandable for the model.

The controller notifies the model of the user action, possibly resulting in a change in the model's state. (For example, the controller updates the user's shopping cart.)[5]

A view queries the model in order to generate an appropriate user interface (for example, the view lists the shopping cart's contents). Note that the view gets its own data from the model.

The controller may (in some implementations) issue a general instruction to the view to render itself. In others, the view is automatically notified by the model of changes in state (Observer) which require a screen update.

The user interface waits for further user interactions, which restarts the cycle.

To the author, it seems as if someone edited this Wikipedia definition, tortuously couching concepts in the most generic terms possible in order to account for the use of the term “MVC” by current web frameworks. I doubt such a broad definition would ever be agreed to by the original authors of the MVC pattern. But *even so*, it seems most MVC web frameworks fail to meet even this falsely generic definition.

For example, do your templates (views) always query models directly as is claimed in “note that the view gets its own data from the model”? Probably not. My “controllers” tend to do this, massaging the data for easier use by the “view” (template). What do you do when your “controller” returns JSON? Do your controllers use a template to generate JSON? If not, what's the “view” then? Most MVC-style GUI web frameworks have some sort of event system hooked up that lets the view detect when the model changes. The web just has no such facility in its current form; it's effectively pull-only.

So, in the interest of not mistaking desire with reality, and instead of trying to jam the square peg that is the web into the round hole of “MVC”, we just punt and say there are two things: resources and views. The resource tree represents a site structure, the view presents a resource. The templates are really just an implementation detail of any given view. A view doesn't need a template to return a response. There's no “controller”; it just doesn't exist. The “model” is either represented by the resource tree or by a “domain model” (like an SQLAlchemy model) that is separate from the framework entirely. This seems to us like more reasonable terminology, given the current constraints of the web.

Pyramid Applications Are Extensible; I Don't Believe in Application Extensibility

Any Pyramid application written obeying certain constraints is *extensible*. This feature is discussed in the Pyramid documentation chapters named *Extending an Existing Pyramid Application* and *Advanced Configuration*. It is made possible by the use of the *Zope Component Architecture* within Pyramid.

“Extensible” in this context means:

- The behavior of an application can be overridden or extended in a particular *deployment* of the application without requiring that the deployer modify the source of the original application.
- The original developer is not required to anticipate any extensibility plug points at application creation time to allow fundamental application behavior to be overridden or extended.
- The original developer may optionally choose to anticipate an application-specific set of plug points, which may be hooked by a deployer. If they choose to use the facilities provided by the ZCA, the original developer does not need to think terribly hard about the mechanics of introducing such a plug point.

Many developers seem to believe that creating extensible applications is not worth it. They instead suggest that modifying the source of a given application for each deployment to override behavior is more reasonable. Much discussion about version control branching and merging typically ensues.

It's clear that making every application extensible isn't required. The majority of web applications only have a single deployment, and thus needn't be extensible at all. However some web applications have multiple deployments, and others have *many* deployments. For example, a generic content management system (CMS) may have basic functionality that needs to be extended for a particular deployment. That CMS may be deployed for many organizations at many places. Some number of deployments of this CMS may be deployed centrally by a third party and managed as a group. It's easier to be able to extend such a system for each deployment via preordained plug points than it is to continually keep each software branch of the system in sync with some upstream source. The upstream developers may change code in such a way that your changes to the same codebase conflict with theirs in fiddly, trivial ways. Merging such changes repeatedly over the lifetime of a deployment can be difficult and time consuming, and it's often useful to be able to modify an application for a particular deployment in a less invasive way.

If you don't want to think about Pyramid application extensibility at all, you needn't. You can ignore extensibility entirely. However if you follow the set of rules defined in *Extending an Existing Pyramid Application*, you don't need to *make* your application extensible. Any application you write in the framework just *is* automatically extensible at a basic level. The mechanisms that deployers use to extend it will be necessarily coarse. Typically views, routes, and resources will be capable of being overridden. But for most minor (and even some major) customizations, these are often the only override plug points necessary. If the application doesn't do exactly what the deployment requires, it's often possible for a deployer to override a view, route, or resource, and quickly make it do what they want it to do in ways *not necessarily anticipated by the original developer*. Here are some example scenarios demonstrating the benefits of such a feature.

- If a deployment needs a different styling, the deployer may override the main template and the CSS in a separate Python package which defines overrides.
- If a deployment needs an application page to do something differently, or to expose more or different information, the deployer may override the view that renders the page within a separate Python package.
- If a deployment needs an additional feature, the deployer may add a view to the override package.

As long as the fundamental design of the upstream package doesn't change, these types of modifications often survive across many releases of the upstream package without needing to be revisited.

Extending an application externally is not a panacea, and carries a set of risks similar to branching and merging. Sometimes major changes upstream will cause you to revisit and update some of your modifications. But you won't regularly need to deal with meaningless textual merge conflicts that trivial changes to upstream packages often entail when it comes time to update the upstream package, because if you extend an application externally, there just is no textual merge done. Your modifications will also, for whatever it's worth, be contained in one, canonical, well-defined place.

Branching an application and continually merging in order to get new features and bug fixes is clearly useful. You can do that with a Pyramid application just as usefully as you can do it with any application. But deployment of an application written in Pyramid makes it possible to avoid the need for this even if the application doesn't define any plug points ahead of time. It's possible that promoters of competing web frameworks dismiss this feature in favor of branching and merging because applications written in their framework of choice aren't extensible out of the box in a comparably fundamental way.

While Pyramid applications are fundamentally extensible even if you don't write them with specific extensibility in mind, if you're moderately adventurous, you can also take it a step further. If you learn more about the *Zope Component Architecture*, you can optionally use it to expose other more domain-specific configuration plug points while developing an application. The plug points you expose needn't be as coarse as the ones provided automatically by Pyramid itself. For example, you might compose your own directive that configures a set of views for a pre-baked purpose (e.g., `restview` or `somesuch`), allowing other people to refer to that directive when they make declarations in the `includeme` of their customization package. There is a cost for this: the developer of an application that defines custom plug points for its deployers will need to understand the ZCA or they will need to develop their own similar extensibility system.

Ultimately any argument about whether the extensibility features lent to applications by Pyramid are good or bad is mostly pointless. You needn't take advantage of the extensibility features provided by a particular Pyramid application in order to affect a modification for a particular set of its deployments. You can ignore the application's extensibility plug points entirely, and use version control branching and merging to manage application deployment modifications instead, as if you were deploying an application written using any other web framework.

Zope 3 Enforces “TTW” Authorization Checks by Default; Pyramid Does Not

Challenge

Pyramid performs automatic authorization checks only at *view* execution time. Zope 3 wraps context objects with a security proxy, which causes Zope 3 also to do security checks during attribute access. I like this, because it means:

1. When I use the security proxy machinery, I can have a view that conditionally displays certain HTML elements (like form fields) or prevents certain attributes from being modified depending on the permissions that the accessing user possesses with respect to a context object.
2. I want to also expose my resources via a REST API using Twisted Web. If Pyramid performed authorization based on attribute access via Zope3’s security proxies, I could enforce my authorization policy in both Pyramid and in the Twisted-based system the same way.

Defense

Pyramid was developed by folks familiar with Zope 2, which has a “through the web” security model. This TTW security model was the precursor to Zope 3’s security proxies. Over time, as the Pyramid developers (working in Zope 2) created such sites, we found authorization checks during code interpretation extremely useful in a minority of projects. But much of the time, TTW authorization checks usually slowed down the development velocity of projects that had no delegation requirements. In particular, if we weren’t allowing untrusted users to write arbitrary Python code to be executed by our application, the burden of through the web security checks proved too costly to justify. We (collectively) haven’t written an application on top of which untrusted developers are allowed to write code in many years, so it seemed to make sense to drop this model by default in a new web framework.

And since we tend to use the same toolkit for all web applications, it’s just never been a concern to be able to use the same set of restricted-execution code under two different web frameworks.

Justifications for disabling security proxies by default notwithstanding, given that Zope 3 security proxies are viral by nature, the only requirement to use one is to make sure you wrap a single object in a security proxy and make sure to access that object normally when you want proxy security checks to happen. It is possible to override the Pyramid traverser for a given application (see *Changing the Traverser*). To get Zope3-like behavior, it is possible to plug in a different traverser which returns Zope3-security-proxy-wrapped objects for each traversed object (including the *context* and the *root*). This would have the effect of creating a more Zope3-like environment without much effort.

Pyramid uses its own HTTP exception class hierarchy rather than `webob.exc`

New in version 1.1.

The HTTP exception classes defined in `pyramid.httpexceptions` are very much like the ones defined in `webob.exc`, (e.g., `HTTPNotFound` or `HTTPForbidden`). They have the same names and largely the same behavior, and all have a very similar implementation, but not the same identity. Here's why they have a separate identity.

- Making them separate allows the HTTP exception classes to subclass `pyramid.response.Response`. This speeds up response generation slightly due to the way the Pyramid router works. The same speed up could be gained by monkeypatching `webob.response.Response`, but it's usually the case that monkeypatching turns out to be evil and wrong.
- Making them separate allows them to provide alternate `__call__` logic, which also speeds up response generation.
- Making them separate allows the exception classes to provide for the proper value of `RequestClass` (`pyramid.request.Request`).
- Making them separate gives us freedom from thinking about backwards compatibility code present in `webob.exc` related to Python 2.4, which we no longer support in Pyramid 1.1+.
- We change the behavior of two classes (`HTTPNotFound` and `HTTPForbidden`) in the module so that they can be used by Pyramid internally for `notfound` and `forbidden` exceptions.
- Making them separate allows us to influence the docstrings of the exception classes to provide Pyramid-specific documentation.
- Making them separate allows us to silence a stupid deprecation warning under Python 2.6 when the response objects are used as exceptions (related to `self.message`).

Pyramid has simpler traversal machinery than does Zope

Zope's default traverser:

- Allows developers to mutate the traversal name stack while traversing (they can add and remove path elements).
- Attempts to use an adaptation to obtain the next element in the path from the currently traversed object, falling back to `__bobo_traverse__`, `__getitem__`, and eventually `__getattr__`.

Zope's default traverser allows developers to mutate the traversal name stack during traversal by mutating `REQUEST['TraversalNameStack']`. Pyramid's default traverser (`pyramid.traversal.ResourceTreeTraverser`) does not offer a way to do this. It does not maintain a stack as a request attribute and, even if it did, it does not pass the request to resource objects while it's traversing. While it was handy at times, this feature was abused in frameworks built atop Zope (like CMF and Plone), often making it difficult to tell exactly what was happening when a traversal didn't match a view. I felt it was better for folks that wanted the feature to make them replace the traverser rather than build that particular honey pot in to the default traverser.

Zope uses multiple mechanisms to attempt to obtain the next element in the resource tree based on a name. It first tries an adaptation of the current resource to `ITraversable`, and if that fails, it falls back to attempting a number of magic methods on the resource (`__bobo_traverse__`, `__getitem__`, and `__getattr__`). My experience while both using Zope and attempting to reimplement its publisher in `repoze.zope2` led me to believe the following:

- The *default* traverser should be as simple as possible. Zope's publisher is somewhat difficult to follow and replicate due to the fallbacks it tried when one traversal method failed. It is also slow.
- The *entire traverser* should be replaceable, not just elements of the traversal machinery. Pyramid has a few big components rather than a plethora of small ones. If the entire traverser is replaceable, it's an antipattern to make portions of the default traverser replaceable. Doing so is a "knobs on knobs" pattern, which is unfortunately somewhat endemic in Zope. In a "knobs on knobs" pattern, a replaceable subcomponent of a larger component is made configurable using the same configuration mechanism that can be used to replace the larger component. For example, in Zope, you can replace the default traverser by registering an adapter. But you can also (or alternately) control how the default traverser traverses by registering one or more adapters. As a result of being able to either replace the larger component entirely or turn knobs on the default implementation of the larger component, no one understands when (or whether) they should ever override the larger component entirely. This results, over time, in a rusting together of the larger "replaceable" component and the framework itself because people come to depend on the availability of the default component in order just to turn its knobs. The default component effectively becomes part of the framework, which entirely subverts the goal of making it replaceable. In Pyramid, typically if a component is replaceable, it will itself have no knobs (it will be solid state). If you want to influence behavior controlled by that component, you will replace the component instead of turning knobs attached to the component.

Microframeworks have smaller Hello World programs

Self-described "microframeworks" exist. Bottle and Flask are two that are becoming popular. Bobo doesn't describe itself as a microframework, but its intended user base is much the same. Many others exist. We've even (only as a teaching tool, not as any sort of official project) created one using Pyramid.

The videos use BFG, a precursor to Pyramid, but the resulting code is available for Pyramid too). Microframeworks are small frameworks with one common feature: each allows its users to create a fully functional application that lives in a single Python file.

Some developers and microframework authors point out that Pyramid’s “hello world” single-file program is longer (by about five lines) than the equivalent program in their favorite microframework. Guilty as charged.

This loss isn’t for lack of trying. Pyramid is useful in the same circumstance in which microframeworks claim dominance: single-file applications. But Pyramid doesn’t sacrifice its ability to credibly support larger applications in order to achieve “hello world” lines of code parity with the current crop of microframeworks. Pyramid’s design instead tries to avoid some common pitfalls associated with naive declarative configuration schemes. The subsections which follow explain the rationale.

Application programmers don’t control the module-scope codepath (import-time side-effects are evil)

Imagine a directory structure with a set of Python files in it:

```
.
|-- app.py
|-- app2.py
`-- config.py
```

The contents of `app.py`:

```
1 from config import decorator
2 from config import L
3 import pprint
4
5 @decorator
6 def foo():
7     pass
8
9 if __name__ == '__main__':
10     import app2
11     pprint.pprint(L)
```

The contents of `app2.py`:

```

1 import app
2
3 @app.decorator
4 def bar():
5     pass

```

The contents of `config.py`:

```

1 L = []
2
3 def decorator(func):
4     L.append(func)
5     return func

```

If we `cd` to the directory that holds these files, and we run `python app.py`, given the directory structure and code above, what happens? Presumably, our `decorator` decorator will be used twice, once by the decorated function `foo` in `app.py`, and once by the decorated function `bar` in `app2.py`. Since each time the decorator is used, the list `L` in `config.py` is appended to, we’d expect a list with two elements to be printed, right? Sadly, no:

```

[chrism@thinko]$ python app.py
[<function foo at 0x7f4ea41ab1b8>,
<function foo at 0x7f4ea41ab230>,
<function bar at 0x7f4ea41ab2a8>]

```

By visual inspection, that outcome (three different functions in the list) seems impossible. We defined only two functions, and we decorated each of those functions only once, so we believe that the `decorator` decorator will run only twice. However, what we believe is in fact wrong, because the code at module scope in our `app.py` module was *executed twice*. The code is executed once when the script is run as `__main__` (via `python app.py`), and then it is executed again when `app2.py` imports the same file as `app`.

What does this have to do with our comparison to microframeworks? Many microframeworks in the current crop (e.g., Bottle and Flask) encourage you to attach configuration decorators to objects defined at module scope. These decorators execute arbitrarily complex registration code, which populates a singleton registry that is a global which is in turn defined in external Python module. This is analogous to the above example: the “global registry” in the above example is the list `L`.

Let’s see what happens when we use the same pattern with the Groundhog microframework. Replace the contents of `app.py` above with this:

```

1 from config import gh
2
3 @gh.route('/foo/')
4 def foo():
5     return 'foo'
6
7 if __name__ == '__main__':
8     import app2
9     pprint.pprint(L)

```

Replace the contents of `app2.py` above with this:

```

1 import app
2
3 @app.gh.route('/bar/')
4 def bar():
5     'return bar'

```

Replace the contents of `config.py` above with this:

```

1 from groundhog import Groundhog
2 gh = Groundhog('myapp', 'seekrit')

```

How many routes will be registered within the routing table of the “gh” Groundhog application? If you answered three, you are correct. How many would a casual reader (and any sane developer) expect to be registered? If you answered two, you are correct. Will the double registration be a problem? With our Groundhog framework’s `route` method backing this application, not really. It will slow the application down a little bit, because it will need to miss twice for a route when it does not match. Will it be a problem with another framework, another application, or another decorator? Who knows. You need to understand the application in its totality, the framework in its totality, and the chronology of execution to be able to predict what the impact of unintentional code double-execution will be.

The encouragement to use decorators which perform population of an external registry has an unintended consequence: the application developer now must assert ownership of every code path that executes Python module scope code. Module-scope code is presumed by the current crop of decorator-based microframeworks to execute once and only once. If it executes more than once, weird things will start to happen. It is up to the application developer to maintain this invariant. Unfortunately, in reality this is an impossible task, because Python programmers *do not own the module scope code path, and never will*. Anyone who tries to sell you on the idea that they do so is simply mistaken. Test runners that you may want to use to run your code’s tests often perform imports of arbitrary code in strange orders that manifest bugs like the one demonstrated above. API documentation generation tools do the same. Some people

even think it's safe to use the Python `reload` command, or delete objects from `sys.modules`, each of which has hilarious effects when used against code that has import-time side effects.

Global registry-mutating microframework programmers therefore will at some point need to start reading the tea leaves about what *might* happen if module scope code gets executed more than once, like we do in the previous paragraph. When Python programmers assume they can use the module-scope code path to run arbitrary code (especially code which populates an external registry), and this assumption is challenged by reality, the application developer is often required to undergo a painful, meticulous debugging process to find the root cause of an inevitably obscure symptom. The solution is often to rearrange application import ordering, or move an import statement from module-scope into a function body. The rationale for doing so can never be expressed adequately in the commit message which accompanies the fix, and can't be documented succinctly enough for the benefit of the rest of the development team so that the problem never happens again. It will happen again, especially if you are working on a project with other people who haven't yet internalized the lessons you learned while you stepped through module-scope code using `pdb`. This is a very poor situation in which to find yourself as an application developer: you probably didn't even know you or your team signed up for the job, because the documentation offered by decorator-based microframeworks don't warn you about it.

Folks who have a large investment in eager decorator-based configuration that populates an external data structure (such as microframework authors) may argue that the set of circumstances I outlined above is anomalous and contrived. They will argue that it just will never happen. If you never intend your application to grow beyond one or two or three modules, that's probably true. However, as your codebase grows, and becomes spread across a greater number of modules, the circumstances in which module-scope code will be executed multiple times will become more and more likely to occur and less and less predictable. It's not responsible to claim that double-execution of module-scope code will never happen. It will; it's just a matter of luck, time, and application complexity.

If microframework authors do admit that the circumstance isn't contrived, they might then argue that real damage will never happen as the result of the double-execution (or triple-execution, etc.) of module scope code. You would be wise to disbelieve this assertion. The potential outcomes of multiple execution are too numerous to predict because they involve delicate relationships between application and framework code as well as chronology of code execution. It's literally impossible for a framework author to know what will happen in all circumstances. But even if given the gift of omniscience for some limited set of circumstances, the framework author almost certainly does not have the double-execution anomaly in mind when coding new features. They're thinking of adding a feature, not protecting against problems that might be caused by the 1% multiple execution case. However, any 1% case may cause 50% of your pain on a project, so it'd be nice if it never occurred.

Responsible microframeworks actually offer a back-door way around the problem. They allow you to disuse decorator-based configuration entirely. Instead of requiring you to do the following:

```
1 | gh = Groundhog('myapp', 'seekrit')
2 |
```



```

3 @gh.route('/foo/')
4 def foo():
5     return 'foo'
6
7 if __name__ == '__main__':
8     gh.run()

```

They allow you to disuse the decorator syntax and go almost all-imperative:

```

1 def foo():
2     return 'foo'
3
4 gh = Groundhog('myapp', 'seekrit')
5
6 if __name__ == '__main__':
7     gh.add_route(foo, '/foo/')
8     gh.run()

```

This is a generic mode of operation that is encouraged in the Pyramid documentation. Some existing microframeworks (Flask, in particular) allow for it as well. None (other than Pyramid) *encourage* it. If you never expect your application to grow beyond two or three or four or ten modules, it probably doesn't matter very much which mode you use. If your application grows large, however, imperative configuration can provide better predictability.



Astute readers may notice that Pyramid has configuration decorators too. Aha! Don't these decorators have the same problems? No. These decorators do not populate an external Python module when they are executed. They only mutate the functions (and classes and methods) to which they're attached. These mutations must later be found during a scan process that has a predictable and structured import phase. Module-localized mutation is actually the best-case circumstance for double-imports. If a module only mutates itself and its contents at import time, if it is imported twice, that's OK, because each decorator invocation will always be mutating an independent copy of the object to which it's attached, not a shared resource like a registry in another module. This has the effect that double-registrations will never be performed.

Routes need relative ordering

Consider the following simple Groundhog application:

```

1 from groundhog import Groundhog
2 app = Groundhog('myapp', 'seekrit')
3
4 @app.route('/admin')
5 def admin():
6     return '<html>admin page</html>'
7
8 @app.route('/:action')
9 def do_action(action):
10     if action == 'add':
11         return '<html>add</html>'
12     if action == 'delete':
13         return '<html>delete</html>'
14     return app.abort(404)
15
16 if __name__ == '__main__':
17     app.run()

```

If you run this application and visit the URL `/admin`, you will see the “admin” page. This is the intended result. However, what if you rearrange the order of the function definitions in the file?

```

1 from groundhog import Groundhog
2 app = Groundhog('myapp', 'seekrit')
3
4 @app.route('/:action')
5 def do_action(action):
6     if action == 'add':
7         return '<html>add</html>'
8     if action == 'delete':
9         return '<html>delete</html>'
10     return app.abort(404)
11
12 @app.route('/admin')
13 def admin():
14     return '<html>admin page</html>'
15
16 if __name__ == '__main__':
17     app.run()

```

If you run this application and visit the URL `/admin`, your app will now return a 404 error. This is probably not what you intended. The reason you see a 404 error when you rearrange function definition ordering is that routing declarations expressed via our microframework’s routing decorators have an *ordering*, and that ordering matters.

In the first case, where we achieved the expected result, we first added a route with the pattern `/admin`, then we added a route with the pattern `/:action` by virtue of adding routing patterns via decorators at module scope. When a request with a `PATH_INFO` of `/admin` enters our application, the web framework loops over each of our application's route patterns in the order in which they were defined in our module. As a result, the view associated with the `/admin` routing pattern will be invoked because it matches first. All is right with the world.

In the second case, where we did not achieve the expected result, we first added a route with the pattern `/:action`, then we added a route with the pattern `/admin`. When a request with a `PATH_INFO` of `/admin` enters our application, the web framework loops over each of our application's route patterns in the order in which they were defined in our module. As a result, the view associated with the `/:action` routing pattern will be invoked because it matches first. A 404 error is raised. This is not what we wanted; it just happened due to the order in which we defined our view functions.

This is because Groundhog routes are added to the routing map in import order, and matched in the same order when a request comes in. Bottle, like Groundhog, as of this writing, matches routes in the order in which they're defined at Python execution time. Flask, on the other hand, does not order route matching based on import order. Instead it reorders the routes you add to your application based on their "complexity". Other microframeworks have varying strategies to do route ordering.

Your application may be small enough where route ordering will never cause an issue. If your application becomes large enough, however, being able to specify or predict that ordering as your application grows larger will be difficult. At some point, you will likely need to start controlling route ordering more explicitly, especially in applications that require extensibility.

If your microframework orders route matching based on complexity, you'll need to understand what is meant by "complexity", and you'll need to attempt to inject a "less complex" route to have it get matched before any "more complex" one to ensure that it's tried first.

If your microframework orders its route matching based on relative import/execution of function decorator definitions, you will need to ensure that you execute all of these statements in the "right" order, and you'll need to be cognizant of this import/execution ordering as you grow your application or try to extend it. This is a difficult invariant to maintain for all but the smallest applications.

In either case, your application must import the non-`__main__` modules which contain configuration decorations somehow for their configuration to be executed. Does that make you a little uncomfortable? It should, because *Application programmers don't control the module-scope codepath (import-time side-effects are evil)*.

Pyramid uses neither decorator import time ordering nor does it attempt to divine the relative complexity of one route to another as a means to define a route match ordering. In Pyramid, you have to maintain relative route ordering imperatively via the chronology of multiple executions of the `pyramid.config.Configurator.add_route()` method. The order in which you repeatedly call `add_route` becomes the order of route matching.

If needing to maintain this imperative ordering truly bugs you, you can use *traversal* instead of route matching, which is a completely declarative (and completely predictable) mechanism to map code to URLs. While URL dispatch is easier to understand for small non-extensible applications, traversal is a great fit for very large applications and applications that need to be arbitrarily extensible.

“Stacked object proxies” are too clever / thread locals are a nuisance

Some microframeworks use the `import` statement to get a handle to an object which *is not logically global*:

```
1 from flask import request
2
3 @app.route('/login', methods=['POST', 'GET'])
4 def login():
5     error = None
6     if request.method == 'POST':
7         if valid_login(request.form['username'],
8                         request.form['password']):
9             return log_the_user_in(request.form['username'])
10        else:
11            error = 'Invalid username/password'
12        # this is executed if the request method was GET or the
13        # credentials were invalid
```

The Pylons 1.X web framework uses a similar strategy. It calls these things “Stacked Object Proxies”, so, for purposes of this discussion, I’ll do so as well.

Import statements in Python (`import foo`, `from bar import baz`) are most frequently performed to obtain a reference to an object defined globally within an external Python module. However, in normal programs, they are never used to obtain a reference to an object that has a lifetime measured by the scope of the body of a function. It would be absurd to try to import, for example, a variable named `i` representing a loop counter defined in the body of a function. For example, we’d never try to import `i` from the code below:

```
1 def afunc():
2     for i in range(10):
3         print(i)
```

By its nature, the `request` object that is created as the result of a WSGI server’s call into a long-lived web framework cannot be global, because the lifetime of a single request will be much shorter than the lifetime of the process running the framework. A request object created by a web framework actually has more

similarity to the `i` loop counter in our example above than it has to any comparable importable object defined in the Python standard library or in normal library code.

However, systems which use stacked object proxies promote locally scoped objects, such as `request`, out to module scope, for the purpose of being able to offer users a nice spelling involving `import`. They, for what I consider dubious reasons, would rather present to their users the canonical way of getting at a `request` as `from framework import request` instead of a saner `from myframework.threadlocals import get_request`; `request = get_request()`, even though the latter is more explicit.

It would be *most* explicit if the microframeworks did not use thread local variables at all. Pyramid view functions are passed a request object. Many of Pyramid’s APIs require that an explicit request object be passed to them. It is *possible* to retrieve the current Pyramid request as a threadlocal variable, but it is an “in case of emergency, break glass” type of activity. This explicitness makes Pyramid view functions more easily unit testable, as you don’t need to rely on the framework to manufacture suitable “dummy” request (and other similarly-scoped) objects during test setup. It also makes them more likely to work on arbitrary systems, such as async servers, that do no monkeypatching.

Explicitly WSGI

Some microframeworks offer a `run()` method of an application object that executes a default server configuration for easy execution.

Pyramid doesn’t currently try to hide the fact that its router is a WSGI application behind a convenience `run()` API. It just tells people to import a WSGI server and use it to serve up their Pyramid application as per the documentation of that WSGI server.

The extra lines saved by abstracting away the serving step behind `run()` seems to have driven dubious second-order decisions related to its API in some microframeworks. For example, Bottle contains a `ServerAdapter` subclass for each type of WSGI server it supports via its `app.run()` mechanism. This means that there exists code in `bottle.py` that depends on the following modules: `wsgiref`, `flup`, `paste`, `cherrypy`, `fapws`, `tornado`, `google.appengine`, `twisted.web`, `diesel`, `gevent`, `unicorn`, `eventlet`, and `rocket`. You choose the kind of server you want to run by passing its name into the `run` method. In theory, this sounds great: I can try out Bottle on `unicorn` just by passing in a name! However, to fully test Bottle, all of these third-party systems must be installed and functional. The Bottle developers must monitor changes to each of these packages and make sure their code still interfaces properly with them. This increases the number of packages required for testing greatly; this is a *lot* of requirements. It is likely difficult to fully automate these tests due to requirements conflicts and build issues.

As a result, for single-file apps, we currently don’t bother to offer a `run()` shortcut. We tell folks to import their WSGI server of choice and run it by hand. For the people who want a server abstraction layer, we suggest that they use PasteDeploy. In PasteDeploy-based systems, the onus for making sure that the server can interface with a WSGI application is placed on the server developer, not the web framework developer, making it more likely to be timely and correct.

Wrapping up

Here’s a diagrammed version of the simplest pyramid application, where the inlined comments take into account what we’ve discussed in the *Microframeworks have smaller Hello World programs* section.

```
1 from pyramid.response import Response # explicit response, no thread local
2 from wsgiref.simple_server import make_server # explicitly WSGI
3
4 def hello_world(request): # accepts a request; no request thread local
5     # explicit response object means no response threadlocal
6     return Response('Hello world!')
7
8 if __name__ == '__main__':
9     from pyramid.config import Configurator
10    config = Configurator() # no global application object
11    config.add_view(hello_world) # explicit non-decorator registration
12    app = config.make_wsgi_app() # explicitly WSGI
13    server = make_server('0.0.0.0', 8080, app)
14    server.serve_forever() # explicitly WSGI
```

Pyramid doesn’t offer pluggable apps

It is “Pyramidic” to compose multiple external sources into the same configuration using `include()`. Any number of includes can be done to compose an application; includes can even be done from within other includes. Any directive can be used within an include that can be used outside of one (such as `add_view()`).

Pyramid has a conflict detection system that will throw an error if two included externals try to add the same configuration in a conflicting way (such as both externals trying to add a route using the same name, or both externals trying to add a view with the same set of predicates). It’s awful tempting to call this set of features something that can be used to compose a system out of “pluggable applications”. But in reality, there are a number of problems with claiming this:

- The terminology is strained. Pyramid really has no notion of a plurality of “applications”, just a way to compose configuration from multiple sources to create a single WSGI application. That WSGI application may gain behavior by including or disincluding configuration, but once it’s all composed together, Pyramid doesn’t really provide any machinery which can be used to demarcate the boundaries of one “application” (in the sense of configuration from an external that adds routes, views, etc) from another.

- Pyramid doesn't provide enough "rails" to make it possible to integrate truly honest-to-god, download-an-app-from-a-random-place and-plug-it-in-to-create-a-system "pluggable" applications. Because Pyramid itself isn't opinionated (it doesn't mandate a particular kind of database, it offers multiple ways to map URLs to code, etc), it's unlikely that someone who creates something application-like will be able to casually redistribute it to J. Random Pyramid User and have it just work by asking him to `config.include` a function from the package. This is particularly true of very high level components such as blogs, wikis, twitter clones, commenting systems, etc. The integrator (the Pyramid developer who has downloaded a package advertised as a "pluggable app") will almost certainly have made different choices about e.g. what type of persistence system he's using, and for the integrator to appease the requirements of the "pluggable application", he may be required to set up a different database, make changes to his own code to prevent his application from shadowing the pluggable app (or vice versa), and any other number of arbitrary changes.

For this reason, we claim that Pyramid has "extensible" applications, not pluggable applications. Any Pyramid application can be extended without forking it as long as its configuration statements have been composed into things that can be pulled in via `config.include`.

It's also perfectly reasonable for a single developer or team to create a set of interoperating components which can be enabled or disabled by using `config.include`. That developer or team will be able to provide the "rails" (by way of making high-level choices about the technology used to create the project, so there won't be any issues with plugging all of the components together. The problem only rears its head when the components need to be distributed to *arbitrary* users. Note that Django has a similar problem with "pluggable applications" that need to work for arbitrary third parties, even though they provide many, many more rails than does Pyramid. Even the rails they provide are not enough to make the "pluggable application" story really work without local modification.

Truly pluggable applications need to be created at a much higher level than a web framework, as no web framework can offer enough constraints to really make them work out of the box. They really need to plug into an application, instead. It would be a noble goal to build an application with Pyramid that provides these constraints and which truly does offer a way to plug in applications (Joomla, Plone, Drupal come to mind).

Pyramid Has Zope Things In It, So It's Too Complex

On occasion, someone will feel compelled to post a mailing list message that reads something like this:

```
had a quick look at pyramid ... too complex to me and not really
understand for which benefits.. I feel should consider whether it's time
for me to step back to django .. I always hated zope (useless ?)
complexity and I love simple way of thinking
```

(Paraphrased from a real email, actually.)

Let's take this criticism point-by-point.

Too Complex

If you can understand this hello world program, you can use Pyramid:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 if __name__ == '__main__':
9     config = Configurator()
10    config.add_view(hello_world)
11    app = config.make_wsgi_app()
12    server = make_server('0.0.0.0', 8080, app)
13    server.serve_forever()
```

Pyramid has over 1200 pages of documentation (printed), covering topics from the very basic to the most advanced. *Nothing* is left undocumented, quite literally. It also has an *awesome*, very helpful community. Visit the #pyramid IRC channel on freenode.net and see.

Hate Zope

I’m sorry you feel that way. The Zope brand has certainly taken its share of lumps over the years, and has a reputation for being insular and mysterious. But the word “Zope” is literally quite meaningless without qualification. What *part* of Zope do you hate? “Zope” is a brand, not a technology.

If it’s Zope2-the-web-framework, Pyramid is not that. The primary designers and developers of Pyramid, if anyone, should know. We wrote Pyramid’s predecessor (`repoze.bfg`), in part, because *we* knew that Zope 2 had usability issues and limitations. `repoze.bfg` (and now Pyramid) was written to address these issues.

If it’s Zope3-the-web-framework, Pyramid is *definitely* not that. Making use of lots of Zope 3 technologies is territory already staked out by the *Grok* project. Save for the obvious fact that they’re both web frameworks, Pyramid is very, very different than Grok. Grok exposes lots of Zope technologies to end users. On the other hand, if you need to understand a Zope-only concept while using Pyramid, then we’ve failed on some very basic axis.

If it’s just the word Zope: this can only be guilt by association. Because a piece of software internally uses some package named `zope.foo`, it doesn’t turn the piece of software that uses it into “Zope”. There is a lot of *great* software written that has the word Zope in its name. Zope is not some sort of monolithic thing, and a lot of its software is usable externally. And while it’s not really the job of this document to defend it, Zope has been around for over 10 years and has an incredibly large, active community. If you don’t believe this, <http://pypi-ranking.info/author> is an eye-opening reality check.

Love Simplicity

Years of effort have gone into honing this package and its documentation to make it as simple as humanly possible for developers to use. Everything is a tradeoff, of course, and people have their own ideas about what “simple” is. You may have a style difference if you believe Pyramid is complex. Its developers obviously disagree.

Other Challenges

Other challenges are encouraged to be sent to the Pylons-devel maillist. We’ll try to address them by considering a design change, or at very least via exposition here.

Tutorials

Quick Tour of Pyramid

Pyramid lets you start small and finish big. This *Quick Tour* of Pyramid is for those who want to evaluate Pyramid, whether you are new to Python web frameworks, or a pro in a hurry. For more detailed treatment of each topic, give the *Quick Tutorial for Pyramid* a try.

Installation

Once you have a standard Python environment setup, getting started with Pyramid is a breeze. Unfortunately “standard” is not so simple in Python. For this Quick Tour, it means Python, venv (or virtualenv for Python 2.7), pip, and setuptools.

To save a little bit of typing and to be certain that we use the modules, scripts, and packages installed in our virtual environment, we’ll set an environment variable, too.

As an example, for Python 3.5+ on Linux:

```
# set an environment variable to where you want your virtual_
→environment
$ export VENV=~/.env
# create the virtual environment
$ python3 -m venv $VENV
# install pyramid
$ $VENV/bin/pip install pyramid
# or for a specific released version
$ $VENV/bin/pip install "pyramid==1.7.6"
```

For Windows:

```
# set an environment variable to where you want your virtual_
↳environment
c:\> set VENV=c:\env
# create the virtual environment
c:\> python -m venv %VENV%
# install pyramid
c:\> %VENV%\Scripts\pip install pyramid
# or for a specific released version
c:\> %VENV%\Scripts\pip install "pyramid==1.7.6"
```

Of course Pyramid runs fine on Python 2.7+, as do the examples in this *Quick Tour*. We're showing Python 3 for simplicity. (Pyramid had production support for Python 3 in October 2011.) Also for simplicity, the remaining examples will show only UNIX commands.

See also:

See also: *Quick Tutorial section on Requirements, Installing Pyramid on a UNIX System, Before You Install, Why use \$VENV/bin/pip instead of source bin/activate, then pip, and Installing Pyramid on a Windows System.*

Hello World

Microframeworks have shown that learning starts best from a very small first step. Here's a tiny application in Pyramid:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5
6 def hello_world(request):
7     return Response('<h1>Hello World!</h1>')
8
9
10 if __name__ == '__main__':
11     config = Configurator()
12     config.add_route('hello', '/')
13     config.add_view(hello_world, route_name='hello')
14     app = config.make_wsgi_app()
15     server = make_server('0.0.0.0', 6543, app)
16     server.serve_forever()
```

This simple example is easy to run. Save this as `app.py` and run it:

```
$ $VENV/bin/python ./app.py
```

Next open `http://localhost:6543/` in a browser, and you will see the `Hello World!` message.

New to Python web programming? If so, some lines in the module merit explanation:

1. *Line 10.* `if __name__ == '__main__':` is Python’s way of saying “Start here when running from the command line”.
2. *Lines 11-13.* Use Pyramid’s *configurator* to connect *view* code to a particular URL *route*.
3. *Lines 6-7.* Implement the view code that generates the *response*.
4. *Lines 14-16.* Publish a *WSGI* app using an HTTP server.

As shown in this example, the *configurator* plays a central role in Pyramid development. Building an application from loosely-coupled parts via *Application Configuration* is a central idea in Pyramid, one that we will revisit regularly in this *Quick Tour*.

See also:

See also: *Quick Tutorial Hello World*, *Creating Your First Pyramid Application*, and *Todo List Application in One File*.

Handling web requests and responses

Developing for the web means processing web requests. As this is a critical part of a web application, web developers need a robust, mature set of software for web requests.

Pyramid has always fit nicely into the existing world of Python web development (virtual environments, packaging, scaffolding, one of the first to embrace Python 3, etc.). Pyramid turned to the well-regarded *WebOb* Python library for request and response handling. In our example above, Pyramid hands `hello_world` a request that is *based on WebOb*.

Let’s see some features of requests and responses in action:

```
def hello_world(request):  
    # Some parameters from a request such as /?name=lisa  
    url = request.url  
    name = request.params.get('name', 'No Name Provided')  
  
    body = 'URL %s with name: %s' % (url, name)  
    return Response(  
        content_type="text/plain",  
        body=body  
    )
```

In this Pyramid view, we get the URL being visited from `request.url`. Also if you visited `http://localhost:6543/?name=alice` in a browser, the name is included in the body of the response:

```
URL http://localhost:6543/?name=alice with name: alice
```

Finally we set the response’s content type, and return the `Response`.

See also:

See also: *Quick Tutorial Request and Response* and *Request and Response Objects*.

Views

For the examples above, the `hello_world` function is a “view”. In Pyramid views are the primary way to accept web requests and return responses.

So far our examples place everything in one file:

- the view function
- its registration with the configurator
- the route to map it to an URL
- the WSGI application launcher

Let’s move the views out to their own `views.py` module and change the `app.py` to scan that module, looking for decorators that set up the views.

First our revised `app.py`:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3
4 if __name__ == '__main__':
5     config = Configurator()
6     config.add_route('home', '/')
7     config.add_route('hello', '/howdy')
8     config.add_route('redirect', '/goto')
9     config.add_route('exception', '/problem')
10    config.scan('views')
11    app = config.make_wsgi_app()
12    server = make_server('0.0.0.0', 6543, app)
13    server.serve_forever()
```

We added some more routes, but we also removed the view code. Our views and their registrations (via decorators) are now in a module `views.py`, which is scanned via `config.scan('views')`.

We now have a `views.py` module that is focused on handling requests and responses:

```
1 import cgi
2
3 from pyramid.httpexceptions import HTTPFound
4 from pyramid.response import Response
5 from pyramid.view import view_config
6
7
8 # First view, available at http://localhost:6543/
9 @view_config(route_name='home')
10 def home_view(request):
11     return Response('<p>Visit <a href="/howdy?name=lisa">hello</a></p>')
12
13
14 # /howdy?name=alice which links to the next view
15 @view_config(route_name='hello')
16 def hello_view(request):
17     name = request.params.get('name', 'No Name')
18     body = '<p>Hi %s, this <a href="/goto">redirects</a></p>'
19     # cgi.escape to prevent Cross-Site Scripting (XSS) [CWE 79]
20     return Response(body % cgi.escape(name))
21
22
23 # /goto which issues HTTP redirect to the last view
24 @view_config(route_name='redirect')
25 def redirect_view(request):
26     return HTTPFound(location="/problem")
```

```
27 |
28 |
29 | # /problem which causes a site error
30 | @view_config(route_name='exception')
31 | def exception_view(request):
32 |     raise Exception()
```

We have four views, each leading to the other. If you start at `http://localhost:6543/`, you get a response with a link to the next view. The `hello_view` (available at the URL `/howdy`) has a link to the `redirect_view`, which issues a redirect to the final view.

Earlier we saw `config.add_view` as one way to configure a view. This section introduces `@view_config`. Pyramid's configuration supports *imperative configuration*, such as the `config.add_view` in the previous example. You can also use *declarative configuration* in which a Python *decorator* is placed on the line above the view. Both approaches result in the same final configuration, thus usually it is simply a matter of taste.

See also:

See also: *Quick Tutorial Views*, *Views*, *View Configuration*, and *Debugging View Configuration*.

Routing

Writing web applications usually means sophisticated URL design. We just saw some Pyramid machinery for requests and views. Let's look at features that help with routing.

Above we saw the basics of routing URLs to views in Pyramid:

- Your project's "setup" code registers a route name to be used when matching part of the URL.
- Elsewhere a view is configured to be called for that route name.

i Why do this twice? Other Python web frameworks let you create a route and associate it with a view in one step. As illustrated in *Routes need relative ordering*, multiple routes might match the same URL pattern. Rather than provide ways to help guess, Pyramid lets you be explicit in ordering. Pyramid also gives facilities to avoid the problem.

What if we want part of the URL to be available as data in my view? We can use this route declaration, for example:

```
6 config.add_route('hello', '/howdy/{first}/{last}')
```

With this, URLs such as `/howdy/amy/smith` will assign `amy` to `first` and `smith` to `last`. We can then use this data in our view:

```
5 @view_config(route_name='hello')
6 def hello_world(request):
7     body = '<h1>Hi %(first)s %(last)s!</h1>' % request.matchdict
8     return Response(body)
```

`request.matchdict` contains values from the URL that match the “replacement patterns” (the curly braces) in the route declaration. This information can then be used in your view.

See also:

See also: *Quick Tutorial Routing*, *URL Dispatch*, *Debugging Route Matching*, and *Request Processing*.

Templating

Ouch. We have been making our own `Response` and filling the response body with HTML. You usually won’t embed an HTML string directly in Python, but instead you will use a templating language.

Pyramid doesn’t mandate a particular database system, form library, and so on. It encourages replaceability. This applies equally to templating, which is fortunate: developers have strong views about template languages. That said, the Pylons Project officially supports bindings for Chameleon, Jinja2, and Mako. In this step let’s use Chameleon.

Let’s add `pyramid_chameleon`, a Pyramid *add-on* which enables Chameleon as a *renderer* in our Pyramid application:

```
$ $VENV/bin/pip install pyramid_chameleon
```

With the package installed, we can include the template bindings into our configuration in `app.py`:

```
6 config.add_route('hello', '/howdy/{name}')
7 config.include('pyramid_chameleon')
8 config.scan('views')
```

Now let’s change our `views.py` file:

```
1 from pyramid.view import view_config
2
3
4 @view_config(route_name='hello', renderer='hello_world.pt')
5 def hello_world(request):
6     return dict(name=request.matchdict['name'])
```

Ahh, that looks better. We have a view that is focused on Python code. Our `@view_config` decorator specifies a *renderer* that points to our template file. Our view then simply returns data which is then supplied to our template `hello_world.pt`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Quick Glance</title>
</head>
<body>
<h1>Hello ${name}</h1>
</body>
</html>
```

Since our view returned `dict(name=request.matchdict['name'])`, we can use `name` as a variable in our template via `${name}`.

See also:

See also: *Quick Tutorial Templating*, *Templates*, *Debugging Templates*, and *Available Add-On Template System Bindings*.

Templating with Jinja2

We just said Pyramid doesn't prefer one templating language over another. Time to prove it. Jinja2 is a popular templating system, modeled after Django's templates. Let's add `pyramid_jinja2`, a Pyramid *add-on* which enables Jinja2 as a *renderer* in our Pyramid applications:

```
$ $VENV/bin/pip install pyramid_jinja2
```

With the package installed, we can include the template bindings into our configuration:


```
6 config.add_route('hello', '/howdy/{name}')
7 config.include('pyramid_jinja2')
8 config.scan('views')
```

The only change in our view is to point the renderer at the `.jinja2` file:

```
4 @view_config(route_name='hello', renderer='hello_world.jinja2')
5 def hello_world(request):
6     return dict(name=request.matchdict['name'])
```

Our Jinja2 template is very similar to our previous template:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Hello World</title>
</head>
<body>
<h1>Hello {{ name }}!</h1>
</body>
</html>
```

Pyramid's templating add-ons register a new kind of renderer into your application. The renderer registration maps to different kinds of filename extensions. In this case, changing the extension from `.pt` to `.jinja2` passed the view response through the `pyramid_jinja2` renderer.

See also:

See also: *Quick Tutorial Jinja2*, Jinja2 homepage, and `pyramid_jinja2` Overview.

Static assets

Of course the Web is more than just markup. You need static assets: CSS, JS, and images. Let's point our web app at a directory from which Pyramid will serve some static assets. First let's make another call to the *configurator* in `app.py`:

```
6 config.add_route('hello', '/howdy/{name}')
7 config.add_static_view(name='static', path='static')
8 config.include('pyramid_jinja2')
```

This tells our WSGI application to map requests under `http://localhost:6543/static/` to files and directories inside a `static` directory alongside our Python module.

Next make a directory named `static`, and place `app.css` inside:

```
body {  
    margin: 2em;  
    font-family: sans-serif;  
}
```

All we need to do now is point to it in the `<head>` of our Jinja2 template, `hello_world.jinja2`:

```
4     <title>Hello World</title>  
5     <link rel="stylesheet" href="/static/app.css"/>  
6 </head>
```

This link presumes that our CSS is at a URL starting with `/static/`. What if the site is later moved under `/somesite/static/`? Or perhaps a web developer changes the arrangement on disk? Pyramid provides a helper to allow flexibility on URL generation:

```
4     <title>Hello World</title>  
5     <link rel="stylesheet" href="{ request.static_url('__main__:static/  
6     ↪app.css') }}" />  
6 </head>
```

By using `request.static_url` to generate the full URL to the static assets, you ensure that you stay in sync with the configuration and gain refactoring flexibility later.

See also:

See also: *Quick Tutorial Static Assets*, *Static Assets*, *Preventing HTTP Caching*, and *Influencing HTTP Caching*.

Returning JSON

Modern web apps are more than rendered HTML. Dynamic pages now use JavaScript to update the UI in the browser by requesting server data as JSON. Pyramid supports this with a JSON renderer:

```
9 @view_config(route_name='hello_json', renderer='json')  
10 def hello_json(request):  
11     return [1, 2, 3]
```

This wires up a view that returns some data through the JSON *renderer*, which calls Python’s JSON support to serialize the data into JSON, and sets the appropriate HTTP headers.

We also need to add a route to `app.py` so that our app will know how to respond to a request for `hello.json`.

```
6 config.add_route('hello', '/howdy/{name}')
7 config.add_route('hello_json', 'hello.json')
8 config.add_static_view(name='static', path='static')
```

See also:

See also: *Quick Tutorial JSON*, *Writing View Callables Which Use a Renderer*, *JSON Renderer*, and *Adding and Changing Renderers*.

View classes

So far our views have been simple, free-standing functions. Many times your views are related. They may have different ways to look at or work on the same data, or they may be a REST API that handles multiple operations. Grouping these together as a *view class* makes sense and achieves the following goals.

- Group views
- Centralize some repetitive defaults
- Share some state and helpers

The following shows a “Hello World” example with three operations: view a form, save a change, or press the delete button in our `views.py`:

```
7 # One route, at /howdy/amy, so don't repeat on each @view_config
8 @view_defaults(route_name='hello')
9 class HelloWorldViews:
10     def __init__(self, request):
11         self.request = request
12         # Our templates can now say {{ view.name }}
13         self.name = request.matchdict['name']
14
15     # Retrieving /howdy/amy the first time
16     @view_config(renderer='hello.jinja2')
17     def hello_view(self):
18         return dict()
```

```

19
20     # Posting to /howdy/amy via the "Edit" submit button
21     @view_config(request_param='form.edit', renderer='edit.jinja2')
22     def edit_view(self):
23         print('Edited')
24         return dict()
25
26     # Posting to /howdy/amy via the "Delete" submit button
27     @view_config(request_param='form.delete', renderer='delete.jinja2')
28     def delete_view(self):
29         print('Deleted')
30         return dict()

```

As you can see, the three views are logically grouped together. Specifically:

- The first view is returned when you go to /howdy/amy. This URL is mapped to the `hello` route that we centrally set using the optional `@view_defaults`.
- The second view is returned when the form data contains a field with `form.edit`, such as clicking on `<input type="submit" name="form.edit" value="Save">`. This rule is specified in the `@view_config` for that view.
- The third view is returned when clicking on a button such as `<input type="submit" name="form.delete" value="Delete">`.

Only one route is needed, stated in one place atop the view class. Also, the assignment of name is done in the `__init__` function. Our templates can then use `{{ view.name }}`.

Pyramid view classes, combined with built-in and custom predicates, have much more to offer:

- All the same view configuration parameters as function views
- One route leading to multiple views, based on information in the request or data such as `request_param`, `request_method`, `accept`, `header`, `xhr`, `containment`, and `custom_predicates`

See also:

See also: *Quick Tutorial View Classes*, *Quick Tutorial More View Classes*, and *Defining a View Callable as a Class*.

Quick project startup with scaffolds

So far we have done all of our *Quick Tour* as a single Python file. No Python packages, no structure. Most Pyramid projects, though, aren't developed this way.

To ease the process of getting started, Pyramid provides *scaffolds* that generate sample projects from templates in Pyramid and Pyramid add-ons. Pyramid's `pcreate` command can list the available scaffolds:

```
$ $VENV/bin/pcreate --list
Available scaffolds:
  alchemy:                Pyramid project using SQLAlchemy, SQLite, URL_
↪dispatch, and Jinja2
  pyramid_jinja2_starter:  Pyramid Jinja2 starter project
  starter:                Pyramid starter project using URL dispatch and_
↪Chameleon
  zodb:                   Pyramid project using ZODB, traversal, and_
↪Chameleon
```

The `pyramid_jinja2` add-on gave us a scaffold that we can use. From the parent directory of where we want our Python package to be generated, let's use that scaffold to make our project:

```
$ $VENV/bin/pcreate --scaffold pyramid_jinja2_starter hello_world
```

We next use the normal Python command to set up our package for development:

```
$ cd hello_world
$ $VENV/bin/pip install -e .
```

We are moving in the direction of a full-featured Pyramid project, with a proper setup for Python standards (packaging) and Pyramid configuration. This includes a new way of running your application:

```
$ $VENV/bin/pserve development.ini
```

Let's look at `pserve` and configuration in more depth.

See also:

See also: *Quick Tutorial Scaffolds*, *Creating a Pyramid Project*, and *Creating Pyramid Scaffolds*

Application running with `pserve`

Prior to scaffolds, our project mixed a number of operational details into our code. Why should my main code care which HTTP server I want and what port number to run on?

`pserve` is Pyramid's application runner, separating operational details from your code. When you install Pyramid, a small command program called `pserve` is written to your `bin` directory. This program is an executable Python module. It's very small, getting most of its brains via `import`.

You can run `pserve` with `--help` to see some of its options. Doing so reveals that you can ask `pserve` to watch your development files and reload the server when they change:

```
$ $VENV/bin/pserve development.ini --reload
```

The `pserve` command has a number of other options and operations. Most of the work, though, comes from your project's wiring, as expressed in the configuration file you supply to `pserve`. Let's take a look at this configuration file.

See also:

See also: *What Is This pserve Thing*

Configuration with `.ini` files

Earlier in *Quick Tour* we first met Pyramid's configuration system. At that point we did all configuration in Python code. For example, the port number chosen for our HTTP server was right there in Python code. Our scaffold has moved this decision and more into the `development.ini` file:

```
###
# app configuration
# http://docs.pylonsproject.org/projects/pyramid/en/1.6-branch/narr/
# → environment.html
###

[app:main]
use = egg:hello_world

pyramid.reload_templates = true
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.debug_routematch = false
```

```
pyramid.debug_templates = true
pyramid.default_locale_name = en
pyramid.includes =
    pyramid_debugtoolbar

# By default, the toolbar only appears for clients from IP addresses
# '127.0.0.1' and '::1'.
# debugtoolbar.hosts = 127.0.0.1 ::1

###
# wsgi server configuration
###

[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 6543

###
# logging configuration
# http://docs.pylonsproject.org/projects/pyramid/en/1.6-branch/narr/
↳ logging.html
###

[loggers]
keys = root, hello_world

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

[logger_hello_world]
level = DEBUG
handlers =
qualname = hello_world

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
```

```

formatter = generic

[formatter_generic]
format = %(asctime)s %(levelname)-5.5s [% (name)s: %(lineno)s] [
↳ %(threadName)s] %(message)s

```

Let's take a quick high-level look. First the `.ini` file is divided into sections:

- `[app:main]` configures our WSGI app
- `[server:main]` holds our WSGI server settings
- Various sections afterwards configure our Python logging system

We have a few decisions made for us in this configuration:

1. *Choice of web server:* `use = egg:hello_world` tells `pserve` to use the waitress server.
2. *Port number:* `port = 6543` tells waitress to listen on port 6543.
3. *WSGI app:* What package has our WSGI application in it? `use = egg:hello_world` in the app section tells the configuration what application to load.
4. *Easier development by automatic template reloading:* In development mode, you shouldn't have to restart the server when editing a Jinja2 template. `pyramid.reload_templates = true` sets this policy, which might be different in production.

Additionally the `development.ini` generated by this scaffold wired up Python's standard logging. We'll now see in the console, for example, a log on every request that comes in, as well as traceback information.

See also:

See also: *Quick Tutorial Application Configuration, Environment Variables and .ini File Settings* and *PasteDeploy Configuration Files*

Easier development with debugtoolbar

As we introduce the basics, we also want to show how to be productive in development and debugging. For example, we just discussed template reloading and earlier we showed `--reload` for application reloading.

`pyramid_debugtoolbar` is a popular Pyramid add-on which makes several tools available in your browser. Adding it to your project illustrates several points about configuration.

The scaffold `pyramid_jinja2_starter` is already configured to include the add-on `pyramid_debugtoolbar` in its `setup.py`:


```
11 requires = [  
12     'pyramid',  
13     'pyramid_jinja2',  
14     'pyramid_debugtoolbar',  
15     'waitress',  
16 ]
```

It was installed when you previously ran:

```
$ $VENV/bin/pip install -e .
```

The `pyramid_debugtoolbar` package is a Pyramid add-on, which means we need to include its configuration into our web application. The `pyramid_jinja2` add-on already took care of this for us in its `__init__.py`:

```
16 config.include('pyramid_jinja2')
```

And it uses the `pyramid.includes` facility in our `development.ini`:

```
15 pyramid.includes =  
16     pyramid_debugtoolbar
```

You'll now see a Pyramid logo on the right side of your browser window, which when clicked opens a new window that provides introspective access to debugging information. Even better, if your web application generates an error, you will see a nice traceback on the screen. When you want to disable this toolbar, there's no need to change code: you can remove it from `pyramid.includes` in the relevant `.ini` configuration file.

See also:

See also: *Quick Tutorial* `pyramid_debugtoolbar` and `pyramid_debugtoolbar`

Unit tests and `py.test`

Yikes! We got this far and we haven't yet discussed tests. This is particularly egregious, as Pyramid has had a deep commitment to full test coverage since before its release.

Our `pyramid_jinja2_starter` scaffold generated a `tests.py` module with one unit test in it. It also configured `setup.py` with test requirements: `py.test` as the test runner, `WebTest` for running view tests, and the `pytest-cov` tool which yells at us for code that isn't tested. The highlighted lines show this:

```

11 requires = [
12     'pyramid',
13     'pyramid_jinja2',
14     'pyramid_debugtoolbar',
15     'waitress',
16 ]
17
18 tests_require = [
19     'WebTest >= 1.3.1', # py3 compat
20     'pytest', # includes virtualenv
21     'pytest-cov',
22 ]

```

```

34     zip_safe=False,
35     extras_require={
36         'testing': tests_require,
37     },

```

To install the test requirements, run `$VENV/bin/pip install -e ".[testing]"`. We can now run all our tests:

```
$ $VENV/bin/py.test --cov --cov-report=term-missing
```

This yields the following output.

```

===== test session starts =====
platform darwin -- Python 3.5.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: /Users/stevepiercy/projects/hack-on-pyramid/hello_world, inifile:
plugins: cov-2.2.1
collected 1 items

hello_world/tests.py .
----- coverage: platform darwin, python 3.5.0-final-0 -----
Name                               Stmts   Miss  Cover   Missing
-----
hello_world/__init__.py             11      8    27%    11-23
hello_world/resources.py              5      1    80%      8
hello_world/tests.py                14      0   100%
hello_world/views.py                 4      0   100%
-----
TOTAL                               34      9    74%

```

```
===== 1 passed in 0.22 seconds_
↩=====
```

Our unit test passed, although its coverage is incomplete. What did our test look like?

```
1 import unittest
2 from pyramid import testing
3 from pyramid.i18n import TranslationStringFactory
4
5 _ = TranslationStringFactory('hello_world')
6
7
8 class ViewTests(unittest.TestCase):
9
10     def setUp(self):
11         testing.setUp()
12
13     def tearDown(self):
14         testing.tearDown()
15
16     def test_my_view(self):
17         from hello_world.views import my_view
18         request = testing.DummyRequest()
19         response = my_view(request)
20         self.assertEqual(response['project'], 'hello_world')
```

Pyramid supplies helpers for test writing, which we use in the test setup and teardown. Our one test imports the view, makes a dummy request, and sees if the view returns what we expected.

See also:

See also: *Quick Tutorial Unit Testing*, *Quick Tutorial Functional Testing*, and *Unit, Integration, and Functional Testing*

Logging

It's important to know what is going on inside our web application. In development we might need to collect some output. In production we might need to detect situations when other people use the site. We need *logging*.

Fortunately Pyramid uses the normal Python approach to logging. The scaffold generated in your `development.ini` has a number of lines that configure the logging for you to some reasonable defaults. You then see messages sent by Pyramid (for example, when a new request comes in).

Maybe you would like to log messages in your code? In your Python module, import and set up the logging:

```
3 import logging
4 log = logging.getLogger(__name__)
```

You can now, in your code, log messages:

```
9 def my_view(request):
10     log.debug('Some Message')
```

This will log `Some Message` at a debug log level to the application-configured logger in your `development.ini`. What controls that? These emphasized sections in the configuration file:

```
36 [loggers]
37 keys = root, hello_world
38
39 [handlers]
40 keys = console
41
42 [formatters]
43 keys = generic
44
45 [logger_root]
46 level = INFO
47 handlers = console
48
49 [logger_hello_world]
50 level = DEBUG
51 handlers =
52 qualname = hello_world
```

Our application, a package named `hello_world`, is set up as a logger and configured to log messages at a `DEBUG` or higher level. When you visit `http://localhost:6543`, your console will now show:

```
2016-01-18 13:55:55,040 DEBUG [hello_world.views:10][waitress] Some Message
```

See also:

See also: *Quick Tutorial Logging* and *Logging*.

Sessions

When people use your web application, they frequently perform a task that requires semi-permanent data to be saved. For example, a shopping cart. This is called a *session*.

Pyramid has basic built-in support for sessions. Third party packages such as `pyramid_redis_sessions` provide richer session support. Or you can create your own custom sessioning engine. Let's take a look at the *built-in sessioning support*. In our `__init__.py` we first import the kind of sessioning we want:

```
2 from hello_world.resources import get_root
3 from pyramid.session import SignedCookieSessionFactory
```



As noted in the session docs, this example implementation is not intended for use in settings with security implications.

Now make a “factory” and pass it to the *configurator*’s `session_factory` argument:

```
13 settings.setdefault('jinja2.i18n.domain', 'hello_world')
14
15 my_session_factory = SignedCookieSessionFactory('itsaseekreet')
16 config = Configurator(root_factory=get_root, settings=settings,
17                       session_factory=my_session_factory)
```

Pyramid’s *request* object now has a `session` attribute that we can use in our view code in `views.py`:

```
9 def my_view(request):
10     log.debug('Some Message')
11     session = request.session
12     if 'counter' in session:
13         session['counter'] += 1
14     else:
15         session['counter'] = 0
```

We need to update our Jinja2 template to show counter increment in the session:

```
40         <p class="lead">
41             {% trans %}Hello{% endtrans %} to <span class="font-normal
42             ↪">{{project}}</span>, an&nbsp;application generated&nbsp;by<br>the <span_
43             ↪class="font-normal">Pyramid Web Framework 1.6</span>.</p>
44             <p>Counter: {{ request.session.counter }}</p>
```

See also:

See also: *Quick Tutorial Sessions*, *Sessions*, *Flash Messages*, *pyramid.session*, and *pyramid_redis_sessions*.

Databases

Web applications mean data. Data means databases. Frequently SQL databases. SQL databases frequently mean an “ORM” (object-relational mapper.) In Python, ORM usually leads to the mega-quality *SQLAlchemy*, a Python package that greatly eases working with databases.

Pyramid and SQLAlchemy are great friends. That friendship includes a scaffold!

```
$ $VENV/bin/pcreate --scaffold alchemy sqla_demo
$ cd sqla_demo
$ $VENV/bin/pip install -e .
```

We now have a working sample SQLAlchemy application with all dependencies installed. The sample project provides a console script to initialize a SQLite database with tables. Let’s run it, then start the application:

```
$ $VENV/bin/initialize_sqla_demo_db development.ini
$ $VENV/bin/pserve development.ini
```

The ORM eases the mapping of database structures into a programming language. SQLAlchemy uses “models” for this mapping. The scaffold generated a sample model:

```
class MyModel(Base):
    __tablename__ = 'models'
    id = Column(Integer, primary_key=True)
    name = Column(Text)
    value = Column(Integer)
```

View code, which mediates the logic between web requests and the rest of the system, can then easily get at the data thanks to SQLAlchemy:

```
one = query.filter(MyModel.name == 'one').first()
```

See also:

See also: *Quick Tutorial Databases*, *SQLAlchemy*, *Making Your Script into a Console Script*, *SQLAlchemy + URL dispatch wiki tutorial*, and *Application Transactions with pyramid_tm*.

Forms

Developers have lots of opinions about web forms, thus there are many form libraries for Python. Pyramid doesn't directly bundle a form library, but *Deform* is a popular choice for forms, along with its related *Colander* schema system.

As an example, imagine we want a form that edits a wiki page. The form should have two fields on it, one of them a required title and the other a rich text editor for the body. With Deform we can express this as a Colander schema:

```
class WikiPage(colander.MappingSchema):
    title = colander.SchemaNode(colander.String())
    body = colander.SchemaNode(
        colander.String(),
        widget=deform.widget.RichTextWidget()
    )
```

With this in place, we can render the HTML for a form, perhaps with form data from an existing page:

```
form = self.wiki_form.render()
```

We'd like to handle form submission, validation, and saving:

```
# Get the form data that was posted
controls = self.request.POST.items()
try:
    # Validate and either raise a validation error
    # or return deserialized data from widgets
    appstruct = wiki_form.validate(controls)
except deform.ValidationFailure as e:
    # Bail out and render form with errors
    return dict(title=title, page=page, form=e.render())

# Change the content and redirect to the view
page['title'] = appstruct['title']
page['body'] = appstruct['body']
```

Deform and Colander provide a very flexible combination for forms, widgets, schemas, and validation. Recent versions of Deform also include a retail mode for gaining Deform features on custom forms.

Also the `deform_bootstrap` Pyramid add-on restyles the stock Deform widgets using attractive CSS from Twitter Bootstrap and more powerful widgets from Chosen.

See also:

See also: *Quick Tutorial Forms*, *Deform*, *Colander*, and *deform_bootstrap*.

Conclusion

This *Quick Tour* covered a little about a lot. We introduced a long list of concepts in Pyramid, many of which are expanded on more fully in the Pyramid developer docs.

Quick Tutorial for Pyramid


Pyramid is a web framework for Python 2 and 3. This tutorial gives a Python 3/2-compatible, high-level tour of the major features.

This hands-on tutorial covers “a little about a lot”: practical introductions to the most common facilities. Fun, fast-paced, and most certainly not aimed at experts of the Pyramid web framework.

Contents

Requirements

Let’s get our tutorial environment set up. Most of the set up work is in standard Python development practices (install Python and make an isolated virtual environment.)

 Pyramid encourages standard Python development practices with packaging tools, virtual environments, logging, and so on. There are many variations, implementations, and opinions across the Python community. For consistency, ease of documentation maintenance, and to minimize confusion, the Pyramid *documentation* has adopted specific conventions that are consistent with the *Python Packaging Authority*.

This *Quick Tutorial* is based on:

- **Python 3.5.** Pyramid fully supports Python 3.3+ and Python 2.7+. This tutorial uses **Python 3.5** but runs fine under Python 2.7.
- **venv.** We believe in virtual environments. For this tutorial, we use Python 3.5’s built-in solution *venv*. For Python 2.7, you can install *virtualenv*.
- **pip.** We use *pip* for package management.

- **Workspaces, projects, and packages.** Our home directory will contain a *tutorial workspace* with our Python virtual environment and *Python projects* (a directory with packaging information and *Python packages* of working code.)
- **Unix commands.** Commands in this tutorial use UNIX syntax and paths. Windows users should adjust commands accordingly.



Pyramid was one of the first web frameworks to fully support Python 3 in October 2011.



Windows commands use the plain old MSDOS shell. For PowerShell command syntax, see its documentation.

Steps

1. *Install Python 3*
2. *Create a project directory structure*
3. *Set an environment variable*
4. *Create a virtual environment*
5. *Install Pyramid*

Install Python 3

See the detailed recommendation for your operating system described under *Installing Pyramid*.

- *For Mac OS X Users*
- *If You Don't Yet Have a Python Interpreter (UNIX)*
- *If You Don't Yet Have a Python Interpreter (Windows)*

Create a project directory structure

We will arrive at a directory structure of `workspace -> project -> package`, where our workspace is named `quick_tutorial`. The following tree diagram shows how this will be structured, and where our *virtual environment* will reside as we proceed through the tutorial:

```
`- ~
  `-- projects
      `-- quick_tutorial
          |-- env
          `-- step_one
              |-- intro
              |   |-- __init__.py
              |   `-- app.py
              `-- setup.py
```

For Linux, the commands to do so are as follows:

```
# Mac and Linux
$ cd ~
$ mkdir -p projects/quick_tutorial
$ cd projects/quick_tutorial
```

For Windows:

```
# Windows
c:\> cd \
c:\> mkdir projects\quick_tutorial
c:\> cd projects\quick_tutorial
```

In the above figure, your user home directory is represented by `~`. In your home directory, all of your projects are in the `projects` directory. This is a general convention not specific to Pyramid that many developers use. Windows users will do well to use `c:\` as the location for `projects` in order to avoid spaces in any of the path names.

Next within `projects` is your workspace directory, here named `quick_tutorial`. A workspace is a common term used by integrated development environments (IDE), like PyCharm and PyDev, where virtual environments, specific project files, and repositories are stored.

Set an environment variable

This tutorial will refer frequently to the location of the *virtual environment*. We set an environment variable to save typing later.

```
# Mac and Linux
$ export VENV=~/.projects/quick_tutorial/env
```

```
# Windows
c:\> set VENV=c:\projects\quick_tutorial\env
```

Create a virtual environment

`venv` is a tool to create isolated Python 3 environments, each with its own Python binary and independent set of installed Python packages in its site directories. Let's create one, using the location we just specified in the environment variable.

```
# Mac and Linux
$ python3 -m venv $VENV
```

```
# Windows
c:\> c:\Python35\python -m venv %VENV%
```

See also:

See also Python 3's `venv` module and Python 2's `virtualenv` package.

Update packaging tools in the virtual environment

It's always a good idea to update to the very latest version of packaging tools because the installed Python bundles only the version that was available at the time of its release.

```
# Mac and Linux
$VENV/bin/pip install --upgrade pip setuptools
```

```
# Windows
c:\> %VENV%\Scripts\pip install --upgrade pip setuptools
```

See also:

See also *Why use `$VENV/bin/pip` instead of `source bin/activate`, then `pip`.*

Install Pyramid

We have our Python standard prerequisites out of the way. The Pyramid part is pretty easy.

```
# Mac and Linux
$ $VENV/bin/pip install "pyramid==1.7.6"

# Windows
c:\> %VENV%\Scripts\pip install "pyramid==1.7.6"
```

Our Python virtual environment now has the Pyramid software available.

You can optionally install some of the extra Python packages used in this tutorial.

```
# Mac and Linux
$ $VENV/bin/pip install webtest pytest pytest-cov deform sqlalchemy \
  pyramid_chameleon pyramid_debugtoolbar pyramid_jinja2 waitress \
  pyramid_tm zope.sqlalchemy
```

```
# Windows
c:\> %VENV%\Scripts\pip install webtest deform sqlalchemy pyramid_
  ↳chameleon pyramid_debugtoolbar pyramid_jinja2 waitress pyramid_tm zope.
  ↳sqlalchemy
```

Tutorial Approach

This tutorial uses conventions to keep the introduction focused and concise. Details, references, and deeper discussions are mentioned in “See also” notes.

See also:

This is an example “See also” note.

This “Getting Started” tutorial is broken into independent steps, starting with the smallest possible “single file WSGI app” example. Each of these steps introduce a topic and a very small set of concepts via working code. The steps each correspond to a directory in this repo, where each step/topic/directory is a Python package.

To successfully run each step:

```
$ cd request_response
$ $VENV/bin/pip install -e .
```

...and repeat for each step you would like to work on. In most cases we will start with the results of an earlier step.

Directory tree

As we develop our tutorial, our directory tree will resemble the structure below:

```
quick_tutorial
├── env
├── request_response
│   └── tutorial
│       ├── __init__.py
│       ├── tests.py
│       └── views.py
├── development.ini
└── setup.py
```

Each of the first-level directories (e.g., `request_response`) is a *Python project* (except as noted for the `hello_world` step). The `tutorial` directory is a *Python package*. At the end of each step, we copy a previous directory into a new directory to use as a starting point.

Prelude: Quick Project Startup with Scaffolds

To ease the process of getting started, Pyramid provides *scaffolds* that generate sample projects from templates in Pyramid and Pyramid add-ons.

Background

We’re going to cover a lot in this tutorial, focusing on one topic at a time and writing everything from scratch. As a warm up, though, it sure would be nice to see some pixels on a screen.

Like other web development frameworks, Pyramid provides a number of “scaffolds” that generate working Python, template, and CSS code for sample applications. In this step we’ll use a built-in scaffold to let us preview a Pyramid application, before starting from scratch on Step 1.

Objectives

- Use Pyramid's `pcreate` command to list scaffolds and make a new project.
- Start up a Pyramid application and visit it in a web browser.

Steps

1. Pyramid's `pcreate` command can list the available scaffolds:

```
$ $VENV/bin/pcreate --list
Available scaffolds:
  alchemy: Pyramid project using SQLAlchemy, SQLite, URL_
  ↳dispatch, and Jinja2
  starter: Pyramid starter project using URL dispatch and_
  ↳Chameleon
  zodb:    Pyramid project using ZODB, traversal, and_
  ↳Chameleon
```

2. Tell `pcreate` to use the starter scaffold to make our project:

```
$ $VENV/bin/pcreate --scaffold starter scaffolds
```

3. Install our project in editable mode for development in the current directory:

```
$ cd scaffolds
$ $VENV/bin/pip install -e .
```

4. Start up the application by pointing Pyramid's `pserve` command at the project's (generated) configuration file:

```
$ $VENV/bin/pserve development.ini --reload
```

On start up, `pserve` logs some output:

```
Starting subprocess with file monitor
Starting server in PID 72213.
Starting HTTP server on http://0.0.0.0:6543
```

5. Open `http://localhost:6543/` in your browser.

Analysis

Rather than starting from scratch, `pcreate` can make getting a Python project containing a Pyramid application a quick matter. Pyramid ships with a few scaffolds. But installing a Pyramid add-on can give you new scaffolds from that add-on.

`pserve` is Pyramid's application runner, separating operational details from your code. When you install Pyramid, a small command program called `pserve` is written to your `bin` directory. This program is an executable Python module. It is passed a configuration file (in this case, `development.ini`).

01: Single-File Web Applications

What's the simplest way to get started in Pyramid? A single-file module. No Python packages, no `pip install -e .`, no other machinery.

Background

Microframeworks are all the rage these days. "Microframework" is a marketing term, not a technical one. They have a low mental overhead: they do so little, the only things you have to worry about are *your things*.

Pyramid is special because it can act as a single-file module microframework. You can have a single Python file that can be executed directly by Python. But Pyramid also provides facilities to scale to the largest of applications.

Python has a standard called *WSGI* that defines how Python web applications plug into standard servers, getting passed incoming requests, and returning responses. Most modern Python web frameworks obey an "MVC" (model-view-controller) application pattern, where the data in the model has a view that mediates interaction with outside systems.

In this step we'll see a brief glimpse of WSGI servers, WSGI applications, requests, responses, and views.

Objectives

- Get a running Pyramid web application, as simply as possible.
- Use that as a well-understood base for adding each unit of complexity.
- Initial exposure to WSGI apps, requests, views, and responses.

Steps

1. Make sure you have followed the steps in *Requirements*.
2. Starting from your workspace directory (`~/projects/quick_tutorial`), create a directory for this step:

```
$ mkdir hello_world; cd hello_world
```

3. Copy the following into `hello_world/app.py`:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5
6 def hello_world(request):
7     print('Incoming request')
8     return Response('<body><h1>Hello World!</h1></body>')
9
10
11 if __name__ == '__main__':
12     config = Configurator()
13     config.add_route('hello', '/')
14     config.add_view(hello_world, route_name='hello')
15     app = config.make_wsgi_app()
16     server = make_server('0.0.0.0', 6543, app)
17     server.serve_forever()
```

4. Run the application:

```
$ $VENV/bin/python app.py
```

5. Open `http://localhost:6543/` in your browser.

Analysis

New to Python web programming? If so, some lines in the module merit explanation:

1. *Line 11.* The `if __name__ == '__main__':` is Python’s way of saying, “Start here when running from the command line”, rather than when this module is imported.
2. *Lines 12-14.* Use Pyramid’s *configurator* to connect *view* code to a particular URL *route*.
3. *Lines 6-8.* Implement the *view* code that generates the *response*.
4. *Lines 15-17.* Publish a *WSGI* app using an HTTP server.

As shown in this example, the *configurator* plays a central role in Pyramid development. Building an application from loosely-coupled parts via *Application Configuration* is a central idea in Pyramid, one that we will revisit regularly in this *Quick Tutorial*.

Extra credit

1. Why do we do this:

```
print('Incoming request')
```

...instead of:

```
print 'Incoming request'
```

2. What happens if you return a string of HTML? A sequence of integers?
3. Put something invalid, such as `print xyz`, in the view function. Kill your `python app.py` with `ctrl-C` and restart, then reload your browser. See the exception in the console?
4. The `GI` in `WSGI` stands for “Gateway Interface”. What web standard is this modelled after?

02: Python Packages for Pyramid Applications

Most modern Python development is done using Python packages, an approach Pyramid puts to good use. In this step we redo “Hello World” as a minimal Python package inside a minimal Python project.

Background

Python developers can organize a collection of modules and files into a namespaced unit called a package. If a directory is on `sys.path` and has a special file named `__init__.py`, it is treated as a Python package.

Packages can be bundled up, made available for installation, and installed through a toolchain oriented around a `setup.py` file. For this tutorial, this is all you need to know:

- We will have a directory for each tutorial step as a *project*.
- This project will contain a `setup.py` which injects the features of the project machinery into the directory.
- In this project we will make a `tutorial` subdirectory into a Python *package* using an `__init__.py` Python module file.
- We will run `pip install -e .` to install our project in development mode.

In summary:

- You'll do your development in a Python *package*.
- That package will be part of a *project*.

Objectives

- Make a Python “package” directory with an `__init__.py`.
- Get a minimum Python “project” in place by making a `setup.py`.
- Install our `tutorial` project in development mode.

Steps

1. Make an area for this tutorial step:

```
$ cd ../; mkdir package; cd package
```

2. In `package/setup.py`, enter the following:

```
from setuptools import setup

requires = [
    'pyramid',
]

setup(name='tutorial',
      install_requires=requires,
)
```

3. Make the new project installed for development then make a directory for the actual code:

```
$ $VENV/bin/pip install -e .
$ mkdir tutorial
```

4. Enter the following into `package/tutorial/__init__.py`:

```
# package
```

5. Enter the following into `package/tutorial/app.py`:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.response import Response

def hello_world(request):
    print ('Incoming request')
    return Response('<body><h1>Hello World!</h1></body>')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('hello', '/')
    config.add_view(hello_world, route_name='hello')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 6543, app)
    server.serve_forever()
```

6. Run the WSGI application with:

```
$ $VENV/bin/python tutorial/app.py
```

7. Open `http://localhost:6543/` in your browser.

Analysis

Python packages give us an organized unit of project development. Python projects, via `setup.py`, give us special features when our package is installed (in this case, in local development mode, also called local editable mode as indicated by `-e .`).

In this step we have a Python package called `tutorial`. We use the same name in each step of the tutorial, to avoid unnecessary retyping.

Above this `tutorial` directory we have the files that handle the packaging of this project. At the moment, all we need is a bare-bones `setup.py`.

Everything else is the same about our application. We simply made a Python package with a `setup.py` and installed it in development mode.

Note that the way we're running the app (`python tutorial/app.py`) is a bit of an odd duck. We would never do this unless we were writing a tutorial that tries to capture how this stuff works one step at a time. It's generally a bad idea to run a Python module inside a package directly as a script.

See also:

Python Packages and Working in “Development Mode”.

03: Application Configuration with `.ini` Files

Use Pyramid's `pserve` command with a `.ini` configuration file for simpler, better application running.

Background

Pyramid has a first-class concept of *configuration* distinct from code. This approach is optional, but its presence makes it distinct from other Python web frameworks. It taps into Python's `setuptools` library, which establishes conventions for installing and providing “entry points” for Python projects. Pyramid uses an entry point to let a Pyramid application know where to find the WSGI app.

Objectives

- Modify our `setup.py` to have an entry point telling Pyramid the location of the WSGI app.
- Create an application driven by an `.ini` file.
- Start the application with Pyramid's `pserve` command.
- Move code into the package's `__init__.py`.

Steps

1. First we copy the results of the previous step:

```
$ cd ../; cp -r package ini; cd ini
```

2. Our `ini/setup.py` needs a `setuptools` “entry point” in the `setup()` function:

```
1 from setuptools import setup
2
3 requires = [
4     'pyramid',
5 ]
6
7 setup(name='tutorial',
8       install_requires=requires,
9       entry_points="""\
10     [paste.app_factory]
11     main = tutorial:main
12     """,
13 )
```

3. We can now install our project, thus generating (or re-generating) an “egg” at `ini/tutorial.egg-info`:

```
$ $VENV/bin/pip install -e .
```

4. Let's make a file `ini/development.ini` for our configuration:

```

1 [app:main]
2 use = egg:tutorial
3
4 [server:main]
5 use = egg:pyramid#wsgiref
6 host = 0.0.0.0
7 port = 6543

```

5. We can refactor our startup code from the previous step's `app.py` into `ini/tutorial/__init__.py`:

```

1 from pyramid.config import Configurator
2 from pyramid.response import Response
3
4
5 def hello_world(request):
6     return Response('<body><h1>Hello World!</h1></body>')
7
8
9 def main(global_config, **settings):
10     config = Configurator(settings=settings)
11     config.add_route('hello', '/')
12     config.add_view(hello_world, route_name='hello')
13     return config.make_wsgi_app()

```

6. Now that `ini/tutorial/app.py` isn't used, let's remove it:

```
$ rm tutorial/app.py
```

7. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

8. Open `http://localhost:6543/`.

Analysis

Our `development.ini` file is read by `pserve` and serves to bootstrap our application. Processing then proceeds as described in the Pyramid chapter on *application startup*:

- `pserve` looks for `[app:main]` and finds `use = egg:tutorial`.
- The project's `setup.py` has defined an “entry point” (lines 9-12) for the project's “main” entry point of `tutorial:main`.
- The `tutorial` package's `__init__` has a main function.
- This function is invoked, with the values from certain `.ini` sections passed in.

The `.ini` file is also used for two other functions:

- *Configuring the WSGI server.* `[server:main]` wires up the choice of which WSGI *server* for your WSGI *application*. In this case, we are using `wsgiref` bundled in the Python library. It also wires up the *port number*: `port = 6543` tells `wsgiref` to listen on port 6543.
- *Configuring Python logging.* Pyramid uses Python standard logging, which needs a number of configuration values. The `.ini` serves this function. This provides the console log output that you see on startup and each request.

We moved our startup code from `app.py` to the package's `tutorial/__init__.py`. This isn't necessary, but it is a common style in Pyramid to take the WSGI app bootstrapping out of your module's code and put it in the package's `__init__.py`.

The `pserve` application runner has a number of command-line arguments and options. We are using `--reload` which tells `pserve` to watch the filesystem for changes to relevant code (Python files, the INI file, etc.) and, when something changes, restart the application. Very handy during development.

Extra credit

1. If you don't like configuration and/or `.ini` files, could you do this yourself in Python code?
2. Can we have multiple `.ini` configuration files for a project? Why might you want to do that?
3. The entry point in `setup.py` didn't mention `__init__.py` when it declared `tutorial:main` function. Why not?
4. What is the purpose of `**settings`? What does the `**` signify?

See also:

Creating a Pyramid Project, Creating Pyramid Scaffolds, What Is This pserve Thing, Environment Variables and .ini File Settings, PasteDeploy Configuration Files

04: Easier Development with debugtoolbar

Error handling and introspection using the `pyramid_debugtoolbar` add-on.

Background

As we introduce the basics, we also want to show how to be productive in development and debugging. For example, we just discussed template reloading, and earlier we showed `--reload` for application reloading.

`pyramid_debugtoolbar` is a popular Pyramid add-on which makes several tools available in your browser. Adding it to your project illustrates several points about configuration.

Objectives

- Install and enable the toolbar to help during development.
- Explain Pyramid add-ons.
- Show how an add-on gets configured into your application.

Steps

1. First we copy the results of the previous step, as well as install the `pyramid_debugtoolbar` package:

```
$ cd ../; cp -r ini debugtoolbar; cd debugtoolbar
$ $VENV/bin/pip install -e .
$ $VENV/bin/pip install pyramid_debugtoolbar
```

2. Our `debugtoolbar/development.ini` gets a configuration entry for `pyramid`. includes:


```
1 [app:main]
2 use = egg:tutorial
3 pyramid.includes =
4     pyramid_debugtoolbar
5
6 [server:main]
7 use = egg:pyramid#wsgiref
8 host = 0.0.0.0
9 port = 6543
```

3. Run the WSGI application with:

```
$ $VENV/bin/pserve development.ini --reload
```

4. Open `http://localhost:6543/` in your browser. See the handy toolbar on the right.

Analysis

`pyramid_debugtoolbar` is a full-fledged Python package, available on PyPI just like thousands of other Python packages. Thus we start by installing the `pyramid_debugtoolbar` package into our virtual environment using normal Python package installation commands.

The `pyramid_debugtoolbar` Python package is also a Pyramid add-on, which means we need to include its add-on configuration into our web application. We could do this with imperative configuration in `tutorial/__init__.py` by using `config.include`. Pyramid also supports wiring in add-on configuration via our `development.ini` using `pyramid.includes`. We use this to load the configuration for the debugtoolbar.

You'll now see an attractive button on the right side of your browser, which you may click to provide introspective access to debugging information in a new browser tab. Even better, if your web application generates an error, you will see a nice traceback on the screen. When you want to disable this toolbar, there's no need to change code: you can remove it from `pyramid.includes` in the relevant `.ini` configuration file (thus showing why configuration files are handy).

Note that the toolbar injects a small amount of HTML/CSS into your app just before the closing `</body>` tag in order to display itself. If you start to experience otherwise inexplicable client-side weirdness, you can shut it off by commenting out the `pyramid_debugtoolbar` line in `pyramid.includes` temporarily.

See also:

See also `pyramid_debugtoolbar`.

Extra credit

1. Why don't we add `pyramid_debugtoolbar` to the list of `install_requires` dependencies in `debugtoolbar/setup.py`?
2. Introduce a bug into your application. Change:

```
def hello_world(request):  
    return Response('<body><h1>Hello World!</h1></body>')
```

to:

```
def hello_world(request):  
    return xResponse('<body><h1>Hello World!</h1></body>')
```

Save, and visit `http://localhost:6543/` again. Notice the nice traceback display. On the lowest line, click the “screen” icon to the right, and try typing the variable names `request` and `Response`. What else can you discover?

05: Unit Tests and `pytest`

Provide unit testing for our project's Python code.

Background

As the mantra says, “Untested code is broken code.” The Python community has had a long culture of writing test scripts which ensure that your code works correctly as you write it and maintain it in the future. Pyramid has always had a deep commitment to testing, with 100% test coverage from the earliest pre-releases.

Python includes a unit testing framework in its standard library. Over the years a number of Python projects, such as `pytest`, have extended this framework with alternative test runners that provide more convenience and functionality. The Pyramid developers use `pytest`, which we'll use in this tutorial.

Don't worry, this tutorial won't be pedantic about “test-driven development” (TDD). We'll do just enough to ensure that, in each step, we haven't majorly broken the code. As you're writing your code, you might find this more convenient than changing to your browser constantly and clicking reload.

We'll also leave discussion of `pytest-cov` for another section.

Objectives

- Write unit tests that ensure the quality of our code.
- Install a Python package (`pytest`) which helps in our testing.

Steps

1. First we copy the results of the previous step, as well as install the `pytest` package:

```
$ cd ../; cp -r debugtoolbar unit_testing; cd unit_testing
$ $VENV/bin/pip install -e .
$ $VENV/bin/pip install pytest
```

2. Now we write a simple unit test in `unit_testing/tutorial/tests.py`:

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_hello_world(self):
14        from tutorial import hello_world
15
16        request = testing.DummyRequest()
17        response = hello_world(request)
18        self.assertEqual(response.status_code, 200)
```

3. Now run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
.
1 passed in 0.14 seconds
```

Analysis

Our `tests.py` imports the Python standard unit testing framework. To make writing Pyramid-oriented tests more convenient, Pyramid supplies some `pyramid.testing` helpers which we use in the test setup and teardown. Our one test imports the view, makes a dummy request, and sees if the view returns what we expect.

The `tests.TutorialViewTests.test_hello_world` test is a small example of a unit test. First, we import the view inside each test. Why not import at the top, like in normal Python code? Because imports can cause effects that break a test. We'd like our tests to be in *units*, hence the name *unit* testing. Each test should isolate itself to the correct degree.

Our test then makes a fake incoming web request, then calls our Pyramid view. We test the HTTP status code on the response to make sure it matches our expectations.

Note that our use of `pyramid.testing.setUp()` and `pyramid.testing.tearDown()` aren't actually necessary here; they are only necessary when your test needs to make use of the `config` object (it's a Configurator) to add stuff to the configuration state before calling the view.

Extra credit

1. Change the test to assert that the response status code should be 404 (meaning, not found). Run `py.test` again. Read the error report and see if you can decipher what it is telling you.
2. As a more realistic example, put the `tests.py` back as you found it, and put an error in your view, such as a reference to a non-existing variable. Run the tests and see how this is more convenient than reloading your browser and going back to your code.
3. Finally, for the most realistic test, read about Pyramid `Response` objects and see how to change the response code. Run the tests and see how testing confirms the “contract” that your code claims to support.
4. How could we add a unit test assertion to test the HTML value of the response body?
5. Why do we import the `hello_world` view function *inside* the `test_hello_world` method instead of at the top of the module?

See also:

See also *Unit, Integration, and Functional Testing*

06: Functional Testing with WebTest

Write end-to-end full-stack testing using `webtest`.

Background

Unit tests are a common and popular approach to test-driven development (TDD). In web applications, though, the templating and entire apparatus of a web site are important parts of the delivered quality. We'd like a way to test these.

WebTest is a Python package that does functional testing. With WebTest you can write tests which simulate a full HTTP request against a WSGI application, then test the information in the response. For speed purposes, WebTest skips the setup/teardown of an actual HTTP server, providing tests that run fast enough to be part of TDD.

Objectives

- Write a test which checks the contents of the returned HTML.

Steps

1. First we copy the results of the previous step, as well as install the `webtest` package:

```
$ cd ../; cp -r unit_testing functional_testing; cd functional_testing
$ $VENV/bin/pip install -e .
$ $VENV/bin/pip install webtest
```

2. Let's extend `functional_testing/tutorial/tests.py` to include a functional test:

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
```

```

8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_hello_world(self):
14        from tutorial import hello_world
15
16        request = testing.DummyRequest()
17        response = hello_world(request)
18        self.assertEqual(response.status_code, 200)
19
20
21    class TutorialFunctionalTests(unittest.TestCase):
22        def setUp(self):
23            from tutorial import main
24            app = main({})
25            from webtest import TestApp
26
27            self.testapp = TestApp(app)
28
29        def test_hello_world(self):
30            res = self.testapp.get('/', status=200)
31            self.assertIn(b'<h1>Hello World!</h1>', res.body)

```

Be sure this file is not executable, or pytest may not include your tests.

3. Now run the tests:

```

$ $VENV/bin/py.test tutorial/tests.py -q
..
2 passed in 0.25 seconds

```

Analysis

We now have the end-to-end testing we were looking for. WebTest lets us simply extend our existing pytest-based test approach with functional tests that are reported in the same output. These new tests not only cover our templating, but they didn't dramatically increase the execution time of our tests.

Extra credit

1. Why do our functional tests use `b' '`?

07: Basic Web Handling With Views

Organize a views module with decorators and multiple views.

Background

For the examples so far, the `hello_world` function is a “view”. In Pyramid, views are the primary way to accept web requests and return responses.

So far our examples place everything in one file:

- The view function
- Its registration with the configurator
- The route to map it to a URL
- The WSGI application launcher

Let’s move the views out to their own `views.py` module and change our startup code to scan that module, looking for decorators that set up the views. Let’s also add a second view and update our tests.

Objectives

- Move views into a module that is scanned by the configurator.
- Create decorators that do declarative configuration.

Steps

1. Let’s begin by using the previous package as a starting point for a new distribution, then making it active:

```
$ cd ..; cp -r functional_testing views; cd views
$ $VENV/bin/pip install -e .
```

2. Our `views/tutorial/__init__.py` gets a lot shorter:

```

1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.add_route('home', '/')
7     config.add_route('hello', '/howdy')
8     config.scan('.views')
9     return config.make_wsgi_app()

```

3. Let's add a module `views/tutorial/views.py` that is focused on handling requests and responses:

```

1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4
5 # First view, available at http://localhost:6543/
6 @view_config(route_name='home')
7 def home(request):
8     return Response('<body>Visit <a href="/howdy">hello</a></body>')
9
10
11 # /howdy
12 @view_config(route_name='hello')
13 def hello(request):
14     return Response('<body>Go back <a href="/">home</a></body>')

```

4. Update the tests to cover the two new views:

```

1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_home(self):
14        from .views import home

```



```
15         request = testing.DummyRequest()
16         response = home(request)
17         self.assertEqual(response.status_code, 200)
18         self.assertIn(b'Visit', response.body)
19
20
21     def test_hello(self):
22         from .views import hello
23
24         request = testing.DummyRequest()
25         response = hello(request)
26         self.assertEqual(response.status_code, 200)
27         self.assertIn(b'Go back', response.body)
28
29
30 class TutorialFunctionalTests(unittest.TestCase):
31     def setUp(self):
32         from tutorial import main
33         app = main({})
34         from webtest import TestApp
35
36         self.testapp = TestApp(app)
37
38     def test_home(self):
39         res = self.testapp.get('/', status=200)
40         self.assertIn(b'<body>Visit', res.body)
41
42     def test_hello(self):
43         res = self.testapp.get('/howdy', status=200)
44         self.assertIn(b'<body>Go back', res.body)
```

5. Now run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
....
4 passed in 0.28 seconds
```

6. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

7. Open <http://localhost:6543/> and <http://localhost:6543/howdy> in your browser.

Analysis

We added some more URLs, but we also removed the view code from the application startup code in `tutorial/__init__.py`. Our views, and their view registrations (via decorators) are now in a module `views.py`, which is scanned via `config.scan('.views')`.

We have two views, each leading to the other. If you start at `http://localhost:6543/`, you get a response with a link to the next view. The `hello` view (available at the URL `/howdy`) has a link back to the first view.

This step also shows that the name appearing in the URL, the name of the “route” that maps a URL to a view, and the name of the view, can all be different. More on routes later.

Earlier we saw `config.add_view` as one way to configure a view. This section introduces `@view_config`. Pyramid’s configuration supports *imperative configuration*, such as the `config.add_view` in the previous example. You can also use *declarative configuration*, in which a Python decorator is placed on the line above the view. Both approaches result in the same final configuration, thus usually, it is simply a matter of taste.

Extra credit

1. What does the dot in `.views` signify?
2. Why might `assertIn` be a better choice in testing the text in responses than `assertEqual`?

See also:

Views, *View Configuration*, and *Debugging View Configuration*

08: HTML Generation With Templating

Most web frameworks don’t embed HTML in programming code. Instead, they pass data into a templating system. In this step we look at the basics of using HTML templates in Pyramid.

Background

Ouch. We have been making our own `Response` and filling the response body with HTML. You usually won't embed an HTML string directly in Python, but instead will use a templating language.

Pyramid doesn't mandate a particular database system, form library, and so on. It encourages replaceability. This applies equally to templating, which is fortunate: developers have strong views about template languages. As of Pyramid 1.5a2, Pyramid doesn't even bundle a template language!

It does, however, have strong ties to Jinja2, Mako, and Chameleon. In this step we see how to add `pyramid_chameleon` to your project, then change your views to use templating.

Objectives

- Enable the `pyramid_chameleon` Pyramid add-on.
- Generate HTML from template files.
- Connect the templates as “renderers” for view code.
- Change the view code to simply return data.

Steps

1. Let's begin by using the previous package as a starting point for a new project:

```
$ cd ../; cp -r views templating; cd templating
```

2. This step depends on `pyramid_chameleon`, so add it as a dependency in `templating/setup.py`:

```
1 from setuptools import setup
2
3 requires = [
4     'pyramid',
5     'pyramid_chameleon',
6 ]
7
```

```

8 setup(name='tutorial',
9       install_requires=requires,
10      entry_points="""\
11          [paste.app_factory]
12          main = tutorial:main
13      """,
14      )

```

3. Now we can activate the development-mode distribution:

```
$ $VENV/bin/pip install -e .
```

4. We need to connect pyramid_chameleon as a renderer by making a call in the setup of templating/tutorial/__init__.py:

```

1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.include('pyramid_chameleon')
7     config.add_route('home', '/')
8     config.add_route('hello', '/howdy')
9     config.scan('.views')
10    return config.make_wsgi_app()

```

5. Our templating/tutorial/views.py no longer has HTML in it:

```

1 from pyramid.view import view_config
2
3
4 # First view, available at http://localhost:6543/
5 @view_config(route_name='home', renderer='home.pt')
6 def home(request):
7     return {'name': 'Home View'}
8
9
10 # /howdy
11 @view_config(route_name='hello', renderer='home.pt')
12 def hello(request):
13     return {'name': 'Hello View'}

```

6. Instead we have templating/tutorial/home.pt as a template:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Quick Tutorial: ${name}</title>
</head>
<body>
<h1>Hi ${name}</h1>
</body>
</html>
```

7. For convenience, change `templating/development.ini` to reload templates automatically with `pyramid.reload_templates`:

```
[app:main]
use = egg:tutorial
pyramid.reload_templates = true
pyramid.includes =
    pyramid_debugtoolbar

[server:main]
use = egg:pyramid#wsgiref
host = 0.0.0.0
port = 6543
```

8. Our unit tests in `templating/tutorial/tests.py` can focus on data:

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_home(self):
14        from .views import home
15
16        request = testing.DummyRequest()
17        response = home(request)
18        # Our view now returns data
```

```

19         self.assertEqual('Home View', response['name'])
20
21     def test_hello(self):
22         from .views import hello
23
24         request = testing.DummyRequest()
25         response = hello(request)
26         # Our view now returns data
27         self.assertEqual('Hello View', response['name'])
28
29
30 class TutorialFunctionalTests(unittest.TestCase):
31     def setUp(self):
32         from tutorial import main
33         app = main({})
34         from webtest import TestApp
35
36         self.testapp = TestApp(app)
37
38     def test_home(self):
39         res = self.testapp.get('/', status=200)
40         self.assertIn(b'<h1>Hi Home View', res.body)
41
42     def test_hello(self):
43         res = self.testapp.get('/howdy', status=200)
44         self.assertIn(b'<h1>Hi Hello View', res.body)

```

9. Now run the tests:

```

$ $VENV/bin/py.test tutorial/tests.py -q
....
4 passed in 0.46 seconds

```

10. Run your Pyramid application with:

```

$ $VENV/bin/pserve development.ini --reload

```

11. Open <http://localhost:6543/> and <http://localhost:6543/howdy> in your browser.

Analysis

Ahh, that looks better. We have a view that is focused on Python code. Our `@view_config` decorator specifies a *renderer* that points to our template file. Our view then simply returns data which is then supplied to our template. Note that we used the same template for both views.

Note the effect on testing. We can focus on having a data-oriented contract with our view code.

See also:

Templates, Debugging Templates, and Available Add-On Template System Bindings.

09: Organizing Views With View Classes

Change our view functions to be methods on a view class, then move some declarations to the class level.

Background

So far our views have been simple, free-standing functions. Many times your views are related to one another. They may be different ways to look at or work on the same data, or be a REST API that handles multiple operations. Grouping these views together as a *view class* makes sense:

- Group views.
- Centralize some repetitive defaults.
- Share some state and helpers.

In this step we just do the absolute minimum to convert the existing views to a view class. In a later tutorial step, we'll examine view classes in depth.

Objectives

- Group related views into a view class.
- Centralize configuration with class-level `@view_defaults`.

Steps

1. First we copy the results of the previous step:

```
$ cd ../; cp -r templating view_classes; cd view_classes
$ $VENV/bin/pip install -e .
```

2. Our `view_classes/tutorial/views.py` now has a view class with our two views:

```
1 from pyramid.view import (
2     view_config,
3     view_defaults
4 )
5
6 @view_defaults(renderer='home.pt')
7 class TutorialViews:
8     def __init__(self, request):
9         self.request = request
10
11     @view_config(route_name='home')
12     def home(self):
13         return {'name': 'Home View'}
14
15     @view_config(route_name='hello')
16     def hello(self):
17         return {'name': 'Hello View'}
```

3. Our unit tests in `view_classes/tutorial/tests.py` don't run, so let's modify them to import the view class, and make an instance before getting a response:

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_home(self):
14        from .views import TutorialViews
15
16        request = testing.DummyRequest()
17        inst = TutorialViews(request)
18        response = inst.home()
```



```
19         self.assertEqual('Home View', response['name'])
20
21     def test_hello(self):
22         from .views import TutorialViews
23
24         request = testing.DummyRequest()
25         inst = TutorialViews(request)
26         response = inst.hello()
27         self.assertEqual('Hello View', response['name'])
28
29
30 class TutorialFunctionalTests(unittest.TestCase):
31     def setUp(self):
32         from tutorial import main
33         app = main({})
34         from webtest import TestApp
35
36         self.testapp = TestApp(app)
37
38     def test_home(self):
39         res = self.testapp.get('/', status=200)
40         self.assertIn(b'<h1>Hi Home View', res.body)
41
42     def test_hello(self):
43         res = self.testapp.get('/howdy', status=200)
44         self.assertIn(b'<h1>Hi Hello View', res.body)
```

4. Now run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
....
4 passed in 0.34 seconds
```

5. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

6. Open <http://localhost:6543/> and <http://localhost:6543/howdy> in your browser.

Analysis

To ease the transition to view classes, we didn't introduce any new functionality. We simply changed the view functions to methods on a view class, then updated the tests.

In our `TutorialViews` view class, you can see that our two view classes are logically grouped together as methods on a common class. Since the two views shared the same template, we could move that to a `@view_defaults` decorator at the class level.

The tests needed to change. Obviously we needed to import the view class. But you can also see the pattern in the tests of instantiating the view class with the dummy request first, then calling the view method being tested.

See also:

Defining a View Callable as a Class

10: Handling Web Requests and Responses

Web applications handle incoming requests and return outgoing responses. Pyramid makes working with requests and responses convenient and reliable.

Objectives

- Learn the background on Pyramid’s choices for requests and responses.
- Grab data out of the request.
- Change information in the response headers.

Background

Developing for the web means processing web requests. As this is a critical part of a web application, web developers need a robust, mature set of software for web requests and returning web responses.

Pyramid has always fit nicely into the existing world of Python web development (virtual environments, packaging, scaffolding, first to embrace Python 3, and so on). Pyramid turned to the well-regarded *WebOb* Python library for request and response handling. In our example above, Pyramid hands `hello_world` a `request` that is *based on WebOb*.

Steps

1. First we copy the results of the `view_classes` step:

```
$ cd ../; cp -r view_classes request_response; cd request_response
$ $VENV/bin/pip install -e .
```

2. Simplify the routes in request_response/tutorial/__init__.py:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.add_route('home', '/')
7     config.add_route('plain', '/plain')
8     config.scan('.views')
9     return config.make_wsgi_app()
```

3. We only need one view in request_response/tutorial/views.py:

```
1 from pyramid.httpexceptions import HTTPFound
2 from pyramid.response import Response
3 from pyramid.view import view_config
4
5
6 class TutorialViews:
7     def __init__(self, request):
8         self.request = request
9
10    @view_config(route_name='home')
11    def home(self):
12        return HTTPFound(location='/plain')
13
14    @view_config(route_name='plain')
15    def plain(self):
16        name = self.request.params.get('name', 'No Name Provided')
17
18        body = 'URL %s with name: %s' % (self.request.url, name)
19        return Response(
20            content_type='text/plain',
21            body=body
22        )
```

4. Update the tests in request_response/tutorial/tests.py:

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_home(self):
14        from .views import TutorialViews
15
16        request = testing.DummyRequest()
17        inst = TutorialViews(request)
18        response = inst.home()
19        self.assertEqual(response.status, '302 Found')
20
21    def test_plain_without_name(self):
22        from .views import TutorialViews
23
24        request = testing.DummyRequest()
25        inst = TutorialViews(request)
26        response = inst.plain()
27        self.assertIn(b'No Name Provided', response.body)
28
29    def test_plain_with_name(self):
30        from .views import TutorialViews
31
32        request = testing.DummyRequest()
33        request.GET['name'] = 'Jane Doe'
34        inst = TutorialViews(request)
35        response = inst.plain()
36        self.assertIn(b'Jane Doe', response.body)
37
38
39 class TutorialFunctionalTests(unittest.TestCase):
40     def setUp(self):
41         from tutorial import main
42
43         app = main({})
44         from webtest import TestApp
45
46         self.testapp = TestApp(app)
```

```
47
48     def test_plain_without_name(self):
49         res = self.testapp.get('/plain', status=200)
50         self.assertIn(b'No Name Provided', res.body)
51
52     def test_plain_with_name(self):
53         res = self.testapp.get('/plain?name=Jane%20Doe', status=200)
54         self.assertIn(b'Jane Doe', res.body)
```

5. Now run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
.....
5 passed in 0.30 seconds
```

6. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

7. Open <http://localhost:6543/> in your browser. You will be redirected to <http://localhost:6543/plain>.

8. Open <http://localhost:6543/plain?name=alice> in your browser.

Analysis

In this view class, we have two routes and two views, with the first leading to the second by an HTTP redirect. Pyramid can *generate redirects* by returning a special object from a view or raising a special exception.

In this Pyramid view, we get the URL being visited from `request.url`. Also, if you visited <http://localhost:6543/plain?name=alice>, the name is included in the body of the response:

```
URL http://localhost:6543/plain?name=alice with name: alice
```

Finally, we set the response's content type and body, then return the response.

We updated the unit and functional tests to prove that our code does the redirection, but also handles sending and not sending `/plain?name`.

Extra credit

1. Could we also raise `HTTPFound(location='/plain')` instead of returning it? If so, what's the difference?

See also:

Request and Response Objects, generate redirects

11: Dispatching URLs To Views With Routing


Routing matches incoming URL patterns to view code. Pyramid's routing has a number of useful features.

Background

Writing web applications usually means sophisticated URL design. We just saw some Pyramid machinery for requests and views. Let's look at features that help in routing.

Previously we saw the basics of routing URLs to views in Pyramid.

- Your project's "setup" code registers a route name to be used when matching part of the URL
- Elsewhere a view is configured to be called for that route name.

 Why do this twice? Other Python web frameworks let you create a route and associate it with a view in one step. As illustrated in *Routes need relative ordering*, multiple routes might match the same URL pattern. Rather than provide ways to help guess, Pyramid lets you be explicit in ordering. Pyramid also gives facilities to avoid the problem. It's relatively easy to build a system that uses implicit route ordering with Pyramid too. See The Groundhog series of screencasts if you're interested in doing so.

Objectives

- Define a route that extracts part of the URL into a Python dictionary.
- Use that dictionary data in a view.

Steps

1. First we copy the results of the `view_classes` step:

```
$ cd ../; cp -r view_classes routing; cd routing
$ $VENV/bin/pip install -e .
```

2. Our routing/tutorial/___init___ .py needs a route with a replacement pattern:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.include('pyramid_chameleon')
7     config.add_route('home', '/howdy/{first}/{last}')
8     config.scan('.views')
9     return config.make_wsgi_app()
```

3. We just need one view in routing/tutorial/views.py:

```
1 from pyramid.view import (
2     view_config,
3     view_defaults
4 )
5
6
7 @view_defaults(renderer='home.pt')
8 class TutorialViews:
9     def __init__(self, request):
10         self.request = request
11
12     @view_config(route_name='home')
13     def home(self):
14         first = self.request.matchdict['first']
15         last = self.request.matchdict['last']
16         return {
17             'name': 'Home View',
18             'first': first,
19             'last': last
20         }
```

4. We just need one view in routing/tutorial/home.pt:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
```

```

4     <title>Quick Tutorial: ${name}</title>
5 </head>
6 <body>
7 <h1>${name}</h1>
8 <p>First: ${first}, Last: ${last}</p>
9 </body>
10 </html>

```

5. Update routing/tutorial/tests.py:

```

1  import unittest
2
3  from pyramid import testing
4
5
6  class TutorialViewTests(unittest.TestCase):
7      def setUp(self):
8          self.config = testing.setUp()
9
10     def tearDown(self):
11         testing.tearDown()
12
13     def test_home(self):
14         from .views import TutorialViews
15
16         request = testing.DummyRequest()
17         request.matchdict['first'] = 'First'
18         request.matchdict['last'] = 'Last'
19         inst = TutorialViews(request)
20         response = inst.home()
21         self.assertEqual(response['first'], 'First')
22         self.assertEqual(response['last'], 'Last')
23
24
25  class TutorialFunctionalTests(unittest.TestCase):
26      def setUp(self):
27          from tutorial import main
28          app = main({})
29          from webtest import TestApp
30
31          self.testapp = TestApp(app)
32
33      def test_home(self):
34          res = self.testapp.get('/howdy/Jane/Doe', status=200)
35          self.assertIn(b'Jane', res.body)

```



```
36 self.assertIn(b'Doe', res.body)
```

6. Now run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
..
2 passed in 0.39 seconds
```

7. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

8. Open <http://localhost:6543/howdy/amy/smith> in your browser.

Analysis

In `__init__.py` we see an important change in our route declaration:

```
config.add_route('hello', '/howdy/{first}/{last}')
```

With this we tell the *configurator* that our URL has a “replacement pattern”. With this, URLs such as `/howdy/amy/smith` will assign `amy` to `first` and `smith` to `last`. We can then use this data in our view:

```
self.request.matchdict['first']
self.request.matchdict['last']
```

`request.matchdict` contains values from the URL that match the “replacement patterns” (the curly braces) in the route declaration. This information can then be used anywhere in Pyramid that has access to the request.

Extra credit

1. What happens if you go to the URL <http://localhost:6543/howdy/>? Is this the result that you expected?

See also:

Weird Stuff You Can Do With URL Dispatch

12: Templating With jinja2

We just said Pyramid doesn't prefer one templating language over another. Time to prove it. Jinja2 is a popular templating system, used in Flask and modeled after Django's templates. Let's add `pyramid_jinja2`, a Pyramid *add-on* which enables Jinja2 as a *renderer* in our Pyramid applications.

Objectives

- Show Pyramid's support for different templating systems.
- Learn about installing Pyramid add-ons.

Steps

1. In this step let's start by copying the `view_class` step's directory, and then installing the `pyramid_jinja2` add-on.

```
$ cd ../; cp -r view_classes jinja2; cd jinja2
$ $VENV/bin/pip install -e .
$ $VENV/bin/pip install pyramid_jinja2
```

2. We need to include `pyramid_jinja2` in `jinja2/tutorial/__init__.py`:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.include('pyramid_jinja2')
7     config.add_route('home', '/')
8     config.add_route('hello', '/howdy')
9     config.scan('.views')
10    return config.make_wsgi_app()
```

3. Our `jinja2/tutorial/views.py` simply changes its `renderer`:

```
1 from pyramid.view import (
2     view_config,
3     view_defaults
4 )
5
6
7 @view_defaults(renderer='home.jinja2')
8 class TutorialViews:
9     def __init__(self, request):
10         self.request = request
11
12     @view_config(route_name='home')
13     def home(self):
14         return {'name': 'Home View'}
15
16     @view_config(route_name='hello')
17     def hello(self):
18         return {'name': 'Hello View'}
```

4. Add `jinja2/tutorial/home.jinja2` as a template:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Quick Tutorial: {{ name }}</title>
</head>
<body>
<h1>Hi {{ name }}</h1>
</body>
</html>
```

5. Now run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
....
4 passed in 0.40 seconds
```

6. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

7. Open `http://localhost:6543/` in your browser.

Analysis

Getting a Pyramid add-on into Pyramid is simple. First you use normal Python package installation tools to install the add-on package into your Python virtual environment. You then tell Pyramid’s configurator to run the setup code in the add-on. In this case the setup code told Pyramid to make a new “renderer” available that looked for `.jinja2` file extensions.

Our view code stayed largely the same. We simply changed the file extension on the renderer. For the template, the syntax for Chameleon and Jinja2’s basic variable insertion is very similar.

Extra credit

1. Our project now depends on `pyramid_jinja2`. We installed that dependency manually. What is another way we could have made the association?
2. We used `config.include` which is an imperative configuration to get the *Configurator* to load `pyramid_jinja2`’s configuration. What is another way could include it into the config?

See also:

Jinja2 homepage, and `pyramid_jinja2` Overview

13: CSS/JS/Images Files With Static Assets

Of course the Web is more than just markup. You need static assets: CSS, JS, and images. Let’s point our web app at a directory where Pyramid will serve some static assets.

Objectives

- Publish a directory of static assets at a URL.
- Use Pyramid to help generate URLs to files in that directory.

Steps

1. First we copy the results of the `view_classes` step:

```
$ cd ../; cp -r view_classes static_assets; cd static_assets
$ $VENV/bin/pip install -e .
```

2. We add a call `config.add_static_view` in `static_assets/tutorial/__init__.py`:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.include('pyramid_chameleon')
7     config.add_route('home', '/')
8     config.add_route('hello', '/howdy')
9     config.add_static_view(name='static', path='tutorial:static')
10    config.scan('.views')
11    return config.make_wsgi_app()
```

3. We can add a CSS link in the `<head>` of our template at `static_assets/tutorial/home.pt`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Quick Tutorial: ${name}</title>
    <link rel="stylesheet"
        href="${request.static_url('tutorial:static/app.css')}"/>
</head>
<body>
<h1>Hi ${name}</h1>
</body>
</html>
```

4. Add a CSS file at `static_assets/tutorial/static/app.css`:

```
body {
    margin: 2em;
    font-family: sans-serif;
}
```

5. Make sure we haven't broken any existing code by running the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
....
4 passed in 0.50 seconds
```

6. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

7. Open `http://localhost:6543/` in your browser and note the new font.

Analysis

We changed our WSGI application to map requests under `http://localhost:6543/static/` to files and directories inside a `static` directory inside our `tutorial` package. This directory contained `app.css`.

We linked to the CSS in our template. We could have hard-coded this link to `/static/app.css`. But what if the site is later moved under `/somesite/static/`? Or perhaps the web developer changes the arrangement on disk? Pyramid gives a helper that provides flexibility on URL generation:

```
{request.static_url('tutorial:static/app.css')}
```

This matches the `path='tutorial:static'` in our `config.add_static_view` registration. By using `request.static_url` to generate the full URL to the static assets, you both ensure you stay in sync with the configuration and gain refactoring flexibility later.

Extra credit

1. There is also a `request.static_path` API. How does this differ from `request.static_url`?

See also:

Static Assets, Preventing HTTP Caching, and Influencing HTTP Caching

14: AJAX Development With JSON Renderers

Modern web apps are more than rendered HTML. Dynamic pages now use JavaScript to update the UI in the browser by requesting server data as JSON. Pyramid supports this with a *JSON renderer*.

Background

As we saw in *08: HTML Generation With Templating*, view declarations can specify a renderer. Output from the view is then run through the renderer, which generates and returns the response. We first used a Chameleon renderer, then a Jinja2 renderer.

Renderers aren't limited, however, to templates that generate HTML. Pyramid supplies a JSON renderer which takes Python data, serializes it to JSON, and performs some other functions such as setting the content type. In fact you can write your own renderer (or extend a built-in renderer) containing custom logic for your unique application.

Steps

1. First we copy the results of the `view_classes` step:

```
$ cd ../; cp -r view_classes json; cd json
$ $VENV/bin/pip install -e .
```

2. We add a new route for `hello_json` in `json/tutorial/__init__.py`:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.include('pyramid_chameleon')
7     config.add_route('home', '/')
8     config.add_route('hello', '/howdy')
9     config.add_route('hello_json', 'howdy.json')
10    config.scan('.views')
11    return config.make_wsgi_app()
```

3. Rather than implement a new view, we will “stack” another decorator on the `hello` view in `views.py`:

```
1 from pyramid.view import (
2     view_config,
3     view_defaults
4 )
5
6
7 @view_defaults(renderer='home.pt')
8 class TutorialViews:
9     def __init__(self, request):
10         self.request = request
11
12     @view_config(route_name='home')
13     def home(self):
14         return {'name': 'Home View'}
15
16     @view_config(route_name='hello')
17     @view_config(route_name='hello_json', renderer='json')
18     def hello(self):
19         return {'name': 'Hello View'}
```

4. We need a new functional test at the end of `json/tutorial/tests.py`:

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_home(self):
14        from .views import TutorialViews
15
16        request = testing.DummyRequest()
17        inst = TutorialViews(request)
18        response = inst.home()
19        self.assertEqual('Home View', response['name'])
20
21    def test_hello(self):
22        from .views import TutorialViews
23
```



```
24         request = testing.DummyRequest()
25         inst = TutorialViews(request)
26         response = inst.hello()
27         self.assertEqual('Hello View', response['name'])
28
29
30 class TutorialFunctionalTests(unittest.TestCase):
31     def setUp(self):
32         from tutorial import main
33         app = main({})
34         from webtest import TestApp
35
36         self.testapp = TestApp(app)
37
38     def test_home(self):
39         res = self.testapp.get('/', status=200)
40         self.assertIn(b'<h1>Hi Home View', res.body)
41
42     def test_hello(self):
43         res = self.testapp.get('/howdy', status=200)
44         self.assertIn(b'<h1>Hi Hello View', res.body)
45
46     def test_hello_json(self):
47         res = self.testapp.get('/howdy.json', status=200)
48         self.assertIn(b'{"name": "Hello View"}', res.body)
49         self.assertEqual(res.content_type, 'application/json')
50
```

5. Run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
.....
5 passed in 0.47 seconds
```

6. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

7. Open `http://localhost:6543/howdy.json` in your browser and you will see the resulting JSON response.

Analysis

Earlier we changed our view functions and methods to return Python data. This change to a data-oriented view layer made test writing easier, decoupling the templating from the view logic.

Since Pyramid has a JSON renderer as well as the templating renderers, it is an easy step to return JSON. In this case we kept the exact same view and arranged to return a JSON encoding of the view data. We did this by:

- Adding a route to map `/howdy.json` to a route name.
- Providing a `@view_config` that associated that route name with an existing view.
- *Overriding* the view defaults in the view config that mentions the `hello_json` route, so that when the route is matched, we use the JSON renderer rather than the `home.pt` template renderer that would otherwise be used.

In fact, for pure AJAX-style web applications, we could re-use the existing route by using Pyramid's view predicates to match on the `Accepts:` header sent by modern AJAX implementations.

Pyramid's JSON renderer uses the base Python JSON encoder, thus inheriting its strengths and weaknesses. For example, Python can't natively JSON encode `DateTime` objects. There are a number of solutions for this in Pyramid, including extending the JSON renderer with a custom renderer.

See also:

Writing View Callables Which Use a Renderer, JSON Renderer, and Adding and Changing Renderers

15: More With View Classes

Group views into a class, sharing configuration, state, and logic.

Background

As part of its mission to help build more ambitious web applications, Pyramid provides many more features for views and view classes.

The Pyramid documentation discusses views as a Python “callable”. This callable can be a function, an object with a `__call__`, or a Python class. In this last case, methods on the class can be decorated with `@view_config` to register the class methods with the *configurator* as a view.

At first, our views were simple, free-standing functions. Many times your views are related: different ways to look at or work on the same data, or a REST API that handles multiple operations. Grouping these together as a *view class* makes sense:

- Group views.
- Centralize some repetitive defaults.
- Share some state and helpers.

Pyramid views have *view predicates* that determine which view is matched to a request, based on factors such as the request method, the form parameters, and so on. These predicates provide many axes of flexibility.

The following shows a simple example with four operations: view a home page which leads to a form, save a change, and press the delete button.

Objectives

- Group related views into a view class.
- Centralize configuration with class-level `@view_defaults`.
- Dispatch one route/URL to multiple views based on request data.
- Share states and logic between views and templates via the view class.

Steps

1. First we copy the results of the previous step:

```
$ cd ../; cp -r templating more_view_classes; cd more_view_classes
$ $VENV/bin/pip install -e .
```

2. Our route in `more_view_classes/tutorial/__init__.py` needs some replacement patterns:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.include('pyramid_chameleon')
7     config.add_route('home', '/')
8     config.add_route('hello', '/howdy/{first}/{last}')
9     config.scan('.views')
10    return config.make_wsgi_app()
```

3. Our `more_view_classes/tutorial/views.py` now has a view class with several views:

```
1 from pyramid.view import (
2     view_config,
3     view_defaults
4 )
5
6
7 @view_defaults(route_name='hello')
8 class TutorialViews(object):
9     def __init__(self, request):
10         self.request = request
11         self.view_name = 'TutorialViews'
12
13     @property
14     def full_name(self):
15         first = self.request.matchdict['first']
16         last = self.request.matchdict['last']
17         return first + ' ' + last
18
19 @view_config(route_name='home', renderer='home.pt')
20 def home(self):
21     return {'page_title': 'Home View'}
22
23 # Retrieving /howdy/first/last the first time
24 @view_config(renderer='hello.pt')
25 def hello(self):
```

```
26         return {'page_title': 'Hello View'}
27
28     # Posting to /howdy/first/last via the "Edit" submit button
29     @view_config(request_method='POST', renderer='edit.pt')
30     def edit(self):
31         new_name = self.request.params['new_name']
32         return {'page_title': 'Edit View', 'new_name': new_name}
33
34     # Posting to /howdy/first/last via the "Delete" submit button
35     @view_config(request_method='POST', request_param='form.delete',
36                 renderer='delete.pt')
37     def delete(self):
38         print ('Deleted')
39         return {'page_title': 'Delete View'}
```

4. Our primary view needs a template at `more_view_classes/tutorial/home.pt`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Quick Tutorial: ${view.view_name} - ${page_title}</title>
</head>
<body>
<h1>${view.view_name} - ${page_title}</h1>

<p>Go to the <a href="${request.route_url('hello', first='jane',
    last='doe')}">form</a>.</p>
</body>
</html>
```

5. Ditto for our other view from the previous section at `more_view_classes/tutorial/hello.pt`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Quick Tutorial: ${view.view_name} - ${page_title}</title>
</head>
<body>
<h1>${view.view_name} - ${page_title}</h1>
<p>Welcome, ${view.full_name}</p>
<form method="POST"
    action="${request.current_route_url()}">
    <input name="new_name"/>
```

```

    <input type="submit" name="form.edit" value="Save"/>
    <input type="submit" name="form.delete" value="Delete"/>
</form>
</body>
</html>

```

6. We have an edit view that also needs a template at `more_view_classes/tutorial/edit.pt`:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Quick Tutorial: ${view.view_name} - ${page_title}</title>
</head>
<body>
<h1>${view.view_name} - ${page_title}</h1>
<p>You submitted <code>${new_name}</code></p>
</body>
</html>

```

7. And finally the delete view's template at `more_view_classes/tutorial/delete.pt`:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Quick Tutorial: ${page_title}</title>
</head>
<body>
<h1>${view.view_name} - ${page_title}</h1>
</body>
</html>

```

8. Our tests in `more_view_classes/tutorial/tests.py` fail, so let's modify them:

```

1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9

```

```
10     def tearDown(self):
11         testing.tearDown()
12
13     def test_home(self):
14         from .views import TutorialViews
15
16         request = testing.DummyRequest()
17         inst = TutorialViews(request)
18         response = inst.home()
19         self.assertEqual('Home View', response['page_title'])
20
21 class TutorialFunctionalTests(unittest.TestCase):
22     def setUp(self):
23         from tutorial import main
24         app = main({})
25         from webtest import TestApp
26
27         self.testapp = TestApp(app)
28
29     def test_home(self):
30         res = self.testapp.get('/', status=200)
31         self.assertIn(b'TutorialViews - Home View', res.body)
```

9. Now run the tests:

```
$ $VENV/bin/py.test tutorial/tests.py -q
..
2 passed in 0.40 seconds
```

10. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

11. Open <http://localhost:6543/howdy/jane/doe> in your browser. Click the Save and Delete buttons, and watch the output in the console window.

Analysis

As you can see, the four views are logically grouped together. Specifically:

- We have a home view available at <http://localhost:6543/> with a clickable link to the hello view.

- The second view is returned when you go to `/howdy/jane/does`. This URL is mapped to the `hello` route that we centrally set using the optional `@view_defaults`.
- The third view is returned when the form is submitted with a `POST` method. This rule is specified in the `@view_config` for that view.
- The fourth view is returned when clicking on a button such as `<input type="submit" name="form.delete" value="Delete"/>`.

In this step we show, using the following information as criteria, how to decide which view to use:

- Method of the HTTP request (`GET`, `POST`, etc.)
- Parameter information in the request (submitted form field names)

We also centralize part of the view configuration to the class level with `@view_defaults`, then in one view, override that default just for that one view. Finally, we put this commonality between views to work in the view class by sharing:

- State assigned in `TutorialViews.__init__`
- A computed value

These are then available both in the view methods and in the templates (e.g., `${view.view_name}` and `${view.full_name}`).

As a note, we made a switch in our templates on how we generate URLs. We previously hardcoded the URLs, such as:

```
<a href="/howdy/jane/does">Howdy</a>
```

In `home.pt` we switched to:

```
<a href="${request.route_url('hello', first='jane',
                             last='does')}">form</a>
```

Pyramid has rich facilities to help generate URLs in a flexible, non-error prone fashion.

Extra credit

1. Why could our template do `${view.full_name}` and not have to do `${view.full_name() }`?
2. The `edit` and `delete` views are both receive `POST` requests. Why does the `edit` view configuration not catch the `POST` used by `delete`?
3. We used Python `@property` on `full_name`. If we reference this many times in a template or view code, it would re-compute this every time. Does Pyramid provide something that will cache the initial computation on a property?
4. Can you associate more than one route with the same view?
5. There is also a `request.route_path` API. How does this differ from `request.route_url`?

See also:

Defining a View Callable as a Class, Weird Stuff You Can Do With URL Dispatch

16: Collecting Application Info With Logging

Capture debugging and error output from your web applications using standard Python logging.

Background

It's important to know what is going on inside our web application. In development we might need to collect some output. In production, we might need to detect problems when other people use the site. We need *logging*.

Fortunately Pyramid uses the normal Python approach to logging. The scaffold generated in your `development.ini` has a number of lines that configure the logging for you to some reasonable defaults. You then see messages sent by Pyramid, for example, when a new request comes in.

Objectives

- Inspect the configuration setup used for logging.
- Add logging statements to your view code.

Steps

1. First we copy the results of the `view_classes` step:

```
$ cd ../; cp -r view_classes logging; cd logging
$ $VENV/bin/pip install -e .
```

2. Extend `logging/tutorial/views.py` to log a message:

```
1 import logging
2 log = logging.getLogger(__name__)
3
4 from pyramid.view import (
5     view_config,
6     view_defaults
7 )
8
9
10 @view_defaults(renderer='home.pt')
11 class TutorialViews:
12     def __init__(self, request):
13         self.request = request
14
15     @view_config(route_name='home')
16     def home(self):
17         log.debug('In home view')
18         return {'name': 'Home View'}
19
20     @view_config(route_name='hello')
21     def hello(self):
22         log.debug('In hello view')
23         return {'name': 'Hello View'}
```

3. Finally let's edit `development.ini` configuration file to enable logging for our Pyramid application:

```
[app:main]
use = egg:tutorial
pyramid.reload_templates = true
pyramid.includes =
    pyramid_debugtoolbar

[server:main]
```

```
use = egg:pyramid#wsgiref
host = 0.0.0.0
port = 6543

# Begin logging configuration

[loggers]
keys = root, tutorial

[logger_tutorial]
level = DEBUG
handlers =
qualname = tutorial

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(asctime)s %(levelname)-5.5s [%(name)s][%(threadName)s]
→ %(message)s

# End logging configuration
```

4. Make sure the tests still pass:

```
$ $VENV/bin/py.test tutorial/tests.py -q
....
4 passed in 0.41 seconds
```

5. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

6. Open `http://localhost:6543/` and `http://localhost:6543/howdy` in your browser. Note, both in the console and in the debug toolbar, the message that you logged.

Analysis

In our configuration file `development.ini`, our tutorial Python package is set up as a logger and configured to log messages at a `DEBUG` or higher level. When you visit `http://localhost:6543`, your console will now show:

```
2013-08-09 10:42:42,968 DEBUG [tutorial.views][MainThread] In home view
```

Also, if you have configured your Pyramid application to use the `pyramid_debugtoolbar`, logging statements appear in one of its menus.

See also:

See also *Logging*.

17: Transient Data Using Sessions

Store and retrieve non-permanent data in Pyramid sessions.

Background

When people use your web application, they frequently perform a task that requires semi-permanent data to be saved. For example, a shopping cart. This is called a *session*.

Pyramid has basic built-in support for sessions. Third party packages such as `pyramid_redis_sessions` provide richer session support. Or you can create your own custom sessioning engine. Let's take a look at the *built-in sessioning support*.

Objectives

- Make a session factory using a built-in, simple Pyramid sessioning system.
- Change our code to use a session.

Steps

1. First we copy the results of the `view_classes` step:

```
$ cd ../; cp -r view_classes sessions; cd sessions
$ $VENV/bin/pip install -e .
```

2. Our `sessions/tutorial/__init__.py` needs a choice of session factory to get registered with the *configurator*:

```
1 from pyramid.config import Configurator
2 from pyramid.session import SignedCookieSessionFactory
3
4
5 def main(global_config, **settings):
6     my_session_factory = SignedCookieSessionFactory(
7         'itsaseekreet')
8     config = Configurator(settings=settings,
9                           session_factory=my_session_factory)
10    config.include('pyramid_chameleon')
11    config.add_route('home', '/')
12    config.add_route('hello', '/howdy')
13    config.scan('.views')
14    return config.make_wsgi_app()
```

3. Our views in `sessions/tutorial/views.py` can now use `request.session`:

```
1 from pyramid.view import (
2     view_config,
3     view_defaults
4 )
5
6
7 @view_defaults(renderer='home.pt')
8 class TutorialViews:
```

```

9     def __init__(self, request):
10         self.request = request
11
12     @property
13     def counter(self):
14         session = self.request.session
15         if 'counter' in session:
16             session['counter'] += 1
17         else:
18             session['counter'] = 1
19
20         return session['counter']
21
22
23     @view_config(route_name='home')
24     def home(self):
25         return {'name': 'Home View'}
26
27     @view_config(route_name='hello')
28     def hello(self):
29         return {'name': 'Hello View'}

```

4. The template at `sessions/tutorial/home.pt` can display the value:

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <title>Quick Tutorial: ${name}</title>
5  </head>
6  <body>
7      <h1>Hi ${name}</h1>
8      <p>Count: ${view.counter}</p>
9  </body>
10 </html>

```

5. Make sure the tests still pass:

```

$ $VENV/bin/py.test tutorial/tests.py -q
....
4 passed in 0.42 seconds

```

6. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

7. Open `http://localhost:6543/` and `http://localhost:6543/howdy` in your browser. As you reload and switch between those URLs, note that the counter increases and is *not* specific to the URL.
8. Restart the application and revisit the page. Note that counter still increases from where it left off.

Analysis

Pyramid's *request* object now has a `session` attribute that we can use in our view code. It acts like a dictionary.

Since all the views are using the same counter, we made the counter a Python property at the view class level. With this, each reload will increase the counter displayed in our template.

In web development, “flash messages” are notes for the user that need to appear on a screen after a future web request. For example, when you add an item using a form `POST`, the site usually issues a second HTTP Redirect web request to view the new item. You might want a message to appear after that second web request saying “Your item was added.” You can't just return it in the web response for the `POST`, as it will be tossed out during the second web request.

Flash messages are a technique where messages can be stored between requests, using sessions, then removed when they finally get displayed.

See also:

Sessions, *Flash Messages*, and *pyramid.session*.

18: Forms and Validation with Deform

Schema-driven, autogenerated forms with validation.

Background

Modern web applications deal extensively with forms. Developers, though, have a wide range of philosophies about how frameworks should help them with their forms. As such, Pyramid doesn't directly bundle one particular form library. Instead there are a variety of form libraries that are easy to use in Pyramid.

Deform is one such library. In this step, we introduce Deform for our forms. This also gives us Colander for schemas and validation.

Objectives

- Make a schema using Colander, the companion to Deform.
- Create a form with Deform and change our views to handle validation.

Steps

1. First we copy the results of the `view_classes` step:

```
$ cd ../; cp -r view_classes forms; cd forms
```

2. Let's edit `forms/setup.py` to declare a dependency on Deform (which then pulls in Colander as a dependency:

```
1 from setuptools import setup
2
3 requires = [
4     'pyramid',
5     'pyramid_chameleon',
6     'deform'
7 ]
8
9 setup(name='tutorial',
10       install_requires=requires,
11       entry_points="""\
12     [paste.app_factory]
13     main = tutorial:main
14     """,
15 )
```

3. We can now install our project in development mode:

```
$ $VENV/bin/pip install -e .
```

4. Register a static view in `forms/tutorial/__init__.py` for Deform's CSS, JavaScript, etc., as well as our demo wiki page's views:


```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     config = Configurator(settings=settings)
6     config.include('pyramid_chameleon')
7     config.add_route('wiki_view', '/')
8     config.add_route('wikipage_add', '/add')
9     config.add_route('wikipage_view', '/{uid}')
10    config.add_route('wikipage_edit', '/{uid}/edit')
11    config.add_static_view('deform_static', 'deform:static/')
12    config.scan('.views')
13    return config.make_wsgi_app()
```

5. Implement the new views, as well as the form schemas and some dummy data, in `forms/tutorial/views.py`:

```
1 import colander
2 import deform.widget
3
4 from pyramid.httpexceptions import HTTPFound
5 from pyramid.view import view_config
6
7 pages = {
8     '100': dict(uid='100', title='Page 100', body='<em>100</em>'),
9     '101': dict(uid='101', title='Page 101', body='<em>101</em>'),
10    '102': dict(uid='102', title='Page 102', body='<em>102</em>')
11 }
12
13 class WikiPage(colander.MappingSchema):
14     title = colander.SchemaNode(colander.String())
15     body = colander.SchemaNode(
16         colander.String(),
17         widget=deform.widget.RichTextWidget()
18     )
19
20
21 class WikiViews(object):
22     def __init__(self, request):
23         self.request = request
24
25     @property
26     def wiki_form(self):
27         schema = WikiPage()
28         return deform.Form(schema, buttons=('submit',))
```

```

29
30 @property
31 def reqts(self):
32     return self.wiki_form.get_widget_resources()
33
34 @view_config(route_name='wiki_view', renderer='wiki_view.pt')
35 def wiki_view(self):
36     return dict(pages=pages.values())
37
38 @view_config(route_name='wikipage_add',
39              renderer='wikipage_addedit.pt')
40 def wikipage_add(self):
41     form = self.wiki_form.render()
42
43     if 'submit' in self.request.params:
44         controls = self.request.POST.items()
45         try:
46             appstruct = self.wiki_form.validate(controls)
47         except deform.ValidationFailure as e:
48             # Form is NOT valid
49             return dict(form=e.render())
50
51             # Form is valid, make a new identifier and add to list
52             last_uid = int(sorted(pages.keys())[-1])
53             new_uid = str(last_uid + 1)
54             pages[new_uid] = dict(
55                 uid=new_uid, title=appstruct['title'],
56                 body=appstruct['body']
57             )
58
59             # Now visit new page
60             url = self.request.route_url('wikipage_view', uid=new_uid)
61             return HTTPFound(url)
62
63     return dict(form=form)
64
65 @view_config(route_name='wikipage_view', renderer='wikipage_view.pt
66 →')
67 def wikipage_view(self):
68     uid = self.request.matchdict['uid']
69     page = pages[uid]
70     return dict(page=page)
71
72 @view_config(route_name='wikipage_edit',
73              renderer='wikipage_addedit.pt')
74 def wikipage_edit(self):

```

```
74         uid = self.request.matchdict['uid']
75         page = pages[uid]
76
77         wiki_form = self.wiki_form
78
79         if 'submit' in self.request.params:
80             controls = self.request.POST.items()
81             try:
82                 appstruct = wiki_form.validate(controls)
83             except deform.ValidationFailure as e:
84                 return dict(page=page, form=e.render())
85
86             # Change the content and redirect to the view
87             page['title'] = appstruct['title']
88             page['body'] = appstruct['body']
89
90             url = self.request.route_url('wikipage_view',
91                                         uid=page['uid'])
92             return HTTPFound(url)
93
94         form = wiki_form.render(page)
95
96         return dict(page=page, form=form)
```

6. A template for the top of the “wiki” in forms/tutorial/wiki_view.pt:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <title>Wiki: View</title>
5  </head>
6  <body>
7      <h1>Wiki</h1>
8
9      <a href="${request.route_url('wikipage_add')}">Add
10         WikiPage</a>
11      <ul>
12          <li tal:repeat="page pages">
13              <a href="${request.route_url('wikipage_view', uid=page.uid)}">
14                  ${page.title}
15              </a>
16          </li>
17      </ul>
18  </body>
19  </html>
```

7. Another template for adding/editing in forms/tutorial/wikipage_addedit.pt:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <title>WikiPage: Add/Edit</title>
5   <link rel="stylesheet"
6     href="{request.static_url('deform:static/css/bootstrap.min.
7     ↪css')}"
8     type="text/css" media="screen" charset="utf-8"/>
9   <link rel="stylesheet"
10     href="{request.static_url('deform:static/css/form.css')}"
11     type="text/css"/>
12   <tal:block tal:repeat="reqt view.reqs['css']">
13     <link rel="stylesheet" type="text/css"
14       href="{request.static_url(reqt) }"/>
15   </tal:block>
16   <script src="{request.static_url('deform:static/scripts/jquery-2.
17     ↪0.3.min.js')}"
18     type="text/javascript"></script>
19   <script src="{request.static_url('deform:static/scripts/bootstrap.
20     ↪min.js')}"
21     type="text/javascript"></script>
22   <tal:block tal:repeat="reqt view.reqs['js']">
23     <script src="{request.static_url(reqt) }"
24       type="text/javascript"></script>
25   </tal:block>
26 </head>
27 <body>
28 <h1>Wiki</h1>
29 <p>${structure: form}</p>
30 <script type="text/javascript">
31   deform.load()
32 </script>
33 </body>
34 </html>

```

8. Add a template at forms/tutorial/wikipage_view.pt for viewing a wiki page:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <title>WikiPage: View</title>
5 </head>

```

```
6 <body>
7 <a href="{request.route_url('wiki_view')}}">
8     Up
9 </a> |
10 <a href="{request.route_url('wikipage_edit', uid=page.uid)}">
11     Edit
12 </a>
13
14 <h1>${page.title}</h1>
15 <p>${structure: page.body}</p>
16 </body>
17 </html>
```

9. Our tests in `forms/tutorial/tests.py` don't run, so let's modify them:

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class TutorialViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_home(self):
14        from .views import WikiViews
15
16        request = testing.DummyRequest()
17        inst = WikiViews(request)
18        response = inst.wiki_view()
19        self.assertEqual(len(response['pages']), 3)
20
21
22 class TutorialFunctionalTests(unittest.TestCase):
23     def setUp(self):
24         from tutorial import main
25
26         app = main({})
27         from webtest import TestApp
28
29         self.testapp = TestApp(app)
30
31     def tearDown(self):
```

```

32         testing.tearDown()
33
34     def test_home(self):
35         res = self.testapp.get('/', status=200)
36         self.assertIn(b'<title>Wiki: View</title>', res.body)

```

10. Run the tests:

```

$ $VENV/bin/py.test tutorial/tests.py -q
..
2 passed in 0.45 seconds

```

11. Run your Pyramid application with:

```

$ $VENV/bin/pserve development.ini --reload

```

12. Open <http://localhost:6543/> in a browser.

Analysis

This step helps illustrate the utility of asset specifications for static assets. We have an outside package called Deform with static assets which need to be published. We don't have to know where on disk it is located. We point at the package, then the path inside the package.

We just need to include a call to `add_static_view` to make that directory available at a URL. For Pyramid-specific packages, Pyramid provides a facility (`config.include()`) which even makes that unnecessary for consumers of a package. (Deform is not specific to Pyramid.)

Our forms have rich widgets which need the static CSS and JavaScript just mentioned. Deform has a resource registry which allows widgets to specify which JavaScript and CSS are needed. Our `wikipage_addedit.pt` template shows how we iterated over that data to generate markup that includes the needed resources.

Our add and edit views use a pattern called *self-posting forms*. Meaning, the same URL is used to GET the form as is used to POST the form. The route, the view, and the template are the same URL whether you are walking up to it for the first time or you clicked a button.

Inside the view we do `if 'submit' in self.request.params:` to see if this form was a POST where the user clicked on a particular button `<input name="submit">`.

The form controller then follows a typical pattern:

- If you are doing a `GET`, skip over and just return the form.
- If you are doing a `POST`, validate the form contents.
- If the form is invalid, bail out by re-rendering the form with the supplied `POST` data.
- If the validation succeeded, perform some action and issue a redirect via `HTTPFound`.

We are, in essence, writing our own form controller. Other Pyramid-based systems, including `pyramid_deform`, provide a form-centric view class which automates much of this branching and routing.

Extra credit

1. Give a try at a button that goes to a delete view for a particular wiki page.

19: Databases Using SQLAlchemy

Store and retrieve data using the SQLAlchemy ORM atop the SQLite database.

Background

Our Pyramid-based wiki application now needs database-backed storage of pages. This frequently means an SQL database. The Pyramid community strongly supports the SQLAlchemy project and its object-relational mapper (ORM) as a convenient, Pythonic way to interface to databases.

In this step we hook up SQLAlchemy to a SQLite database table, providing storage and retrieval for the wiki pages in the previous step.



The `alchemy` scaffold is really helpful for getting an SQLAlchemy project going, including generation of the console script. Since we want to see all the decisions, we will forgo convenience in this tutorial, and wire it up ourselves.

Objectives

- Store pages in SQLite by using SQLAlchemy models.
- Use SQLAlchemy queries to list/add/view/edit pages.
- Provide a database-initialize command by writing a Pyramid *console script* which can be run from the command line.

Steps

1. We are going to use the forms step as our starting point:

```
$ cd ../; cp -r forms databases; cd databases
```

2. We need to add some dependencies in `databases/setup.py` as well as an “entry point” for the command-line script:

```
1 from setuptools import setup
2
3 requires = [
4     'pyramid',
5     'pyramid_chameleon',
6     'deform',
7     'sqlalchemy',
8     'pyramid_tm',
9     'zope.sqlalchemy'
10 ]
11
12 setup(name='tutorial',
13       install_requires=requires,
14       entry_points="""\
15     [paste.app_factory]
16     main = tutorial:main
17     [console_scripts]
18     initialize_tutorial_db = tutorial.initialize_db:main
19     """,
20 )
```



We aren't yet doing `$VENV/bin/pip install -e .` as we will change it later.

3. Our configuration file at `databases/development.ini` wires together some new pieces:

```
[app:main]
use = egg:tutorial
pyramid.reload_templates = true
pyramid.includes =
    pyramid_debugtoolbar
    pyramid_tm

sqlalchemy.url = sqlite:///%(here)s/sqltutorial.sqlite

[server:main]
use = egg:pyramid#wsgiref
host = 0.0.0.0
port = 6543

# Begin logging configuration

[loggers]
keys = root, tutorial, sqlalchemy.engine.base.Engine

[logger_tutorial]
level = DEBUG
handlers =
qualname = tutorial

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

[logger_sqlalchemy.engine.base.Engine]
level = INFO
handlers =
qualname = sqlalchemy.engine.base.Engine

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic
```

```
[formatter_generic]
format = %(asctime)s %(levelname)-5.5s [%(name)s][%(threadName)s]
↳ %(message)s

# End logging configuration
```

4. This engine configuration now needs to be read into the application through changes in `databases/tutorial/__init__.py`:

```
1 from pyramid.config import Configurator
2
3 from sqlalchemy import engine_from_config
4
5 from .models import DBSession, Base
6
7 def main(global_config, **settings):
8     engine = engine_from_config(settings, 'sqlalchemy.')
9     DBSession.configure(bind=engine)
10    Base.metadata.bind = engine
11
12    config = Configurator(settings=settings,
13                          root_factory='tutorial.models.Root')
14    config.include('pyramid_chameleon')
15    config.add_route('wiki_view', '/')
16    config.add_route('wiki_page_add', '/add')
17    config.add_route('wiki_page_view', '/{uid}')
18    config.add_route('wiki_page_edit', '/{uid}/edit')
19    config.add_static_view('deform_static', 'deform:static/')
20    config.scan('.views')
21    return config.make_wsgi_app()
```

5. Make a command-line script at `databases/tutorial/initialize_db.py` to initialize the database:

```
1 import os
2 import sys
3 import transaction
4
5 from sqlalchemy import engine_from_config
6
7 from pyramid.paster import (
8     get_appsettings,
9     setup_logging,
10 )
```

```
11
12 from .models import (
13     DBSession,
14     Page,
15     Base,
16 )
17
18
19 def usage(argv):
20     cmd = os.path.basename(argv[0])
21     print('usage: %s <config_uri>\n'
22           '(example: "%s development.ini")' % (cmd, cmd))
23     sys.exit(1)
24
25
26 def main(argv=sys.argv):
27     if len(argv) != 2:
28         usage(argv)
29     config_uri = argv[1]
30     setup_logging(config_uri)
31     settings = get_appsettings(config_uri)
32     engine = engine_from_config(settings, 'sqlalchemy.')
33     DBSession.configure(bind=engine)
34     Base.metadata.create_all(engine)
35     with transaction.manager:
36         model = Page(title='Root', body='<p>Root</p>')
37         DBSession.add(model)
```

6. Since `setup.py` changed, we now run it:

```
$ $VENV/bin/pip install -e .
```

7. The script references some models in `databases/tutorial/models.py`:

```
1 from pyramid.security import Allow, Everyone
2
3 from sqlalchemy import (
4     Column,
5     Integer,
6     Text,
7 )
8
9 from sqlalchemy.ext.declarative import declarative_base
10
```

```

11 from sqlalchemy.orm import (
12     scoped_session,
13     sessionmaker,
14 )
15
16 from zope.sqlalchemy import ZopeTransactionExtension
17
18 DBSession = scoped_session(
19     sessionmaker(extension=ZopeTransactionExtension()))
20 Base = declarative_base()
21
22
23 class Page(Base):
24     __tablename__ = 'wikipages'
25     uid = Column(Integer, primary_key=True)
26     title = Column(Text, unique=True)
27     body = Column(Text)
28
29
30 class Root(object):
31     __acl__ = [(Allow, Everyone, 'view'),
32               (Allow, 'group:editors', 'edit')]
33
34     def __init__(self, request):
35         pass

```

8. Let's run this console script, thus producing our database and table:

```

$ $VENV/bin/initialize_tutorial_db development.ini

2016-04-16 13:01:33,055 INFO [sqlalchemy.engine.base.
↳Engine][MainThread] SELECT CAST('test plain returns' AS VARCHAR(60))
↳AS anon_1
2016-04-16 13:01:33,055 INFO [sqlalchemy.engine.base.
↳Engine][MainThread] ()
2016-04-16 13:01:33,056 INFO [sqlalchemy.engine.base.
↳Engine][MainThread] SELECT CAST('test unicode returns' AS
↳VARCHAR(60)) AS anon_1
2016-04-16 13:01:33,056 INFO [sqlalchemy.engine.base.
↳Engine][MainThread] ()
2016-04-16 13:01:33,057 INFO [sqlalchemy.engine.base.
↳Engine][MainThread] PRAGMA table_info("wikipages")
2016-04-16 13:01:33,057 INFO [sqlalchemy.engine.base.
↳Engine][MainThread] ()
2016-04-16 13:01:33,058 INFO [sqlalchemy.engine.base.
↳Engine][MainThread]

```

```
CREATE TABLE wikipages (  
    uid INTEGER NOT NULL,  
    title TEXT,  
    body TEXT,  
    PRIMARY KEY (uid),  
    UNIQUE (title)  
)  
  
2016-04-16 13:01:33,058 INFO [sqlalchemy.engine.base.  
→Engine][MainThread] ()  
2016-04-16 13:01:33,059 INFO [sqlalchemy.engine.base.  
→Engine][MainThread] COMMIT  
2016-04-16 13:01:33,062 INFO [sqlalchemy.engine.base.  
→Engine][MainThread] BEGIN (implicit)  
2016-04-16 13:01:33,062 INFO [sqlalchemy.engine.base.  
→Engine][MainThread] INSERT INTO wikipages (title, body) VALUES (?, ?)  
2016-04-16 13:01:33,063 INFO [sqlalchemy.engine.base.  
→Engine][MainThread] ('Root', '<p>Root</p>')  
2016-04-16 13:01:33,063 INFO [sqlalchemy.engine.base.  
→Engine][MainThread] COMMIT
```

9. With our data now driven by SQLAlchemy queries, we need to update our databases/
tutorial/views.py:

```
1 import colander  
2 import deform.widget  
3  
4 from pyramid.httpexceptions import HTTPFound  
5 from pyramid.view import view_config  
6  
7 from .models import DBSession, Page  
8  
9  
10 class WikiPage(colander.MappingSchema):  
11     title = colander.SchemaNode(colander.String())  
12     body = colander.SchemaNode(  
13         colander.String(),  
14         widget=deform.widget.RichTextWidget()  
15     )  
16  
17  
18 class WikiViews(object):  
19     def __init__(self, request):  
20         self.request = request
```

```

21
22     @property
23     def wiki_form(self):
24         schema = WikiPage()
25         return deform.Form(schema, buttons=('submit',))
26
27     @property
28     def reqts(self):
29         return self.wiki_form.get_widget_resources()
30
31     @view_config(route_name='wiki_view', renderer='wiki_view.pt')
32     def wiki_view(self):
33         pages = DBSession.query(Page).order_by(Page.title)
34         return dict(title='Wiki View', pages=pages)
35
36     @view_config(route_name='wikipage_add',
37                  renderer='wikipage_addedit.pt')
38     def wikipage_add(self):
39         form = self.wiki_form.render()
40
41         if 'submit' in self.request.params:
42             controls = self.request.POST.items()
43             try:
44                 appstruct = self.wiki_form.validate(controls)
45             except deform.ValidationFailure as e:
46                 # Form is NOT valid
47                 return dict(form=e.render())
48
49                 # Add a new page to the database
50                 new_title = appstruct['title']
51                 new_body = appstruct['body']
52                 DBSession.add(Page(title=new_title, body=new_body))
53
54                 # Get the new ID and redirect
55                 page = DBSession.query(Page).filter_by(title=new_title).
one()
56                 new_uid = page.uid
57
58                 url = self.request.route_url('wikipage_view', uid=new_uid)
59                 return HTTPFound(url)
60
61         return dict(form=form)
62
63
64     @view_config(route_name='wikipage_view', renderer='wikipage_view.pt
one()

```

```
65     def wiki_page_view(self):
66         uid = int(self.request.matchdict['uid'])
67         page = DBSession.query(Page).filter_by(uid=uid).one()
68         return dict(page=page)
69
70
71     @view_config(route_name='wiki_page_edit',
72                  renderer='wiki_page_addedit.pt')
73     def wiki_page_edit(self):
74         uid = int(self.request.matchdict['uid'])
75         page = DBSession.query(Page).filter_by(uid=uid).one()
76
77         wiki_form = self.wiki_form
78
79         if 'submit' in self.request.params:
80             controls = self.request.POST.items()
81             try:
82                 appstruct = wiki_form.validate(controls)
83             except deform.ValidationFailure as e:
84                 return dict(page=page, form=e.render())
85
86             # Change the content and redirect to the view
87             page.title = appstruct['title']
88             page.body = appstruct['body']
89             url = self.request.route_url('wiki_page_view', uid=uid)
90             return HTTPFound(url)
91
92         form = self.wiki_form.render(dict(
93             uid=page.uid, title=page.title, body=page.body)
94         )
95
96         return dict(page=page, form=form)
```

10. Our tests in `databases/tutorial/tests.py` changed to include SQLAlchemy bootstrapping:

```
1 import unittest
2 import transaction
3
4 from pyramid import testing
5
6
7 def _initTestingDB():
8     from sqlalchemy import create_engine
9     from .models import (
```

```

10         DBSession,
11         Page,
12         Base
13     )
14     engine = create_engine('sqlite://')
15     Base.metadata.create_all(engine)
16     DBSession.configure(bind=engine)
17     with transaction.manager:
18         model = Page(title='FrontPage', body='This is the front page')
19         DBSession.add(model)
20     return DBSession
21
22
23 class WikiViewTests(unittest.TestCase):
24     def setUp(self):
25         self.session = _initTestingDB()
26         self.config = testing.setUp()
27
28     def tearDown(self):
29         self.session.remove()
30         testing.tearDown()
31
32     def test_wiki_view(self):
33         from tutorial.views import WikiViews
34
35         request = testing.DummyRequest()
36         inst = WikiViews(request)
37         response = inst.wiki_view()
38         self.assertEqual(response['title'], 'Wiki View')
39
40
41 class WikiFunctionalTests(unittest.TestCase):
42     def setUp(self):
43         from pyramid.paster import get_app
44         app = get_app('development.ini')
45         from webtest import TestApp
46         self.testapp = TestApp(app)
47
48     def tearDown(self):
49         from .models import DBSession
50         DBSession.remove()
51
52     def test_it(self):
53         res = self.testapp.get('/', status=200)
54         self.assertIn(b'Wiki: View', res.body)
55         res = self.testapp.get('/add', status=200)

```



```
56 |         self.assertIn(b'Add/Edit', res.body)
```

11. Run the tests in your package using `py.test`:

```
$ $VENV/bin/py.test tutorial/tests.py -q
..
2 passed in 1.41 seconds
```

12. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

13. Open `http://localhost:6543/` in a browser.

Analysis

Let's start with the dependencies. We made the decision to use `SQLAlchemy` to talk to our database. We also, though, installed `pyramid_tm` and `zope.sqlalchemy`. Why?

Pyramid has a strong orientation towards support for transactions. Specifically, you can install a transaction manager into your application either as middleware or a Pyramid “tween”. Then, just before you return the response, all transaction-aware parts of your application are executed.

This means Pyramid view code usually doesn't manage transactions. If your view code or a template generates an error, the transaction manager aborts the transaction. This is a very liberating way to write code.

The `pyramid_tm` package provides a “tween” that is configured in the `development.ini` configuration file. That installs it. We then need a package that makes `SQLAlchemy`, and thus the RDBMS transaction manager, integrate with the Pyramid transaction manager. That's what `zope.sqlalchemy` does.

Where do we point at the location on disk for the SQLite file? In the configuration file. This lets consumers of our package change the location in a safe (non-code) way. That is, in configuration. This configuration-oriented approach isn't required in Pyramid; you can still make such statements in your `__init__.py` or some companion module.

The `initialize_tutorial_db` is a nice example of framework support. You point your setup at the location of some `[console_scripts]`, and these get generated into your virtual environment's `bin`

directory. Our console script follows the pattern of being fed a configuration file with all the bootstrapping. It then opens SQLAlchemy and creates the root of the wiki, which also makes the SQLite file. Note the `with transaction.manager` part that puts the work in the scope of a transaction, as we aren't inside a web request where this is done automatically.

The `models.py` does a little bit of extra work to hook up SQLAlchemy into the Pyramid transaction manager. It then declares the model for a `Page`.

Our views have changes primarily around replacing our dummy dictionary-of-dictionaries data with proper database support: list the rows, add a row, edit a row, and delete a row.

Extra credit

1. Why all this code? Why can't I just type two lines and have magic ensue?
2. Give a try at a button that deletes a wiki page.

20: Logins with authentication

Login views that authenticate a username and password against a list of users.

Background

Most web applications have URLs that allow people to add/edit/delete content via a web browser. Time to add *security* to the application. In this first step we introduce authentication. That is, logging in and logging out, using Pyramid's rich facilities for pluggable user storage.

In the next step we will introduce protection of resources with authorization security statements.

Objectives

- Introduce the Pyramid concepts of authentication.
- Create login and logout views.

Steps

1. We are going to use the view classes step as our starting point:

```
$ cd ../; cp -r view_classes authentication; cd authentication
```

2. Add `bcrypt` as a dependency in `authentication/setup.py`:

```
1 from setuptools import setup
2
3 requires = [
4     'pyramid',
5     'pyramid_chameleon',
6     'bcrypt'
7 ]
8
9 setup(name='tutorial',
10       install_requires=requires,
11       entry_points="""\
12     [paste.app_factory]
13     main = tutorial:main
14     """,
15 )
```

3. We can now install our project in development mode:

```
$ $VENV/bin/pip install -e .
```

4. Put the security hash in the `authentication/development.ini` configuration file as `tutorial.secret` instead of putting it in the code:

```
1 [app:main]
2 use = egg:tutorial
3 pyramid.reload_templates = true
4 pyramid.includes =
5     pyramid_debugtoolbar
6 tutorial.secret = 98zd
7
8 [server:main]
9 use = egg:pyramid#wsgiref
10 host = 0.0.0.0
11 port = 6543
```

5. Get authentication (and for now, authorization policies) and login route into the *configurator* in `authentication/tutorial/__init__.py`:

```

1 from pyramid.authentication import AuthTktAuthenticationPolicy
2 from pyramid.authorization import ACLAuthorizationPolicy
3 from pyramid.config import Configurator
4
5 from .security import groupfinder
6
7
8 def main(global_config, **settings):
9     config = Configurator(settings=settings)
10    config.include('pyramid_chameleon')
11
12    # Security policies
13    authn_policy = AuthTktAuthenticationPolicy(
14        settings['tutorial.secret'], callback=groupfinder,
15        hashalg='sha512')
16    authz_policy = ACLAuthorizationPolicy()
17    config.set_authentication_policy(authn_policy)
18    config.set_authorization_policy(authz_policy)
19
20    config.add_route('home', '/')
21    config.add_route('hello', '/howdy')
22    config.add_route('login', '/login')
23    config.add_route('logout', '/logout')
24    config.scan('.views')
25    return config.make_wsgi_app()

```

6. Create an authentication/tutorial/security.py module that can find our user information by providing an *authentication policy callback*:

```

1 import bcrypt
2
3
4 def hash_password(pw):
5     pwhash = bcrypt.hashpw(pw.encode('utf8'), bcrypt.gensalt())
6     return pwhash.decode('utf8')
7
8 def check_password(pw, hashed_pw):
9     expected_hash = hashed_pw.encode('utf8')
10    return bcrypt.checkpw(pw.encode('utf8'), expected_hash)
11
12
13 USERS = {'editor': hash_password('editor'),
14         'viewer': hash_password('viewer')}
15 GROUPS = {'editor': ['group:editors']}
16

```

```
17
18 def groupfinder(userid, request):
19     if userid in USERS:
20         return GROUPS.get(userid, [])
```

7. Update the views in authentication/tutorial/views.py:

```
1 from pyramid.httpexceptions import HTTPFound
2 from pyramid.security import (
3     remember,
4     forget,
5 )
6
7 from pyramid.view import (
8     view_config,
9     view_defaults
10 )
11
12 from .security import (
13     USERS,
14     check_password
15 )
16
17
18 @view_defaults(renderer='home.pt')
19 class TutorialViews:
20     def __init__(self, request):
21         self.request = request
22         self.logged_in = request.authenticated_userid
23
24     @view_config(route_name='home')
25     def home(self):
26         return {'name': 'Home View'}
27
28     @view_config(route_name='hello')
29     def hello(self):
30         return {'name': 'Hello View'}
31
32     @view_config(route_name='login', renderer='login.pt')
33     def login(self):
34         request = self.request
35         login_url = request.route_url('login')
36         referrer = request.url
37         if referrer == login_url:
38             referrer = '/' # never use login form itself as came_from
39         came_from = request.params.get('came_from', referrer)
```

```

40     message = ''
41     login = ''
42     password = ''
43     if 'form.submitted' in request.params:
44         login = request.params['login']
45         password = request.params['password']
46         if check_password(password, USERS.get(login)):
47             headers = remember(request, login)
48             return HTTPFound(location=came_from,
49                             headers=headers)
50     message = 'Failed login'
51
52     return dict(
53         name='Login',
54         message=message,
55         url=request.application_url + '/login',
56         came_from=came_from,
57         login=login,
58         password=password,
59     )
60
61 @view_config(route_name='logout')
62 def logout(self):
63     request = self.request
64     headers = forget(request)
65     url = request.route_url('home')
66     return HTTPFound(location=url,
67                     headers=headers)

```

8. Add a login template at authentication/tutorial/login.pt:

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <title>Quick Tutorial: ${name}</title>
5  </head>
6  <body>
7  <h1>Login</h1>
8  <span tal:replace="message"/>
9
10 <form action="${url}" method="post">
11     <input type="hidden" name="came_from"
12         value="${came_from}"/>
13     <label for="login">Username</label>
14     <input type="text" id="login"
15         name="login"

```

```
16         value="{login}"/><br/>
17     <label for="password">Password</label>
18     <input type="password" id="password"
19         name="password"
20         value="{password}"/><br/>
21     <input type="submit" name="form.submitted"
22         value="Log In"/>
23 </form>
24 </body>
25 </html>
```

9. Provide a login/logout box in authentication/tutorial/home.pt:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <title>Quick Tutorial: ${name}</title>
5  </head>
6  <body>
7
8  <div>
9      <a tal:condition="view.logged_in is None"
10         href="{request.application_url}/login">Log In</a>
11      <a tal:condition="view.logged_in is not None"
12         href="{request.application_url}/logout">Logout</a>
13  </div>
14
15  <h1>Hi ${name}</h1>
16  <p>Visit <a href="{request.route_url('hello')}">hello</a></p>
17 </body>
18 </html>
```

10. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

11. Open `http://localhost:6543/` in a browser.
12. Click the “Log In” link.
13. Submit the login form with the username editor and the password editor.
14. Note that the “Log In” link has changed to “Logout”.
15. Click the “Logout” link.


Analysis

Unlike many web frameworks, Pyramid includes a built-in but optional security model for authentication and authorization. This security system is intended to be flexible and support many needs. In this security model, authentication (who are you) and authorization (what are you allowed to do) are not just pluggable, but decoupled. To learn one step at a time, we provide a system that identifies users and lets them log out.

In this example we chose to use the bundled *AuthTktAuthenticationPolicy* policy. We enabled it in our configuration and provided a ticket-signing secret in our INI file.

Our view class grew a login view. When you reached it via a GET request, it returned a login form. When reached via POST, it processed the submitted username and password against the “groupfinder” callable that we registered in the configuration.

The function `hash_password` uses a one-way hashing algorithm with a salt on the user’s password via `bcrypt`, instead of storing the password in plain text. This is considered to be a “best practice” for security.

 There are alternative libraries to `bcrypt` if it is an issue on your system. Just make sure that the library uses an algorithm approved for storing passwords securely.

The function `check_password` will compare the two hashed values of the submitted password and the user’s password stored in the database. If the hashed values are equivalent, then the user is authenticated, else authentication fails.

In our template, we fetched the `logged_in` value from the view class. We use this to calculate the logged-in user, if any. In the template we can then choose to show a login link to anonymous visitors or a logout link to logged-in users.

Extra credit

1. What is the difference between a user and a principal?
2. Can I use a database behind my `groupfinder` to look up principals?
3. Once I am logged in, does any user-centric information get jammed onto each request? Use `import pdb; pdb.set_trace()` to answer this.

See also:

See also *Security*, *AuthTktAuthenticationPolicy*, `bcrypt`

21: Protecting Resources With Authorization

Assign security statements to resources describing the permissions required to perform an operation.

Background

Our application has URLs that allow people to add/edit/delete content via a web browser. Time to add security to the application. Let's protect our add/edit views to require a login (username of `editor` and password of `editor`). We will allow the other views to continue working without a password.

Objectives

- Introduce the Pyramid concepts of authentication, authorization, permissions, and access control lists (ACLs).
- Make a *root factory* that returns an instance of our class for the top of the application.
- Assign security statements to our root resource.
- Add a permissions predicate on a view.
- Provide a *Forbidden view* to handle visiting a URL without adequate permissions.

Steps

1. We are going to use the authentication step as our starting point:

```
$ cd ..; cp -r authentication authorization; cd authorization
$ $VENV/bin/pip install -e .
```

2. Start by changing `authorization/tutorial/__init__.py` to specify a root factory to the *configurator*:

```

1 from pyramid.authentication import AuthTktAuthenticationPolicy
2 from pyramid.authorization import ACLAuthorizationPolicy
3 from pyramid.config import Configurator
4
5 from .security import groupfinder
6
7
8 def main(global_config, **settings):
9     config = Configurator(settings=settings,
10                          root_factory='.resources.Root')
11     config.include('pyramid_chameleon')
12
13     # Security policies
14     authn_policy = AuthTktAuthenticationPolicy(
15         settings['tutorial.secret'], callback=groupfinder,
16         hashalg='sha512')
17     authz_policy = ACLAuthorizationPolicy()
18     config.set_authentication_policy(authn_policy)
19     config.set_authorization_policy(authz_policy)
20
21     config.add_route('home', '/')
22     config.add_route('hello', '/howdy')
23     config.add_route('login', '/login')
24     config.add_route('logout', '/logout')
25     config.scan('.views')
26     return config.make_wsgi_app()

```

3. That means we need to implement `authorization/tutorial/resources.py`:

```

1 from pyramid.security import Allow, Everyone
2
3
4 class Root(object):
5     __acl__ = [(Allow, Everyone, 'view'),
6               (Allow, 'group:editors', 'edit')]
7
8     def __init__(self, request):
9         pass

```

4. Change `authorization/tutorial/views.py` to require the edit permission on the hello view and implement the forbidden view:

```

1 from pyramid.httpexceptions import HTTPFound
2 from pyramid.security import (

```

```
3     remember,
4     forget,
5     )
6
7 from pyramid.view import (
8     view_config,
9     view_defaults,
10    forbidden_view_config
11 )
12
13 from .security import (
14     USERS,
15     check_password
16 )
17
18
19 @view_defaults(renderer='home.pt')
20 class TutorialViews:
21     def __init__(self, request):
22         self.request = request
23         self.logged_in = request.authenticated_userid
24
25     @view_config(route_name='home')
26     def home(self):
27         return {'name': 'Home View'}
28
29     @view_config(route_name='hello', permission='edit')
30     def hello(self):
31         return {'name': 'Hello View'}
32
33     @view_config(route_name='login', renderer='login.pt')
34     @forbidden_view_config(renderer='login.pt')
35     def login(self):
36         request = self.request
37         login_url = request.route_url('login')
38         referrer = request.url
39         if referrer == login_url:
40             referrer = '/' # never use login form itself as came_from
41         came_from = request.params.get('came_from', referrer)
42         message = ''
43         login = ''
44         password = ''
45         if 'form.submitted' in request.params:
46             login = request.params['login']
47             password = request.params['password']
48             if check_password(password, USERS.get(login)):
```

```

49         headers = remember(request, login)
50         return HTTPFound(location=came_from,
51                           headers=headers)
52     message = 'Failed login'
53
54     return dict(
55         name='Login',
56         message=message,
57         url=request.application_url + '/login',
58         came_from=came_from,
59         login=login,
60         password=password,
61     )
62
63 @view_config(route_name='logout')
64 def logout(self):
65     request = self.request
66     headers = forget(request)
67     url = request.route_url('home')
68     return HTTPFound(location=url,
69                       headers=headers)

```

5. Run your Pyramid application with:

```
$ $VENV/bin/pserve development.ini --reload
```

6. Open <http://localhost:6543/> in a browser.

7. If you are still logged in, click the “Log Out” link.

8. Visit <http://localhost:6543/howdy> in a browser. You should be asked to login.

Analysis

This simple tutorial step can be boiled down to the following:

- A view can require a *permission* (edit).
- The context for our view (the Root) has an access control list (ACL).
- This ACL says that the `edit` permission is available on `Root` to the `group:editors` *principal*.

- The registered `groupfinder` answers whether a particular user (`editor`) has a particular group (`group:editors`).

In summary, `hello` wants `edit` permission, `Root` says `group:editors` has `edit` permission.

Of course, this only applies on `Root`. Some other part of the site (a.k.a. *context*) might have a different ACL.

If you are not logged in and visit `/howdy`, you need to get shown the login screen. How does Pyramid know what is the login page to use? We explicitly told Pyramid that the `login` view should be used by decorating the view with `@forbidden_view_config`.

Extra credit

1. Do I have to put a `renderer` in my `@forbidden_view_config` decorator?
2. Perhaps you would like the experience of not having enough permissions (`forbidden`) to be richer. How could you change this?
3. Perhaps we want to store security statements in a database and allow editing via a browser. How might this be done?
4. What if we want different security statements on different kinds of objects? Or on the same kinds of objects, but in different parts of a URL hierarchy?

Indices and tables

- `genindex`
- `modindex`
- `search`

SQLAlchemy + URL dispatch wiki tutorial

This tutorial introduces an *SQLAlchemy* and *URL dispatch*-based Pyramid application to a developer familiar with Python. When the tutorial is finished, the developer will have created a basic wiki application with authentication and authorization.

For cut and paste purposes, the source code for all stages of this tutorial can be browsed on GitHub at `docs/tutorials/wiki2/src`, which corresponds to the same location if you have Pyramid sources.

Background

This version of the Pyramid wiki tutorial presents a Pyramid application that uses technologies which will be familiar to someone with SQL database experience. It uses *SQLAlchemy* as a persistence mechanism and *URL dispatch* to map URLs to code. It can also be followed by people without any prior Python web framework experience.

To code along with this tutorial, the developer will need a UNIX machine with development tools (Mac OS X with XCode, any Linux or BSD variant, etc.) *or* a Windows system of any kind.



This tutorial runs on both Python 2 and 3 without modification.

Have fun!

Design

Following is a quick overview of the design of our wiki application to help us understand the changes that we will be making as we work through the tutorial.

Overall

We choose to use *reStructuredText* markup in the wiki text. Translation from reStructuredText to HTML is provided by the widely used *docutils* Python module. We will add this module to the dependency list in the project's `setup.py` file.

Models

We'll be using an SQLite database to hold our wiki data, and we'll be using *SQLAlchemy* to access the data in this database.

Within the database, we will define two tables:

- The *users* table which will store the *id*, *name*, *password_hash* and *role* of each wiki user.
- The *pages* table, whose elements will store the wiki pages. There are four columns: *id*, *name*, *data* and *creator_id*.

There is a one-to-many relationship between *users* and *pages* tracking the user who created each wiki page defined by the *creator_id* column on the *pages* table.

URLs like `/PageName` will try to find an element in the *pages* table that has a corresponding name.

To add a page to the wiki, a new row is created and the text is stored in *data*.

A page named `FrontPage` containing the text *This is the front page*, will be created when the storage is initialized, and will be used as the wiki home page.

Wiki Views

There will be three views to handle the normal operations of adding, editing, and viewing wiki pages, plus one view for the wiki front page. Two templates will be used, one for viewing, and one for both adding and editing wiki pages.

As of version 1.5 Pyramid no longer ships with templating systems. In this tutorial, we will use *Jinja2*. Jinja2 is a modern and designer-friendly templating language for Python, modeled after Django's templates.

Security

We'll eventually be adding security to our application. To do this, we'll be using a very simple role-based security model. We'll assign a single role category to each user in our system.

basic An authenticated user who can view content and create new pages. A *basic* user may also edit the pages they have created but not pages created by other users.

editor An authenticated user who can create and edit any content in the system.

In order to accomplish this we'll need to define an authentication policy which can identify users by their *userid* and role. Then we'll need to define a page *resource* which contains the appropriate *ACL*:

Action	Principal	Permission
Allow	Everyone	view
Allow	group:basic	create
Allow	group:editors	edit
Allow	<creator of page>	edit

Permission declarations will be added to the views to assert the security policies as each request is handled.

On the security side of the application there are two additional views for handling login and logout as well as two exception views for handling invalid access attempts and unhandled URLs.

Summary

The URL, actions, template, and permission associated to each view are listed in the following table:

URL	Action	View	Template	Permission
/	Redirect to /FrontPage	view_wiki		
/PageName	Display existing page ²	view_page ¹	view.jinja2	view
/PageName/edit_page	Display edit form with existing content. If the form was submitted, redirect to /PageName	edit_page	edit.jinja2	edit
/add_page/PageName	Create the page <i>PageName</i> in storage, display the edit form without content. If the form was submitted, redirect to /PageName	add_page	edit.jinja2	create
/login	Display login form, Forbidden ³ If the form was submitted, authenticate. <ul style="list-style-type: none"> • If authentication succeeds, redirect to the page from which we came. • If authentication fails, display login form with “login failed” message. 	login	login.jinja2	
/logout	Redirect to /FrontPage	logout		

² Pyramid will return a default 404 Not Found page if the page *PageName* does not exist yet.

Installation

Before you begin

This tutorial assumes that you have already followed the steps in *Installing Pyramid*, except **do not create a virtual environment or install Pyramid**. Thereby you will satisfy the following requirements.

- A Python interpreter is installed on your operating system.
- You’ve satisfied the *Requirements for Installing Packages*.

Create directory to contain the project

We need a workspace for our project files.

On UNIX

```
$ mkdir ~/pyramidtut
```

On Windows

```
c:\> mkdir pyramidtut
```

Create and use a virtual Python environment

Next let’s create a virtual environment workspace for our project. We will use the `VENV` environment variable instead of the absolute path of the virtual environment.

On UNIX

¹ This is the default view for a Page context when there is no view name.

³ `pyramid.exceptions.Forbidden` is reached when a user tries to invoke a view that is not authorized by the authorization policy.

```
$ export VENV=~/.pyramidtut
$ python3 -m venv $VENV
```

On Windows

```
c:\> set VENV=c:\pyramidtut
```

Each version of Python uses different paths, so you will need to adjust the path to the command for your Python version.

Python 2.7:

```
c:\> c:\Python27\Scripts\virtualenv %VENV%
```

Python 3.5:

```
c:\> c:\Python35\Scripts\python -m venv %VENV%
```

Upgrade pip and setuptools in the virtual environment

On UNIX

```
$ $VENV/bin/pip install --upgrade pip setuptools
```

On Windows

```
c:\> %VENV%\Scripts\pip install --upgrade pip setuptools
```

Install Pyramid into the virtual Python environment

On UNIX

```
$ $VENV/bin/pip install "pyramid==1.7.6"
```

On Windows

```
c:\> %VENV%\Scripts\pip install "pyramid==1.7.6"
```

Install SQLite3 and its development packages

If you used a package manager to install your Python or if you compiled your Python from source, then you must install SQLite3 and its development packages. If you downloaded your Python as an installer from <https://www.python.org>, then you already have it installed and can skip this step.

If you need to install the SQLite3 packages, then, for example, using the Debian system and `apt-get`, the command would be the following:

```
$ sudo apt-get install libsqlite3-dev
```

Change directory to your virtual Python environment

Change directory to the `pyramidtut` directory, which is both your workspace and your virtual environment.

On UNIX

```
$ cd pyramidtut
```

On Windows

```
c:\> cd pyramidtut
```

Making a project

Your next step is to create a project. For this tutorial we will use the *scaffold* named *alchemy* which generates an application that uses *SQLAlchemy* and *URL dispatch*.

Pyramid supplies a variety of scaffolds to generate sample projects. We will use `pcreate`, a script that comes with Pyramid, to create our project using a scaffold.

By passing `alchemy` into the `pcreate` command, the script creates the files needed to use SQLAlchemy. By passing in our application name `tutorial`, the script inserts that application name into all the required files. For example, `pcreate` creates the `initialize_tutorial_db` in the `pyramidtut/bin` directory.


The below instructions assume your current working directory is “pyramidtut”.

On UNIX

```
$ $VENV/bin/pcreate -s alchemy tutorial
```

On Windows

```
c:\pyramidtut> %VENV%\Scripts\pcreate -s alchemy tutorial
```

 If you are using Windows, the `alchemy` scaffold may not deal gracefully with installation into a location that contains spaces in the path. If you experience startup problems, try putting both the virtual environment and the project into directories that do not contain spaces in their paths.

Installing the project in development mode

In order to do development on the project easily, you must “register” the project as a development egg in your workspace using the `pip install -e .` command. In order to do so, change directory to the tutorial directory that you created in *Making a project*, and run the `pip install -e .` command using the virtual environment Python interpreter.

On UNIX

```
$ cd tutorial
$ $VENV/bin/pip install -e .
```

On Windows

```
c:\pyramidtut> cd tutorial
c:\pyramidtut\tutorial> %VENV%\Scripts\pip install -e .
```

The console will show `pip` checking for packages and installing missing packages. Success executing this command will show a line like the following:

```
Successfully installed Chameleon-2.24 Mako-1.0.4 MarkupSafe-0.23 \
Pygments-2.1.3 SQLAlchemy-1.0.12 pyramid-chameleon-0.3 \
pyramid-debugtoolbar-2.4.2 pyramid-mako-1.0.2 pyramid-tm-0.12.1 \
transaction-1.4.4 tutorial waitress-0.8.10 zope.sqlalchemy-0.7.6
```

Install testing requirements

In order to run tests, we need to install the testing requirements. This is done through our project’s `setup.py` file, in the `tests_require` and `extras_require` stanzas, and by issuing the command below for your operating system.

```
22 tests_require = [
23     'WebTest >= 1.3.1', # py3 compat
24     'pytest', # includes virtualenv
25     'pytest-cov',
26 ]
```

```
45     extras_require={
46         'testing': tests_require,
47     },
```

On UNIX

```
$ $VENV/bin/pip install -e ".[testing]"
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\pip install -e ".[testing]"
```

Run the tests

After you've installed the project in development mode as well as the testing requirements, you may run the tests for the project. The following commands provide options to `py.test` that specify the module for which its tests shall be run, and to run `py.test` in quiet mode.

On UNIX

```
$ $VENV/bin/py.test -q
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\py.test -q
```

For a successful test run, you should see output that ends like this:

```
..
2 passed in 0.44 seconds
```

Expose test coverage information

You can run the `py.test` command to see test coverage information. This runs the tests in the same way that `py.test` does, but provides additional “coverage” information, exposing which lines of your project are covered by the tests.

We’ve already installed the `pytest-cov` package into our virtual environment, so we can run the tests with coverage.

On UNIX

```
$ $VENV/bin/py.test --cov --cov-report=term-missing
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\py.test --cov \
--cov-report=term-missing
```

If successful, you will see output something like this:

```
===== test session starts =====
platform Python 3.5.1, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: /Users/stevepiercy/projects/pyramidtut/tutorial, inifile:
plugins: cov-2.2.1
collected 2 items

tutorial/tests.py ..

----- coverage: platform Python 3.5.1 -----
Name                                     Stmts   Miss  Cover   Missing
-----
tutorial/__init__.py                      8        6    25%    7-12
tutorial/models/__init__.py              22        0   100%
```


tutorial/models/meta.py	5	0	100%	
tutorial/models/mymodel.py	8	0	100%	
tutorial/routes.py	3	2	33%	2-3
tutorial/scripts/__init__.py	0	0	100%	
tutorial/scripts/initializedb.py	26	16	38%	22-25, 29-45
tutorial/views/__init__.py	0	0	100%	
tutorial/views/default.py	12	0	100%	
tutorial/views/notfound.py	4	2	50%	6-7

TOTAL	88	26	70%	
===== 2 passed in 0.57 seconds =====				

Our package doesn't quite have 100% test coverage.

Test and coverage scaffold defaults

Scaffolds include configuration defaults for `py.test` and test coverage. These configuration files are `pytest.ini` and `.coveragerc`, located at the root of your package. Without these defaults, we would need to specify the path to the module on which we want to run tests and coverage.

On UNIX

```
$ $VENV/bin/py.test --cov=tutorial tutorial/tests.py -q
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\py.test --cov=tutorial \  
--cov-report=term-missing tutorial\tests.py -q
```

`py.test` follows conventions for Python test discovery, and the configuration defaults from the scaffold tell `py.test` where to find the module on which we want to run tests and coverage.

See also:

See `py.test`'s documentation for Usage and Invocations or invoke `py.test -h` to see its full set of options.

Initializing the database

We need to use the `initialize_tutorial_db` *console script* to initialize our database.

i The `initialize_tutorial_db` command does not perform a migration, but rather it simply creates missing tables and adds some dummy data. If you already have a database, you should delete it before running `initialize_tutorial_db` again.

Type the following command, making sure you are still in the `tutorial` directory (the directory with a `development.ini` in it):

On UNIX

```
$ $VENV/bin/initialize_tutorial_db development.ini
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\initialize_tutorial_db development.
↪ini
```

The output to your console should be something like this:

```
2016-05-22 04:03:28,888 INFO [sqlalchemy.engine.base.
↪Engine:1192][MainThread] SELECT CAST('test plain returns' AS_
↪VARCHAR(60)) AS anon_1
2016-05-22 04:03:28,888 INFO [sqlalchemy.engine.base.
↪Engine:1193][MainThread] ()
2016-05-22 04:03:28,888 INFO [sqlalchemy.engine.base.
↪Engine:1192][MainThread] SELECT CAST('test unicode returns' AS_
↪VARCHAR(60)) AS anon_1
2016-05-22 04:03:28,889 INFO [sqlalchemy.engine.base.
↪Engine:1193][MainThread] ()
2016-05-22 04:03:28,890 INFO [sqlalchemy.engine.base.
↪Engine:1097][MainThread] PRAGMA table_info("models")
2016-05-22 04:03:28,890 INFO [sqlalchemy.engine.base.
↪Engine:1100][MainThread] ()
2016-05-22 04:03:28,892 INFO [sqlalchemy.engine.base.
↪Engine:1097][MainThread]
```

```
CREATE TABLE models (
    id INTEGER NOT NULL,
    name TEXT,
    value INTEGER,
    CONSTRAINT pk_models PRIMARY KEY (id)
)

2016-05-22 04:03:28,892 INFO [sqlalchemy.engine.base.
↳Engine:1100][MainThread] ()
2016-05-22 04:03:28,893 INFO [sqlalchemy.engine.base.
↳Engine:686][MainThread] COMMIT
2016-05-22 04:03:28,893 INFO [sqlalchemy.engine.base.
↳Engine:1097][MainThread] CREATE UNIQUE INDEX my_index ON models (name)
2016-05-22 04:03:28,893 INFO [sqlalchemy.engine.base.
↳Engine:1100][MainThread] ()
2016-05-22 04:03:28,894 INFO [sqlalchemy.engine.base.
↳Engine:686][MainThread] COMMIT
2016-05-22 04:03:28,896 INFO [sqlalchemy.engine.base.
↳Engine:646][MainThread] BEGIN (implicit)
2016-05-22 04:03:28,897 INFO [sqlalchemy.engine.base.
↳Engine:1097][MainThread] INSERT INTO models (name, value) VALUES (?, ?)
2016-05-22 04:03:28,897 INFO [sqlalchemy.engine.base.
↳Engine:1100][MainThread] ('one', 1)
2016-05-22 04:03:28,898 INFO [sqlalchemy.engine.base.
↳Engine:686][MainThread] COMMIT
```

Success! You should now have a `tutorial.sqlite` file in your current working directory. This is an SQLite database with a single table defined in it (`models`).

Start the application


Start the application.

On UNIX

```
$ $VENV/bin/pserve development.ini --reload
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\pserve development.ini --reload
```

 Your OS firewall, if any, may pop up a dialog asking for authorization to allow python to accept incoming network connections.

If successful, you will see something like this on your console:

```
Starting subprocess with file monitor
Starting server in PID 82349.
serving on http://127.0.0.1:6543
```

This means the server is ready to accept requests.

Visit the application in a browser


In a browser, visit `http://localhost:6543/`. You will see the generated application’s default page.

One thing you’ll notice is the “debug toolbar” icon on right hand side of the page. You can read more about the purpose of the icon at *The Debug Toolbar*. It allows you to get information about your application while you develop.

Decisions the alchemy scaffold has made for you

Creating a project using the alchemy scaffold makes the following assumptions:

- You are willing to use *SQLAlchemy* as a database access tool.
- You are willing to use *URL dispatch* to map URLs to code.
- You want to use *zope.sqlalchemy*, *pyramid_tm*, and the *transaction* packages to scope sessions to requests.
- You want to use *pyramid_jinja2* to render your templates. Different templating engines can be used, but we had to choose one to make this tutorial. See *Available Add-On Template System Bindings* for some options.

 Pyramid supports any persistent storage mechanism (e.g., object database or filesystem files). It also supports an additional mechanism to map URLs to code (*traversal*). However, for the purposes of this tutorial, we’ll only be using *URL dispatch* and *SQLAlchemy*.

Basic Layout

The starter files generated by the `alchemy` scaffold are very basic, but they provide a good orientation for the high-level patterns common to most *URL dispatch*-based Pyramid projects.

Application configuration with `__init__.py`

A directory on disk can be turned into a Python *package* by containing an `__init__.py` file. Even if empty, this marks a directory as a Python package. We use `__init__.py` both as a marker, indicating the directory in which it's contained is a package, and to contain application configuration code.

Open `tutorial/__init__.py`. It should already contain the following:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6     """
7     config = Configurator(settings=settings)
8     config.include('pyramid_jinja2')
9     config.include('.models')
10    config.include('.routes')
11    config.scan()
12    return config.make_wsgi_app()
```

Let's go over this piece-by-piece. First we need some imports to support later code:

```
1 from pyramid.config import Configurator
2
3
```

`__init__.py` defines a function named `main`. Here is the entirety of the `main` function we've defined in our `__init__.py`:

```
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6     """
7     config = Configurator(settings=settings)
8     config.include('pyramid_jinja2')
9     config.include('.models')
```

```
10     config.include('.routes')
11     config.scan()
12     return config.make_wsgi_app()
```

When you invoke the `pserve development.ini` command, the `main` function above is executed. It accepts some settings and returns a *WSGI* application. (See *Startup* for more about `pserve`.)

Next in `main`, construct a *Configurator* object:

```
7     config = Configurator(settings=settings)
```

`settings` is passed to the *Configurator* as a keyword argument with the dictionary values passed as the `**settings` argument. This will be a dictionary of settings parsed from the `.ini` file, which contains deployment-related values, such as `pyramid.reload_templates`, `sqlalchemy.url`, and so on.

Next include *Jinja2* templating bindings so that we can use renderers with the `.jinja2` extension within our project.


```
8     config.include('pyramid_jinja2')
```

Next include the the package `models` using a dotted Python path. The exact setup of the models will be covered later.

```
9     config.include('.models')
```

Next include the `routes` module using a dotted Python path. This module will be explained in the next section.

```
10    config.include('.routes')
```

 Pyramid's `pyramid.config.Configurator.include()` method is the primary mechanism for extending the configurator and breaking your code into feature-focused modules.

`main` next calls the `scan` method of the configurator (`pyramid.config.Configurator.scan()`), which will recursively scan our `tutorial` package, looking for `@view_config` and other special decorators. When it finds a `@view_config` decorator, a view configuration will be registered, allowing one of our application URLs to be mapped to some code.

```
11 config.scan()
```

Finally `main` is finished configuring things, so it uses the `pyramid.config.Configurator.make_wsgi_app()` method to return a *WSGI* application:

```
12 return config.make_wsgi_app()
```

Route declarations

Open the `tutorial/routes.py` file. It should already contain the following:

```
1 def includeme(config):
2     config.add_static_view('static', 'static', cache_max_age=3600)
3     config.add_route('home', '/')
```

On line 2, we call `pyramid.config.Configurator.add_static_view()` with three arguments: `static` (the name), `static` (the path), and `cache_max_age` (a keyword argument).

This registers a static resource view which will match any URL that starts with the prefix `/static` (by virtue of the first argument to `add_static_view`). This will serve up static resources for us from within the `static` directory of our `tutorial` package, in this case via `http://localhost:6543/static/` and below (by virtue of the second argument to `add_static_view`). With this declaration, we're saying that any URL that starts with `/static` should go to the static view; any remainder of its path (e.g., the `/foo` in `/static/foo`) will be used to compose a path to a static file resource, such as a CSS file.

On line 3, the module registers a *route configuration* via the `pyramid.config.Configurator.add_route()` method that will be used when the URL is `/`. Since this route has a pattern equaling `/`, it is the route that will be matched when the URL `/` is visited, e.g., `http://localhost:6543/`.

View declarations via the `views` package

The main function of a web framework is mapping each URL pattern to code (a *view callable*) that is executed when the requested URL matches the corresponding *route*. Our application uses the `pyramid.view.view_config()` decorator to perform this mapping.

Open `tutorial/views/default.py` in the `views` package. It should already contain the following:

```

1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 from sqlalchemy.exc import DBAPIError
5
6 from ..models import MyModel
7
8
9 @view_config(route_name='home', renderer='../templates/mytemplate.jinja2')
10 def my_view(request):
11     try:
12         query = request.dbsession.query(MyModel)
13         one = query.filter(MyModel.name == 'one').first()
14     except DBAPIError:
15         return Response(db_err_msg, content_type='text/plain', status=500)
16     return {'one': one, 'project': 'tutorial'}
17
18
19 db_err_msg = """\
20 Pyramid is having a problem using your SQL database. The problem
21 might be caused by one of the following things:
22
23 1. You may need to run the "initialize_tutorial_db" script
24    to initialize your database tables. Check your virtual
25    environment's "bin" directory for this script and try to run it.
26
27 2. Your database server may not be running. Check that the
28    database server referred to by the "sqlalchemy.url" setting in
29    your "development.ini" file is running.
30
31 After you fix the problem, please restart the Pyramid application to
32 try it again.
33 """

```

The important part here is that the `@view_config` decorator associates the function it decorates (`my_view`) with a *view configuration*, consisting of:

- a `route_name` (`home`)
- a `renderer`, which is a template from the `templates` subdirectory of the package.

When the pattern associated with the `home` view is matched during a request, `my_view()` will be executed. `my_view()` returns a dictionary; the renderer will use the `templates/mytemplate.jinja2` template to create a response based on the values in the dictionary.

Note that `my_view()` accepts a single argument named `request`. This is the standard call signature for a Pyramid *view callable*.

Remember in our `__init__.py` when we executed the `pyramid.config.Configurator.scan()` method `config.scan()`? The purpose of calling the `scan` method was to find and process this `@view_config` decorator in order to create a view configuration within our application. Without being processed by `scan`, the decorator effectively does nothing. `@view_config` is inert without being detected via a `scan`.

The sample `my_view()` created by the scaffold uses a `try:` and `except:` clause to detect if there is a problem accessing the project database and provide an alternate error response. That response will include the text shown at the end of the file, which will be displayed in the browser to inform the user about possible actions to take to solve the problem.

Content models with the `models` package

In an SQLAlchemy-based application, a *model* object is an object composed by querying the SQL database. The `models` package is where the alchemy scaffold put the classes that implement our models.

First, open `tutorial/models/meta.py`, which should already contain the following:

```
1 from sqlalchemy.ext.declarative import declarative_base
2 from sqlalchemy.schema import MetaData
3
4 # Recommended naming convention used by Alembic, as various different_
5 # ↪ database
6 # providers will autogenerate vastly different names making migrations more
7 # difficult. See: http://alembic.zzzcomputing.com/en/latest/naming.html
8 NAMING_CONVENTION = {
9     "ix": "ix_%(column_0_label)s",
10    "uq": "uq_%(table_name)s_%(column_0_name)s",
11    "ck": "ck_%(table_name)s_%(constraint_name)s",
12    "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
13    "pk": "pk_%(table_name)s"
14 }
15
16 metadata = MetaData(naming_convention=NAMING_CONVENTION)
17 Base = declarative_base(metadata=metadata)
```

`meta.py` contains imports and support code for defining the models. We create a dictionary `NAMING_CONVENTION` as well for consistent naming of support objects like indices and constraints.

```

1 from sqlalchemy.ext.declarative import declarative_base
2 from sqlalchemy.schema import MetaData
3
4 # Recommended naming convention used by Alembic, as various different_
  ↳ database
5 # providers will autogenerate vastly different names making migrations more
6 # difficult. See: http://alembic.zzzcomputing.com/en/latest/naming.html
7 NAMING_CONVENTION = {
8     "ix": "ix_%(column_0_label)s",
9     "uq": "uq_%(table_name)s_%(column_0_name)s",
10    "ck": "ck_%(table_name)s_%(constraint_name)s",
11    "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
12    "pk": "pk_%(table_name)s"
13 }
14

```

Next we create a metadata object from the class `sqlalchemy.schema.MetaData`, using `NAMING_CONVENTION` as the value for the `naming_convention` argument.

A `MetaData` object represents the table and other schema definitions for a single database. We also need to create a declarative Base object to use as a base class for our models. Our models will inherit from this Base, which will attach the tables to the metadata we created, and define our application's database schema.

```

15 metadata = MetaData(naming_convention=NAMING_CONVENTION)
16 Base = declarative_base(metadata=metadata)

```

Next open `tutorial/models/mymodel.py`, which should already contain the following:

```

1 from sqlalchemy import (
2     Column,
3     Index,
4     Integer,
5     Text,
6 )
7
8 from .meta import Base
9
10
11 class MyModel(Base):
12     __tablename__ = 'models'
13     id = Column(Integer, primary_key=True)
14     name = Column(Text)

```

```

15     value = Column(Integer)
16
17
18 Index('my_index', MyModel.name, unique=True, mysql_length=255)

```

Notice we've defined the `models` as a package to make it straightforward for defining models in separate modules. To give a simple example of a model class, we have defined one named `MyModel` in `mymodel.py`:

```

11 class MyModel(Base):
12     __tablename__ = 'models'
13     id = Column(Integer, primary_key=True)
14     name = Column(Text)
15     value = Column(Integer)

```

Our example model does not require an `__init__` method because SQLAlchemy supplies for us a default constructor, if one is not already present, which accepts keyword arguments of the same name as that of the mapped attributes.



Example usage of MyModel:

```
johnny = MyModel(name="John Doe", value=10)
```

The `MyModel` class has a `__tablename__` attribute. This informs SQLAlchemy which table to use to store the data representing instances of this class.

Finally, open `tutorial/models/__init__.py`, which should already contain the following:

```

1 from sqlalchemy import engine_from_config
2 from sqlalchemy.orm import sessionmaker
3 from sqlalchemy.orm import configure_mappers
4 import zope.sqlalchemy
5
6 # import or define all models here to ensure they are attached to the
7 # Base.metadata prior to any initialization routines
8 from .mymodel import MyModel # noqa
9
10 # run configure_mappers after defining all of the models to ensure
11 # all relationships can be setup
12 configure_mappers()

```

```

13
14
15 def get_engine(settings, prefix='sqlalchemy.'):
16     return engine_from_config(settings, prefix)
17
18
19 def get_session_factory(engine):
20     factory = sessionmaker()
21     factory.configure(bind=engine)
22     return factory
23
24
25 def get_tm_session(session_factory, transaction_manager):
26     """
27     Get a ``sqlalchemy.orm.Session`` instance backed by a transaction.
28
29     This function will hook the session to the transaction manager which
30     will take care of committing any changes.
31
32     - When using pyramid_tm it will automatically be committed or aborted
33       depending on whether an exception is raised.
34
35     - When using scripts you should wrap the session in a manager yourself.
36       For example::
37
38         import transaction
39
40         engine = get_engine(settings)
41         session_factory = get_session_factory(engine)
42         with transaction.manager:
43             dbsession = get_tm_session(session_factory, transaction.
44 ↪manager)
45
46     """
47     dbsession = session_factory()
48     zope.sqlalchemy.register(
49         dbsession, transaction_manager=transaction_manager)
50     return dbsession
51
52 def includeme(config):
53     """
54     Initialize the model for a Pyramid app.
55
56     Activate this setup using ``config.include('tutorial.models')``.
57

```

```
58     """
59     settings = config.get_settings()
60
61     # use pyramid_tm to hook the transaction lifecycle to the request
62     config.include('pyramid_tm')
63
64     session_factory = get_session_factory(get_engine(settings))
65     config.registry['dbsession_factory'] = session_factory
66
67     # make request.dbsession available for use in Pyramid
68     config.add_request_method(
69         # r.tm is the transaction manager used by pyramid_tm
70         lambda r: get_tm_session(session_factory, r.tm),
71         'dbsession',
72         reify=True
73     )
```

Our `models/__init__.py` module defines the primary API we will use for configuring the database connections within our application, and it contains several functions we will cover below.

As we mentioned above, the purpose of the `models.meta.metadata` object is to describe the schema of the database. This is done by defining models that inherit from the `Base` object attached to that `metadata` object. In Python, code is only executed if it is imported, and so to attach the `models` table defined in `mymodel.py` to the `metadata`, we must import it. If we skip this step, then later, when we run `sqlalchemy.schema.MetaData.create_all()`, the table will not be created because the `metadata` object does not know about it!

Another important reason to import all of the models is that, when defining relationships between models, they must all exist in order for `SQLAlchemy` to find and build those internal mappings. This is why, after importing all the models, we explicitly execute the function `sqlalchemy.orm.configure_mappers()`, once we are sure all the models have been defined and before we start creating connections.

Next we define several functions for connecting to our database. The first and lowest level is the `get_engine` function. This creates an *SQLAlchemy* database engine using `sqlalchemy.engine_from_config()` from the `sqlalchemy` -prefixed settings in the `development.ini` file's `[app:main]` section. This setting is a URI (something like `sqlite://`).

```
15 def get_engine(settings, prefix='sqlalchemy.'):
16     return engine_from_config(settings, prefix)
```

The function `get_session_factory` accepts an *SQLAlchemy* database engine, and creates a `session_factory` from the *SQLAlchemy* class `sqlalchemy.orm.session.sessionmaker`. This `session_factory` is then used for creating sessions bound to the database engine.

```

19 def get_session_factory(engine):
20     factory = sessionmaker()
21     factory.configure(bind=engine)
22     return factory

```

The function `get_tm_session` registers a database session with a transaction manager, and returns a `dbsession` object. With the transaction manager, our application will automatically issue a transaction commit after every request, unless an exception is raised, in which case the transaction will be aborted.

```

25 def get_tm_session(session_factory, transaction_manager):
26     """
27     Get a ``sqlalchemy.orm.Session`` instance backed by a transaction.
28
29     This function will hook the session to the transaction manager which
30     will take care of committing any changes.
31
32     - When using pyramid_tm it will automatically be committed or aborted
33       depending on whether an exception is raised.
34
35     - When using scripts you should wrap the session in a manager yourself.
36       For example::
37
38         import transaction
39
40         engine = get_engine(settings)
41         session_factory = get_session_factory(engine)
42         with transaction.manager:
43             dbsession = get_tm_session(session_factory, transaction.
↪manager)
44
45     """
46     dbsession = session_factory()
47     zope.sqlalchemy.register(
48         dbsession, transaction_manager=transaction_manager)
49     return dbsession

```

Finally, we define an `includeme` function, which is a hook for use with `pyramid.config.Configurator.include()` to activate code in a Pyramid application add-on. It is the code that is executed above when we ran `config.include('.models')` in our application's main function. This function will take the settings from the application, create an engine, and define a `request.dbsession` property, which we can use to do work on behalf of an incoming request to our application.

```
52 def includeme(config):
53     """
54     Initialize the model for a Pyramid app.
55
56     Activate this setup using ``config.include('tutorial.models')``.
57
58     """
59     settings = config.get_settings()
60
61     # use pyramid_tm to hook the transaction lifecycle to the request
62     config.include('pyramid_tm')
63
64     session_factory = get_session_factory(get_engine(settings))
65     config.registry['dbsession_factory'] = session_factory
66
67     # make request.dbsession available for use in Pyramid
68     config.add_request_method(
69         # r.tm is the transaction manager used by pyramid_tm
70         lambda r: get_tm_session(session_factory, r.tm),
71         'dbsession',
72         reify=True
73     )
```

That's about all there is to it regarding models, views, and initialization code in our stock application.

The `Index` import and the `Index` object creation in `mymodel.py` is not required for this tutorial, and will be removed in the next step.

Defining the Domain Model

The first change we'll make to our stock `pcreate`-generated application will be to define a wiki page *domain model*.



There is nothing special about the filename `user.py` or `page.py` except that they are Python modules. A project may have many models throughout its codebase in arbitrarily named modules. Modules implementing models often have `model` in their names or they may live in a Python subpackage of your application package named `models` (as we've done in this tutorial), but this is only a convention and not a requirement.

Declaring dependencies in our `setup.py` file

The models code in our application will depend on a package which is not a dependency of the original “tutorial” application. The original “tutorial” application was generated by the `pcreate` command; it doesn’t know about our custom application requirements.

We need to add a dependency, the `bcrypt` package, to our tutorial package’s `setup.py` file by assigning this dependency to the `requires` parameter in the `setup()` function.

Open `tutorial/setup.py` and edit it to look like the following:

```

1  import os
2
3  from setuptools import setup, find_packages
4
5  here = os.path.abspath(os.path.dirname(__file__))
6  with open(os.path.join(here, 'README.txt')) as f:
7      README = f.read()
8  with open(os.path.join(here, 'CHANGES.txt')) as f:
9      CHANGES = f.read()
10
11  requires = [
12      'bcrypt',
13      'pyramid',
14      'pyramid_jinja2',
15      'pyramid_debugtoolbar',
16      'pyramid_tm',
17      'SQLAlchemy',
18      'transaction',
19      'zope.sqlalchemy',
20      'waitress',
21  ]
22
23  tests_require = [
24      'WebTest >= 1.3.1', # py3 compat
25      'pytest', # includes virtualenv
26      'pytest-cov',
27  ]
28
29  setup(name='tutorial',
30        version='0.0',
31        description='tutorial',
32        long_description=README + '\n\n' + CHANGES,
33        classifiers=[
34            "Programming Language :: Python",

```



```
35         "Framework :: Pyramid",
36         "Topic :: Internet :: WWW/HTTP",
37         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
38     ],
39     author='',
40     author_email='',
41     url='',
42     keywords='web wsgi bfg pylons pyramid',
43     packages=find_packages(),
44     include_package_data=True,
45     zip_safe=False,
46     extras_require={
47         'testing': tests_require,
48     },
49     install_requires=requires,
50     entry_points="""\
51     [paste.app_factory]
52     main = tutorial:main
53     [console_scripts]
54     initialize_tutorial_db = tutorial.scripts.initializedb:main
55     """,
56 )
```

Only the highlighted line needs to be added.

Running `pip install -e .`

Since a new software dependency was added, you will need to run `pip install -e .` again inside the root of the `tutorial` package to obtain and register the newly added dependency distribution.

Make sure your current working directory is the root of the project (the directory in which `setup.py` lives) and execute the following command.

On UNIX:

```
$ cd tutorial
$ $ENV/bin/pip install -e .
```

On Windows:

```
c:\pyramidtut> cd tutorial
c:\pyramidtut\tutorial> %VENV%\Scripts\pip install -e .
```

Success executing this command will end with a line to the console something like this:

```
Successfully installed bcrypt-2.0.0 cffi-1.5.2 pycparser-2.14 tutorial-0.0
```

Remove mymodel.py

Let's delete the file `tutorial/models/mymodel.py`. The `MyModel` class is only a sample and we're not going to use it.

Add user.py

Create a new file `tutorial/models/user.py` with the following contents:

```
1  import bcrypt
2  from sqlalchemy import (
3      Column,
4      Integer,
5      Text,
6  )
7
8  from .meta import Base
9
10
11 class User(Base):
12     """ The SQLAlchemy declarative model class for a User object. """
13     __tablename__ = 'users'
14     id = Column(Integer, primary_key=True)
15     name = Column(Text, nullable=False, unique=True)
16     role = Column(Text, nullable=False)
17
18     password_hash = Column(Text)
19
20     def set_password(self, pw):
21         pwhash = bcrypt.hashpw(pw.encode('utf8'), bcrypt.gensalt())
22         self.password_hash = pwhash.decode('utf8')
23
```

```
24     def check_password(self, pw):
25         if self.password_hash is not None:
26             expected_hash = self.password_hash.encode('utf8')
27             return bcrypt.checkpw(pw.encode('utf8'), expected_hash)
28         return False
```

This is a very basic model for a user who can authenticate with our wiki.

We discussed briefly in the previous chapter that our models will inherit from an SQLAlchemy `sqlalchemy.ext.declarative.declarative_base()`. This will attach the model to our schema.

As you can see, our `User` class has a class-level attribute `__tablename__` which equals the string `users`. Our `User` class will also have class-level attributes named `id`, `name`, `password_hash`, and `role` (all instances of `sqlalchemy.schema.Column`). These will map to columns in the `users` table. The `id` attribute will be the primary key in the table. The `name` attribute will be a text column, each value of which needs to be unique within the column. The `password_hash` is a nullable text attribute that will contain a securely hashed password¹. Finally, the `role` text attribute will hold the role of the user.

There are two helper methods that will help us later when using the user objects. The first is `set_password` which will take a raw password and transform it using `bcrypt` into an irreversible representation, a process known as “hashing”. The second method, `check_password`, will allow us to compare the hashed value of the submitted password against the hashed value of the password stored in the user’s record in the database. If the two hashed values match, then the submitted password is valid, and we can authenticate the user.

We hash passwords so that it is impossible to decrypt them and use them to authenticate in the application. If we stored passwords foolishly in clear text, then anyone with access to the database could retrieve any password to authenticate as any user.

Add `page.py`

Create a new file `tutorial/models/page.py` with the following contents:

¹ We are using the `bcrypt` package from PyPI to hash our passwords securely. There are other one-way hash algorithms for passwords if `bcrypt` is an issue on your system. Just make sure that it’s an algorithm approved for storing passwords versus a generic one-way hash.

```

1 from sqlalchemy import (
2     Column,
3     ForeignKey,
4     Integer,
5     Text,
6 )
7 from sqlalchemy.orm import relationship
8
9 from .meta import Base
10
11
12 class Page(Base):
13     """ The SQLAlchemy declarative model class for a Page object. """
14     __tablename__ = 'pages'
15     id = Column(Integer, primary_key=True)
16     name = Column(Text, nullable=False, unique=True)
17     data = Column(Text, nullable=False)
18
19     creator_id = Column(ForeignKey('users.id'), nullable=False)
20     creator = relationship('User', backref='created_pages')

```

As you can see, our `Page` class is very similar to the `User` defined above, except with attributes focused on storing information about a wiki page, including `id`, `name`, and `data`. The only new construct introduced here is the `creator_id` column, which is a foreign key referencing the `users` table. Foreign keys are very useful at the schema-level, but since we want to relate `User` objects with `Page` objects, we also define a `creator` attribute as an ORM-level mapping between the two tables. SQLAlchemy will automatically populate this value using the foreign key referencing the user. Since the foreign key has `nullable=False`, we are guaranteed that an instance of `page` will have a corresponding `page.creator`, which will be a `User` instance.

Edit `models/__init__.py`

Since we are using a package for our models, we also need to update our `__init__.py` file to ensure that the models are attached to the metadata.

Open the `tutorial/models/__init__.py` file and edit it to look like the following:

```

1 from sqlalchemy import engine_from_config
2 from sqlalchemy.orm import sessionmaker
3 from sqlalchemy.orm import configure_mappers
4 import zope.sqlalchemy
5

```

```
6 # import or define all models here to ensure they are attached to the
7 # Base.metadata prior to any initialization routines
8 from .page import Page # noqa
9 from .user import User # noqa
10
11 # run configure_mappers after defining all of the models to ensure
12 # all relationships can be setup
13 configure_mappers()
14
15
16 def get_engine(settings, prefix='sqlalchemy.'):
17     return engine_from_config(settings, prefix)
18
19
20 def get_session_factory(engine):
21     factory = sessionmaker()
22     factory.configure(bind=engine)
23     return factory
24
25
26 def get_tm_session(session_factory, transaction_manager):
27     """
28     Get a ``sqlalchemy.orm.Session`` instance backed by a transaction.
29
30     This function will hook the session to the transaction manager which
31     will take care of committing any changes.
32
33     - When using pyramid_tm it will automatically be committed or aborted
34       depending on whether an exception is raised.
35
36     - When using scripts you should wrap the session in a manager yourself.
37       For example::
38
39         import transaction
40
41         engine = get_engine(settings)
42         session_factory = get_session_factory(engine)
43         with transaction.manager:
44             dbsession = get_tm_session(session_factory, transaction.
45 ↪manager)
46
47     """
48     dbsession = session_factory()
49     zope.sqlalchemy.register(
50         dbsession, transaction_manager=transaction_manager)
51     return dbsession
```

```

51
52
53 def includeme(config):
54     """
55     Initialize the model for a Pyramid app.
56
57     Activate this setup using ``config.include('tutorial.models')``.
58
59     """
60     settings = config.get_settings()
61
62     # use pyramid_tm to hook the transaction lifecycle to the request
63     config.include('pyramid_tm')
64
65     session_factory = get_session_factory(get_engine(settings))
66     config.registry['dbsession_factory'] = session_factory
67
68     # make request.dbsession available for use in Pyramid
69     config.add_request_method(
70         # r.tm is the transaction manager used by pyramid_tm
71         lambda r: get_tm_session(session_factory, r.tm),
72         'dbsession',
73         reify=True
74     )

```

Here we align our imports with the names of the models, `Page` and `User`.

Edit `scripts/initializedb.py`

We haven't looked at the details of this file yet, but within the `scripts` directory of your `tutorial` package is a file named `initializedb.py`. Code in this file is executed whenever we run the `initialize_tutorial_db` command, as we did in the installation step of this tutorial².

Since we've changed our model, we need to make changes to our `initializedb.py` script. In particular, we'll replace our import of `MyModel` with those of `User` and `Page`. We'll also change the very end of the script to create two `User` objects (`basic` and `editor`) as well as a `Page`, rather than a `MyModel`, and add them to our `dbsession`.

Open `tutorial/scripts/initializedb.py` and edit it to look like the following:

² The command is named `initialize_tutorial_db` because of the mapping defined in the `[console_scripts]` entry point of our project's `setup.py` file.

```
1 import os
2 import sys
3 import transaction
4
5 from pyramid.paster import (
6     get_appsettings,
7     setup_logging,
8 )
9
10 from pyramid.scripts.common import parse_vars
11
12 from ..models.meta import Base
13 from ..models import (
14     get_engine,
15     get_session_factory,
16     get_tm_session,
17 )
18 from ..models import Page, User
19
20
21 def usage(argv):
22     cmd = os.path.basename(argv[0])
23     print('usage: %s <config_uri> [var=value]\n'
24           '(example: "%s development.ini")' % (cmd, cmd))
25     sys.exit(1)
26
27
28 def main(argv=sys.argv):
29     if len(argv) < 2:
30         usage(argv)
31     config_uri = argv[1]
32     options = parse_vars(argv[2:])
33     setup_logging(config_uri)
34     settings = get_appsettings(config_uri, options=options)
35
36     engine = get_engine(settings)
37     Base.metadata.create_all(engine)
38
39     session_factory = get_session_factory(engine)
40
41     with transaction.manager:
42         dbsession = get_tm_session(session_factory, transaction.manager)
43
44         editor = User(name='editor', role='editor')
45         editor.set_password('editor')
46         dbsession.add(editor)
```

```

47     basic = User(name='basic', role='basic')
48     basic.set_password('basic')
49     dbsession.add(basic)
50
51
52     page = Page(
53         name='FrontPage',
54         creator=editor,
55         data='This is the front page',
56     )
57     dbsession.add(page)

```

Only the highlighted lines need to be changed.

Installing the project and re-initializing the database

Because our model has changed, and in order to reinitialize the database, we need to rerun the `initialize_tutorial_db` command to pick up the changes we've made to both the `models.py` file and to the `initializedb.py` file. See *Initializing the database* for instructions.

Success will look something like this:

```

2016-05-22 04:12:09,226 INFO [sqlalchemy.engine.base.
↪Engine:1192][MainThread] SELECT CAST('test plain returns' AS_
↪VARCHAR(60)) AS anon_1
2016-05-22 04:12:09,226 INFO [sqlalchemy.engine.base.
↪Engine:1193][MainThread] ()
2016-05-22 04:12:09,226 INFO [sqlalchemy.engine.base.
↪Engine:1192][MainThread] SELECT CAST('test unicode returns' AS_
↪VARCHAR(60)) AS anon_1
2016-05-22 04:12:09,227 INFO [sqlalchemy.engine.base.
↪Engine:1193][MainThread] ()
2016-05-22 04:12:09,227 INFO [sqlalchemy.engine.base.
↪Engine:1097][MainThread] PRAGMA table_info("users")
2016-05-22 04:12:09,227 INFO [sqlalchemy.engine.base.
↪Engine:1100][MainThread] ()
2016-05-22 04:12:09,228 INFO [sqlalchemy.engine.base.
↪Engine:1097][MainThread] PRAGMA table_info("pages")
2016-05-22 04:12:09,228 INFO [sqlalchemy.engine.base.
↪Engine:1100][MainThread] ()
2016-05-22 04:12:09,229 INFO [sqlalchemy.engine.base.
↪Engine:1097][MainThread]
CREATE TABLE users (

```



```
        id INTEGER NOT NULL,
        name TEXT NOT NULL,
        role TEXT NOT NULL,
        password_hash TEXT,
        CONSTRAINT pk_users PRIMARY KEY (id),
        CONSTRAINT uq_users_name UNIQUE (name)
    )

2016-05-22 04:12:09,229 INFO [sqlalchemy.engine.base.
↳Engine:1100][MainThread] ()
2016-05-22 04:12:09,230 INFO [sqlalchemy.engine.base.
↳Engine:686][MainThread] COMMIT
2016-05-22 04:12:09,230 INFO [sqlalchemy.engine.base.
↳Engine:1097][MainThread]
CREATE TABLE pages (
    id INTEGER NOT NULL,
    name TEXT NOT NULL,
    data TEXT NOT NULL,
    creator_id INTEGER NOT NULL,
    CONSTRAINT pk_pages PRIMARY KEY (id),
    CONSTRAINT uq_pages_name UNIQUE (name),
    CONSTRAINT fk_pages_creator_id_users FOREIGN KEY(creator_id)
↳REFERENCES users (id)
)

2016-05-22 04:12:09,231 INFO [sqlalchemy.engine.base.
↳Engine:1100][MainThread] ()
2016-05-22 04:12:09,231 INFO [sqlalchemy.engine.base.
↳Engine:686][MainThread] COMMIT
2016-05-22 04:12:09,782 INFO [sqlalchemy.engine.base.
↳Engine:646][MainThread] BEGIN (implicit)
2016-05-22 04:12:09,783 INFO [sqlalchemy.engine.base.
↳Engine:1097][MainThread] INSERT INTO users (name, role, password_hash)
↳VALUES (?, ?, ?)
2016-05-22 04:12:09,784 INFO [sqlalchemy.engine.base.
↳Engine:1100][MainThread] ('editor', 'editor', b'$2b$12$K/
↳WLVKRL5fMAb6UM58ueTetXlE3rlc5cRK5zFPimK598scXBR/xWC')
2016-05-22 04:12:09,784 INFO [sqlalchemy.engine.base.
↳Engine:1097][MainThread] INSERT INTO users (name, role, password_hash)
↳VALUES (?, ?, ?)
2016-05-22 04:12:09,784 INFO [sqlalchemy.engine.base.
↳Engine:1100][MainThread] ('basic', 'basic', b'$2b$12$JfwLyCJGv3t.
↳RTSmIrh3B.FKXRT9FevkAqafWdK5oq7Hl4mgAQORe')
2016-05-22 04:12:09,785 INFO [sqlalchemy.engine.base.
↳Engine:1097][MainThread] INSERT INTO pages (name, data, creator_id)
↳VALUES (?, ?, ?)
```

```

2016-05-22 04:12:09,785 INFO [sqlalchemy.engine.base.
→Engine:1100][MainThread] ('FrontPage', 'This is the front page', 1)
2016-05-22 04:12:09,786 INFO [sqlalchemy.engine.base.
→Engine:686][MainThread] COMMIT

```

View the application in a browser

We can't. At this point, our system is in a “non-runnable” state; we'll need to change view-related files in the next chapter to be able to start the application successfully. If you try to start the application (see *Start the application*), you'll wind up with a Python traceback on your console that ends with this exception:

```

ImportError: cannot import name MyModel

```

This will also happen if you attempt to run the tests.

Defining Views

A *view callable* in a Pyramid application is typically a simple Python function that accepts a single parameter named *request*. A view callable is assumed to return a *response* object.

The request object has a dictionary as an attribute named *matchdict*. A *matchdict* maps the placeholders in the matching URL pattern to the substrings of the path in the *request* URL. For instance, if a call to `pyramid.config.Configurator.add_route()` has the pattern `{one}/{two}`, and a user visits `http://example.com/foo/bar`, our pattern would be matched against `/foo/bar` and the *matchdict* would look like `{ 'one': 'foo', 'two': 'bar' }`.

Adding the `docutils` dependency

Remember in the previous chapter we added a new dependency of the `bcrypt` package. Again, the view code in our application will depend on a package which is not a dependency of the original “tutorial” application.

We need to add a dependency on the `docutils` package to our tutorial package's `setup.py` file by assigning this dependency to the `requires` parameter in the `setup()` function.

Open `tutorial/setup.py` and edit it to look like the following:

```
1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 with open(os.path.join(here, 'README.txt')) as f:
7     README = f.read()
8 with open(os.path.join(here, 'CHANGES.txt')) as f:
9     CHANGES = f.read()
10
11 requires = [
12     'bcrypt',
13     'docutils',
14     'pyramid',
15     'pyramid_jinja2',
16     'pyramid_debugtoolbar',
17     'pyramid_tm',
18     'SQLAlchemy',
19     'transaction',
20     'zope.sqlalchemy',
21     'waitress',
22 ]
23
24 tests_require = [
25     'WebTest >= 1.3.1', # py3 compat
26     'pytest', # includes virtualenv
27     'pytest-cov',
28 ]
29
30 setup(name='tutorial',
31       version='0.0',
32       description='tutorial',
33       long_description=README + '\n\n' + CHANGES,
34       classifiers=[
35         "Programming Language :: Python",
36         "Framework :: Pyramid",
37         "Topic :: Internet :: WWW/HTTP",
38         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
39     ],
40     author='',
41     author_email='',
42     url='',
43     keywords='web wsgi bfg pylons pyramid',
44     packages=find_packages(),
45     include_package_data=True,
46     zip_safe=False,
```

```

47     extras_require={
48         'testing': tests_require,
49     },
50     install_requires=requires,
51     entry_points="""\
52     [paste.app_factory]
53     main = tutorial:main
54     [console_scripts]
55     initialize_tutorial_db = tutorial.scripts.initializedb:main
56     """,
57 )

```

Only the highlighted line needs to be added.

Again, as we did in the previous chapter, the dependency now needs to be installed, so re-run the `$VENV/bin/pip install -e .` command.

Static assets

Our templates name static assets, including CSS and images. We don't need to create these files within our package's `static` directory because they were provided at the time we created the project.

As an example, the CSS file will be accessed via `http://localhost:6543/static/theme.css` by virtue of the call to the `add_static_view` directive we've made in the `routes.py` file. Any number and type of static assets can be placed in this directory (or subdirectories) and are just referred to by URL or by using the convenience method `static_url`, e.g., `request.static_url('<package>:static/foo.css')` within templates.

Adding routes to `routes.py`

This is the *URL Dispatch* tutorial, so let's start by adding some URL patterns to our app. Later we'll attach views to handle the URLs.

The `routes.py` file contains `pyramid.config.Configurator.add_route()` calls which serve to add routes to our application. First we'll get rid of the existing route created by the template using the name `'home'`. It's only an example and isn't relevant to our application.

We then need to add four calls to `add_route`. Note that the *ordering* of these declarations is very important. Route declarations are matched in the order they're registered.

1. Add a declaration which maps the pattern `/` (signifying the root URL) to the route named `view_wiki`. In the next step, we will map it to our `view_wiki` view callable by virtue of the `@view_config` decorator attached to the `view_wiki` view function, which in turn will be indicated by `route_name='view_wiki'`.
2. Add a declaration which maps the pattern `/ {pagename}` to the route named `view_page`. This is the regular view for a page. Again, in the next step, we will map it to our `view_page` view callable by virtue of the `@view_config` decorator attached to the `view_page` view function, which in turn will be indicated by `route_name='view_page'`.
3. Add a declaration which maps the pattern `/add_page/ {pagename}` to the route named `add_page`. This is the add view for a new page. We will map it to our `add_page` view callable by virtue of the `@view_config` decorator attached to the `add_page` view function, which in turn will be indicated by `route_name='add_page'`.
4. Add a declaration which maps the pattern `/ {pagename}/edit_page` to the route named `edit_page`. This is the edit view for a page. We will map it to our `edit_page` view callable by virtue of the `@view_config` decorator attached to the `edit_page` view function, which in turn will be indicated by `route_name='edit_page'`.

As a result of our edits, the `routes.py` file should look like the following:

```
1 def includeme(config):
2     config.add_static_view('static', 'static', cache_max_age=3600)
3     config.add_route('view_wiki', '/')
4     config.add_route('view_page', '/ {pagename}')
5     config.add_route('add_page', '/add_page/ {pagename}')
6     config.add_route('edit_page', '/ {pagename}/edit_page')
```

The highlighted lines are the ones that need to be added or edited.



The order of the routes is important! If you placed `/ {pagename}/edit_page` *before* `/add_page/ {pagename}`, then we would never be able to add pages. This is because the first route would always match a request to `/add_page/edit_page` whereas we want `/add_page/..` to have priority. This isn't a huge problem in this particular app because wiki pages are always camel case, but it's important to be aware of this behavior in your own apps.

Adding view functions in `views/default.py`

It's time for a major change. Open `tutorial/views/default.py` and edit it to look like the following:

```

1 from pyramid.compat import escape
2 import re
3 from docutils.core import publish_parts
4
5 from pyramid.httpexceptions import (
6     HTTPFound,
7     HTTPNotFound,
8 )
9
10 from pyramid.view import view_config
11
12 from ..models import Page, User
13
14 # regular expression used to find WikiWords
15 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+) ")
16
17 @view_config(route_name='view_wiki')
18 def view_wiki(request):
19     next_url = request.route_url('view_page', pagename='FrontPage')
20     return HTTPFound(location=next_url)
21
22 @view_config(route_name='view_page', renderer='../templates/view.jinja2')
23 def view_page(request):
24     pagename = request.matchdict['pagename']
25     page = request.dbsession.query(Page).filter_by(name=pagename).first()
26     if page is None:
27         raise HTTPNotFound('No such page')
28
29     def add_link(match):
30         word = match.group(1)
31         exists = request.dbsession.query(Page).filter_by(name=word).all()
32         if exists:
33             view_url = request.route_url('view_page', pagename=word)
34             return '<a href="%s">%s</a>' % (view_url, escape(word))
35         else:
36             add_url = request.route_url('add_page', pagename=word)
37             return '<a href="%s">%s</a>' % (add_url, escape(word))
38
39     content = publish_parts(page.data, writer_name='html')['html_body']
40     content = wikiwords.sub(add_link, content)
41     edit_url = request.route_url('edit_page', pagename=page.name)
42     return dict(page=page, content=content, edit_url=edit_url)
43
44 @view_config(route_name='edit_page', renderer='../templates/edit.jinja2')
45 def edit_page(request):
46     pagename = request.matchdict['pagename']

```

```
47     page = request.dbsession.query(Page).filter_by(name=pagename).one()
48     if 'form.submitted' in request.params:
49         page.data = request.params['body']
50         next_url = request.route_url('view_page', pagename=page.name)
51         return HTTPFound(location=next_url)
52     return dict(
53         pagename=page.name,
54         pagedata=page.data,
55         save_url=request.route_url('edit_page', pagename=page.name),
56     )
57
58 @view_config(route_name='add_page', renderer='../templates/edit.jinja2')
59 def add_page(request):
60     pagename = request.matchdict['pagename']
61     if request.dbsession.query(Page).filter_by(name=pagename).count() > 0:
62         next_url = request.route_url('edit_page', pagename=pagename)
63         return HTTPFound(location=next_url)
64     if 'form.submitted' in request.params:
65         body = request.params['body']
66         page = Page(name=pagename, data=body)
67         page.creator = (
68             request.dbsession.query(User).filter_by(name='editor').one()
69         )
70         request.dbsession.add(page)
71         next_url = request.route_url('view_page', pagename=pagename)
72         return HTTPFound(location=next_url)
73     save_url = request.route_url('add_page', pagename=pagename)
74     return dict(pagename=pagename, pagedata='', save_url=save_url)
```

The highlighted lines need to be added or edited.

We added some imports, and created a regular expression to find “WikiWords”.


We got rid of the `my_view` view function and its decorator that was added when we originally rendered the `alchemy` scaffold. It was only an example and isn’t relevant to our application. We also deleted the `db_err_msg` string.

Then we added four *view callable* functions to our `views/default.py` module, as mentioned in the previous step:

- `view_wiki()` - Displays the wiki itself. It will answer on the root URL.
- `view_page()` - Displays an individual page.
- `edit_page()` - Allows the user to edit a page.

- `add_page()` - Allows the user to add a page.

We'll describe each one briefly in the following sections.

 There is nothing special about the filename `default.py` except that it is a Python module. A project may have many view callables throughout its codebase in arbitrarily named modules. Modules implementing view callables often have `view` in their name (or may live in a Python subpackage of your application package named `views`, as in our case), but this is only by convention, not a requirement.

The `view_wiki` view function

Following is the code for the `view_wiki` view function and its decorator:

```
17 @view_config(route_name='view_wiki')
18 def view_wiki(request):
19     next_url = request.route_url('view_page', pagename='FrontPage')
20     return HTTPFound(location=next_url)
```

`view_wiki()` is the *default view* that gets called when a request is made to the root URL of our wiki. It always redirects to a URL which represents the path to our “FrontPage”.

The `view_wiki` view callable always redirects to the URL of a Page resource named “FrontPage”. To do so, it returns an instance of the `pyramid.httpexceptions.HTTPFound` class (instances of which implement the `pyramid.interfaces.IResponse` interface, like `pyramid.response.Response`). It uses the `pyramid.request.Request.route_url()` API to construct a URL to the FrontPage page (i.e., `http://localhost:6543/FrontPage`), and uses it as the “location” of the `HTTPFound` response, forming an HTTP redirect.

The `view_page` view function

Here is the code for the `view_page` view function and its decorator:


```
22 @view_config(route_name='view_page', renderer='../templates/view.jinja2')
23 def view_page(request):
24     pagename = request.matchdict['pagename']
25     page = request.dbsession.query(Page).filter_by(name=pagename).first()
26     if page is None:
27         raise HTTPNotFound('No such page')
28
29     def add_link(match):
30         word = match.group(1)
31         exists = request.dbsession.query(Page).filter_by(name=word).all()
32         if exists:
33             view_url = request.route_url('view_page', pagename=word)
34             return '<a href="%s">%s</a>' % (view_url, escape(word))
35         else:
36             add_url = request.route_url('add_page', pagename=word)
37             return '<a href="%s">%s</a>' % (add_url, escape(word))
38
39     content = publish_parts(page.data, writer_name='html')['html_body']
40     content = wikiwords.sub(add_link, content)
41     edit_url = request.route_url('edit_page', pagename=page.name)
42     return dict(page=page, content=content, edit_url=edit_url)
```

`view_page()` is used to display a single page of our wiki. It renders the *reStructuredText* body of a page (stored as the `data` attribute of a `Page` model object) as HTML. Then it substitutes an HTML anchor for each *WikiWord* reference in the rendered HTML using a compiled regular expression.

The curried function named `add_link` is used as the first argument to `wikiwords.sub`, indicating that it should be called to provide a value for each *WikiWord* match found in the content. If the wiki already contains a page with the matched *WikiWord* name, `add_link()` generates a view link to be used as the substitution value and returns it. If the wiki does not already contain a page with the matched *WikiWord* name, `add_link()` generates an “add” link as the substitution value and returns it.

As a result, the `content` variable is now a fully formed bit of HTML containing various view and add links for *WikiWords* based on the content of our current page object.

We then generate an edit URL, because it’s easier to do here than in the template, and we return a dictionary with a number of arguments. The fact that `view_page()` returns a dictionary (as opposed to a *response* object) is a cue to Pyramid that it should try to use a *renderer* associated with the view configuration to render a response. In our case, the renderer used will be the `view.jinja2` template, as indicated in the `@view_config` decorator that is applied to `view_page()`.

If the page does not exist, then we need to handle that by raising a `pyramid.httpexceptions.HTTPNotFound` to trigger our 404 handling, defined in `tutorial/views/notfound.py`.

i Using `raise` versus `return` with the HTTP exceptions is an important distinction that can commonly mess people up. In `tutorial/views/notfound.py` there is an *exception view* registered for handling the `HTTPNotFound` exception. Exception views are only triggered for raised exceptions. If the `HTTPNotFound` is returned, then it has an internal “stock” template that it will use to render itself as a response. If you aren’t seeing your exception view being executed, this is most likely the problem! See *Using Special Exceptions in View Callables* for more information about exception views.

The `edit_page` view function

Here is the code for the `edit_page` view function and its decorator:

```

44 @view_config(route_name='edit_page', renderer='../templates/edit.jinja2')
45 def edit_page(request):
46     pagename = request.matchdict['pagename']
47     page = request.dbsession.query(Page).filter_by(name=pagename).one()
48     if 'form.submitted' in request.params:
49         page.data = request.params['body']
50         next_url = request.route_url('view_page', pagename=page.name)
51         return HTTPFound(location=next_url)
52     return dict(
53         pagename=page.name,
54         pagedata=page.data,
55         save_url=request.route_url('edit_page', pagename=page.name),
56     )

```

`edit_page()` is invoked when a user clicks the “Edit this Page” button on the view form. It renders an edit form, but it also acts as the handler for the form which it renders. The `matchdict` attribute of the request passed to the `edit_page` view will have a `'pagename'` key matching the name of the page that the user wants to edit.

If the view execution *is* a result of a form submission (i.e., the expression `'form.submitted'` in `request.params` is `True`), the view grabs the `body` element of the request parameters and sets it as the `data` attribute of the page object. It then redirects to the `view_page` view of the wiki page.

If the view execution is *not* a result of a form submission (i.e., the expression `'form.submitted'` in `request.params` is `False`), the view simply renders the edit form, passing the page object and a `save_url` which will be used as the action of the generated form.

i Since our `request.dbsession` defined in the previous chapter is registered with the `pyramid_tm` transaction manager, any changes we make to objects managed by the that session will

be committed automatically. In the event that there was an error (even later, in our template code), the changes would be aborted. This means the view itself does not need to concern itself with commit/rollback logic.

The add_page view function

Here is the code for the add_page view function and its decorator:

```
58 @view_config(route_name='add_page', renderer='../templates/edit.jinja2')
59 def add_page(request):
60     pagename = request.matchdict['pagename']
61     if request.dbsession.query(Page).filter_by(name=pagename).count() > 0:
62         next_url = request.route_url('edit_page', pagename=pagename)
63         return HTTPFound(location=next_url)
64     if 'form.submitted' in request.params:
65         body = request.params['body']
66         page = Page(name=pagename, data=body)
67         page.creator = (
68             request.dbsession.query(User).filter_by(name='editor').one())
69         request.dbsession.add(page)
70         next_url = request.route_url('view_page', pagename=pagename)
71         return HTTPFound(location=next_url)
72     save_url = request.route_url('add_page', pagename=pagename)
73     return dict(pagename=pagename, pagedata='', save_url=save_url)
```

add_page() is invoked when a user clicks on a *WikiWord* which isn't yet represented as a page in the system. The add_link function within the view_page view generates URLs to this view. add_page() also acts as a handler for the form that is generated when we want to add a page object. The matchdict attribute of the request passed to the add_page() view will have the values we need to construct URLs and find model objects.

The matchdict will have a 'pagename' key that matches the name of the page we'd like to add. If our add view is invoked via, for example, `http://localhost:6543/add_page/SomeName`, the value for 'pagename' in the matchdict will be 'SomeName'.

Next a check is performed to determine whether the Page already exists in the database. If it already exists, then the client is redirected to the edit_page view, else we continue to the next check.

If the view execution is a result of a form submission (i.e., the expression 'form.submitted' in request.params is True), we grab the page body from the form data, create a Page object with this page body and the name taken from matchdict['pagename'], and save it into the database using

`request.dbession.add`. Since we have not yet covered authentication, we don't have a logged-in user to add as the page's creator. Until we get to that point in the tutorial, we'll just assume that all pages are created by the editor user. Thus we query for that object, and set it on `page.creator`. Finally, we redirect the client back to the `view_page` view for the newly created page.

If the view execution is *not* a result of a form submission (i.e., the expression `'form.submitted'` in `request.params` is `False`), the view callable renders a template. To do so, it generates a `save_url` which the template uses as the form post URL during rendering. We're lazy here, so we're going to use the same template (`templates/edit.jinja2`) for the add view as well as the page edit view. To do so we create a dummy `Page` object in order to satisfy the edit form's desire to have *some* page object exposed as `page`. Pyramid will render the template associated with this view to a response.

Adding templates

The `view_page`, `add_page` and `edit_page` views that we've added reference a *template*. Each template is a *Jinja2* template. These templates will live in the `templates` directory of our tutorial package. Jinja2 templates must have a `.jinja2` extension to be recognized as such.

The `layout.jinja2` template

Update `tutorial/templates/layout.jinja2` with the following content, as indicated by the emphasized lines:

```

1  <!DOCTYPE html>
2  <html lang="{{request.locale_name}}">
3    <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta name="description" content="pyramid web application">
8      <meta name="author" content="Pylons Project">
9      <link rel="shortcut icon" href="{{request.static_url('tutorial:static/
    ↳pyramid-16x16.png')}}">
10
11     <title>{% block subtitle %}{% endblock %}Pyramid tutorial wiki (based_
    ↳on TurboGears 20-Minute Wiki)</title>
12
13     <!-- Bootstrap core CSS -->
14     <link href="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/css/
    ↳bootstrap.min.css" rel="stylesheet">
15

```

```
16     <!-- Custom styles for this scaffold -->
17     <link href="{{request.static_url('tutorial:static/theme.css')}}" rel=
↪ "stylesheet">
18
19     <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media
↪ queries -->
20     <!--[if lt IE 9]>
21         <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></
↪ script>
22         <script src="//oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js">
↪ </script>
23         <![endif]-->
24     </head>
25
26     <body>
27
28         <div class="starter-template">
29             <div class="container">
30                 <div class="row">
31                     <div class="col-md-2">
32                         
33                     </div>
34                     <div class="col-md-10">
35                         <div class="content">
36                             {% block content %}{% endblock %}
37                         </div>
38                     </div>
39                 </div>
40                 <div class="row">
41                     <div class="copyright">
42                         Copyright &copy; Pylons Project
43                     </div>
44                 </div>
45             </div>
46         </div>
47
48
49         <!-- Bootstrap core JavaScript
50         ===== -->
51         <!-- Placed at the end of the document so the pages load faster -->
52         <script src="//oss.maxcdn.com/libs/jquery/1.10.2/jquery.min.js"></
↪ script>
53         <script src="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/js/
↪ bootstrap.min.js"></script>
54     </body>
```

```
55 </html>
```

Since we're using a templating engine, we can factor common boilerplate out of our page templates into reusable components. One method for doing this is template inheritance via blocks.

- We have defined two placeholders in the layout template where a child template can override the content. These blocks are named `subtitle` (line 11) and `content` (line 36).
- Please refer to the Jinja2 documentation for more information about template inheritance.

The `view.jinja2` template

Create `tutorial/templates/view.jinja2` and add the following content:

```
1 {% extends 'layout.jinja2' %}
2
3 {% block subtitle %}{{page.name}} - {% endblock subtitle %}
4
5 {% block content %}
6 <p>{{ content|safe }}</p>
7 <p>
8 <a href="{{ edit_url }}">
9     Edit this page
10 </a>
11 </p>
12 <p>
13     Viewing <strong>{{page.name}}</strong>, created by <strong>{{page.
14     ↪ creator.name}}</strong>.
15 </p>
16 <p>You can return to the
17 <a href="{{request.route_url('view_page', pagename='FrontPage')}}">
18     ↪ FrontPage</a>.
19 </p>
20 {% endblock content %}
```

This template is used by `view_page()` for displaying a single wiki page.

- We begin by extending the `layout.jinja2` template defined above, which provides the skeleton of the page (line 1).
- We override the `subtitle` block from the base layout, inserting the page name into the page's title (line 3).

- We override the `content` block from the base layout to insert our markup into the body (lines 5-18).
- We use a variable that is replaced with the `content` value provided by the view (line 6). `content` contains HTML, so the `|safe` filter is used to prevent escaping it (e.g., changing “>” to “>”).
- We create a link that points at the “edit” URL, which when clicked invokes the `edit_page` view for the requested page (line 9).

The `edit.jinja2` template

Create `tutorial/templates/edit.jinja2` and add the following content:

```
1 {% extends 'layout.jinja2' %}
2
3 {% block subtitle %}Edit {{pagename}} - {% endblock subtitle %}
4
5 {% block content %}
6 <p>
7 Editing <strong>{{pagename}}</strong>
8 </p>
9 <p>You can return to the
10 <a href="{{request.route_url('view_page', pagename='FrontPage')}}">
   ↪FrontPage</a>.
11 </p>
12 <form action="{{ save_url }}" method="post">
13 <div class="form-group">
14     <textarea class="form-control" name="body" rows="10" cols="60">{{
   ↪pagedata }}</textarea>
15 </div>
16 <div class="form-group">
17     <button type="submit" name="form.submitted" value="Save" class="btn
   ↪btn-default">Save</button>
18 </div>
19 </form>
20 {% endblock content %}
```

This template serves two use cases. It is used by `add_page()` and `edit_page()` for adding and editing a wiki page. It displays a page containing a form and which provides the following:

- Again, we extend the `layout.jinja2` template, which provides the skeleton of the page (line 1).

- Override the `subtitle` block to affect the `<title>` tag in the head of the page (line 3).
- A 10-row by 60-column `textarea` field named `body` that is filled with any existing page data when it is rendered (line 14).
- A submit button that has the name `form.submitted` (line 17).
- The form POSTs back to the `save_url` argument supplied by the view (line 12). The view will use the `body` and `form.submitted` values.

The 404.jinja2 template

Replace `tutorial/templates/404.jinja2` with the following content:

```

1 {% extends "layout.jinja2" %}
2
3 {% block content %}
4 <div class="content">
5   <h1><span class="font-semi-bold">Pyramid tutorial wiki</span> <span
6   ↪class="smaller">(based on TurboGears 20-Minute Wiki)</span></h1>
7   <p class="lead"><span class="font-semi-bold">404</span> Page Not Found</
8   ↪p>
9 </div>
10 {% endblock content %}

```

This template is linked from the `notfound_view` defined in `tutorial/views/notfound.py` as shown here:

```

1 from pyramid.view import notfound_view_config
2
3
4 @notfound_view_config(renderer='../templates/404.jinja2')
5 def notfound_view(request):
6     request.response.status = 404
7     return {}

```

There are several important things to note about this configuration:

- The `notfound_view` in the above snippet is called an *exception view*. For more information see *Using Special Exceptions in View Callables*.

- The `notfound_view` sets the response status to 404. It's possible to affect the response object used by the renderer via *Varying Attributes of Rendered Responses*.
- The `notfound_view` is registered as an exception view and will be invoked **only** if `pyramid.httpexceptions.HTTPNotFound` is raised as an exception. This means it will not be invoked for any responses returned from a view normally. For example, on line 27 of `tutorial/views/default.py` the exception is raised which will trigger the view.

Finally, we may delete the `tutorial/templates/mytemplate.jinja2` template that was provided by the `alchemy` scaffold, as we have created our own templates for the wiki.



Our templates use a `request` object that none of our tutorial views return in their dictionary. `request` is one of several names that are available “by default” in a template when a template renderer is used. See *System Values Used During Rendering* for information about other names that are available by default when a template is used as a renderer.

Viewing the application in a browser

We can finally examine our application in a browser (See *Start the application*). Launch a browser and visit each of the following URLs, checking that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` page object.
- `http://localhost:6543/FrontPage` invokes the `view_page` view of the `FrontPage` page object.
- `http://localhost:6543/FrontPage/edit_page` invokes the `edit_page` view for the `FrontPage` page object.
- `http://localhost:6543/add_page/SomePageName` invokes the `add_page` view for a page. If the page already exists, then it redirects the user to the `edit_page` view for the page object.
- `http://localhost:6543/SomePageName/edit_page` invokes the `edit_page` view for an existing page, or generates an error if the page does not exist.
- To generate an error, visit `http://localhost:6543/foobars/edit_page` which will generate a `NoResultFound: No row was found for one()` error. You'll see an interactive traceback facility provided by `pyramid_debugtoolbar`.

Adding authentication

Pyramid provides facilities for *authentication* and *authorization*. In this section we'll focus solely on the authentication APIs to add login and logout functionality to our wiki.

We will implement authentication with the following steps:

- Add an *authentication policy* and a `request.user` computed property (`security.py`).
- Add routes for `/login` and `/logout` (`routes.py`).
- Add login and logout views (`views/auth.py`).
- Add a login template (`login.jinja2`).
- Add “Login” and “Logout” links to every page based on the user's authenticated state (`layout.jinja2`).
- Make the existing views verify user state (`views/default.py`).
- Redirect to `/login` when a user is denied access to any of the views that require permission, instead of a default “403 Forbidden” page (`views/auth.py`).

Authenticating requests

The core of Pyramid authentication is an *authentication policy* which is used to identify authentication information from a `request`, as well as handling the low-level login and logout operations required to track users across requests (via cookies, headers, or whatever else you can imagine).

Add the authentication policy

Create a new file `tutorial/security.py` with the following content:

```
1 from pyramid.authentication import AuthTktAuthenticationPolicy
2 from pyramid.authorization import ACLAuthorizationPolicy
3
4 from .models import User
5
6
7 class MyAuthenticationPolicy(AuthTktAuthenticationPolicy):
8     def authenticated_userid(self, request):
9         user = request.user
10        if user is not None:
11            return user.id
12
13 def get_user(request):
14     user_id = request.unauthenticated_userid
15     if user_id is not None:
16         user = request.dbsession.query(User).get(user_id)
17         return user
18
19 def includeme(config):
20     settings = config.get_settings()
21     authn_policy = MyAuthenticationPolicy(
22         settings['auth.secret'],
23         hashalg='sha512',
24     )
25     config.set_authentication_policy(authn_policy)
26     config.set_authorization_policy(ACLAuthorizationPolicy())
27     config.add_request_method(get_user, 'user', reify=True)
```

Here we've defined:

- A new authentication policy named `MyAuthenticationPolicy`, which is subclassed from Pyramid's `pyramid.authentication.AuthTktAuthenticationPolicy`, which tracks the *userid* using a signed cookie (lines 7-11).
- A `get_user` function, which can convert the `unauthenticated_userid` from the policy into a `User` object from our database (lines 13-17).
- The `get_user` is registered on the request as `request.user` to be used throughout our application as the authenticated `User` object for the logged-in user (line 27).

The logic in this file is a little bit interesting, so we'll go into detail about what's happening here:

First, the default authentication policies all provide a method named `unauthenticated_userid` which is responsible for the low-level parsing of the information in the request (cookies, headers, etc.). If

a `userid` is found, then it is returned from this method. This is named `unauthenticated_userid` because, at the lowest level, it knows the value of the `userid` in the cookie, but it doesn't know if it's actually a user in our system (remember, anything the user sends to our app is untrusted).

Second, our application should only care about `authenticated_userid` and `request.user`, which have gone through our application-specific process of validating that the user is logged in.

In order to provide an `authenticated_userid` we need a verification step. That can happen anywhere, so we've elected to do it inside of the cached `request.user` computed property. This is a convenience that makes `request.user` the source of truth in our system. It is either `None` or a `User` object from our database. This is why the `get_user` function uses the `unauthenticated_userid` to check the database.

Configure the app

Since we've added a new `tutorial/security.py` module, we need to include it. Open the file `tutorial/__init__.py` and edit the following lines:

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6     """
7     config = Configurator(settings=settings)
8     config.include('pyramid_jinja2')
9     config.include('.models')
10    config.include('.routes')
11    config.include('.security')
12    config.scan()
13    return config.make_wsgi_app()
```

Our authentication policy is expecting a new setting, `auth.secret`. Open the file `development.ini` and add the highlighted line below:

```
18 sqlalchemy.url = sqlite:///%(here)s/tutorial.sqlite
19
20 auth.secret = seekrit
```

Finally, best practices tell us to use a different secret for production, so open `production.ini` and add a different secret:

```
15 sqlalchemy.url = sqlite:///%(here)s/tutorial.sqlite
16
17 auth.secret = real-seekrit
```

Add permission checks

Pyramid has full support for declarative authorization, which we'll cover in the next chapter. However, many people looking to get their feet wet are just interested in authentication with some basic form of home-grown authorization. We'll show below how to accomplish the simple security goals of our wiki, now that we can track the logged-in state of users.

Remember our goals:

- Allow only editor and basic logged-in users to create new pages.
- Only allow editor users and the page creator (possibly a basic user) to edit pages.

Open the file `tutorial/views/default.py` and fix the following imports:

```
5 from pyramid.httpexceptions import (
6     HTTPForbidden,
7     HTTPFound,
8     HTTPNotFound,
9 )
10
11 from pyramid.view import view_config
12
13 from ..models import Page
```

Change the two highlighted lines.

In the same file, now edit the `edit_page` view function:

```
45 @view_config(route_name='edit_page', renderer='../templates/edit.jinja2')
46 def edit_page(request):
47     pagename = request.matchdict['pagename']
48     page = request.dbsession.query(Page).filter_by(name=pagename).one()
49     user = request.user
50     if user is None or (user.role != 'editor' and page.creator != user):
51         raise HTTPForbidden
```

```

52     if 'form.submitted' in request.params:
53         page.data = request.params['body']
54         next_url = request.route_url('view_page', pagename=page.name)
55         return HTTPFound(location=next_url)
56     return dict(
57         pagename=page.name,
58         pagedata=page.data,
59         save_url=request.route_url('edit_page', pagename=page.name),
60     )

```

Only the highlighted lines need to be changed.

If the user either is not logged in or the user is not the page’s creator *and* not an editor, then we raise HTTPForbidden.

In the same file, now edit the `add_page` view function:

```

62 @view_config(route_name='add_page', renderer='../templates/edit.jinja2')
63 def add_page(request):
64     user = request.user
65     if user is None or user.role not in ('editor', 'basic'):
66         raise HTTPForbidden
67     pagename = request.matchdict['pagename']
68     if request.dbsession.query(Page).filter_by(name=pagename).count() > 0:
69         next_url = request.route_url('edit_page', pagename=pagename)
70         return HTTPFound(location=next_url)
71     if 'form.submitted' in request.params:
72         body = request.params['body']
73         page = Page(name=pagename, data=body)
74         page.creator = request.user
75         request.dbsession.add(page)
76         next_url = request.route_url('view_page', pagename=pagename)

```

Only the highlighted lines need to be changed.

If the user either is not logged in or is not in the `basic` or `editor` roles, then we raise `HTTPForbidden`, which will return a “403 Forbidden” response to the user. However, we will hook this later to redirect to the login page. Also, now that we have `request.user`, we no longer have to hard-code the creator as the `editor` user, so we can finally drop that hack.

These simple checks should protect our views.

Login, logout

Now that we've got the ability to detect logged-in users, we need to add the `/login` and `/logout` views so that they can actually login and logout!

Add routes for `/login` and `/logout`

Go back to `tutorial/routes.py` and add these two routes as highlighted:

```
3 config.add_route('view_wiki', '/')
4 config.add_route('login', '/login')
5 config.add_route('logout', '/logout')
6 config.add_route('view_page', '/{pagename}')
```



The preceding lines must be added *before* the following `view_page` route definition:

```
6 config.add_route('view_page', '/{pagename}')
```

This is because `view_page`'s route definition uses a catch-all “replacement marker” `{pagename}` (see *Route Pattern Syntax*), which will catch any route that was not already caught by any route registered before it. Hence, for `login` and `logout` views to have the opportunity of being matched (or “caught”), they must be above `{pagename}`.

Add login, logout, and forbidden views

Create a new file `tutorial/views/auth.py`, and add the following code to it:

```
1 from pyramid.httpexceptions import HTTPFound
2 from pyramid.security import (
3     remember,
4     forget,
5 )
6 from pyramid.view import (
7     forbidden_view_config,
8     view_config,
```

```

9 )
10
11 from ..models import User
12
13
14 @view_config(route_name='login', renderer='../templates/login.jinja2')
15 def login(request):
16     next_url = request.params.get('next', request.referrer)
17     if not next_url:
18         next_url = request.route_url('view_wiki')
19     message = ''
20     login = ''
21     if 'form.submitted' in request.params:
22         login = request.params['login']
23         password = request.params['password']
24         user = request.dbsession.query(User).filter_by(name=login).first()
25         if user is not None and user.check_password(password):
26             headers = remember(request, user.id)
27             return HTTPFound(location=next_url, headers=headers)
28         message = 'Failed login'
29
30     return dict(
31         message=message,
32         url=request.route_url('login'),
33         next_url=next_url,
34         login=login,
35     )
36
37 @view_config(route_name='logout')
38 def logout(request):
39     headers = forget(request)
40     next_url = request.route_url('view_wiki')
41     return HTTPFound(location=next_url, headers=headers)
42
43 @forbidden_view_config()
44 def forbidden_view(request):
45     next_url = request.route_url('login', _query={'next': request.url})
46     return HTTPFound(location=next_url)

```

This code adds three new views to the application:

- The login view renders a login form and processes the post from the login form, checking credentials against our `users` table in the database.

The check is done by first finding a `User` record in the database, then using our `user.check_password` method to compare the hashed passwords.

If the credentials are valid, then we use our authentication policy to store the user's id in the response using `pyramid.security.remember()`.

Finally, the user is redirected back to either the page which they were trying to access (`next`) or the front page as a fallback. This parameter is used by our forbidden view, as explained below, to finish the login workflow.

- The `logout` view handles requests to `/logout` by clearing the credentials using `pyramid.security.forget()`, then redirecting them to the front page.
- The `forbidden_view` is registered using the `pyramid.view.forbidden_view_config` decorator. This is a special *exception view*, which is invoked when a `pyramid.httpexceptions.HTTPForbidden` exception is raised.

This view will handle a forbidden error by redirecting the user to `/login`. As a convenience, it also sets the `next=` query string to the current URL (the one that is forbidding access). This way, if the user successfully logs in, they will be sent back to the page which they had been trying to access.

Add the `login.jinja2` template

Create `tutorial/templates/login.jinja2` with the following content:

```
{% extends 'layout.jinja2' %}

{% block title %}Login - {% endblock title %}

{% block content %}
<p>
<strong>
    Login
</strong><br>
{{ message }}
</p>
<form action="{{ url }}" method="post">
<input type="hidden" name="next" value="{{ next_url }}">
<div class="form-group">
    <label for="login">Username</label>
    <input type="text" name="login" value="{{ login }}">
</div>
<div class="form-group">
    <label for="password">Password</label>
    <input type="password" name="password">
</div>
</form>
</div>
```

```

</div>
<div class="form-group">
    <button type="submit" name="form.submitted" value="Log In" class="btn_
    ↪btn-default">Log In</button>
</div>
</form>
{% endblock content %}

```

The above template is referenced in the login view that we just added in `tutorial/views/auth.py`.

Add “Login” and “Logout” links

Open `tutorial/templates/layout.jinja2` and add the following code as indicated by the highlighted lines.

```

35         <div class="content">
36             {% if request.user is none %}
37             <p class="pull-right">
38                 <a href="{{ request.route_url('login') }}">Login</a>
39             </p>
40             {% else %}
41             <p class="pull-right">
42                 {{request.user.name}} <a href="{{request.route_url('logout')}}"
    ↪">Logout</a>
43             </p>
44             {% endif %}
45             {% block content %}{% endblock %}
46         </div>

```

The `request.user` will be `None` if the user is not authenticated, or a `tutorial.models.User` object if the user is authenticated. This check will make the logout link shown only when the user is logged in, and conversely the login link is only shown when the user is logged out.

Viewing the application in a browser

We can finally examine our application in a browser (See *Start the application*). Launch a browser and visit each of the following URLs, checking that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` page object. It is executable by any user.

- `http://localhost:6543/FrontPage` invokes the `view_page` view of the `FrontPage` page object. There is a “Login” link in the upper right corner while the user is not authenticated, else it is a “Logout” link when the user is authenticated.
- `http://localhost:6543/FrontPage/edit_page` invokes the `edit_page` view for the `FrontPage` page object. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, then a login form will be displayed. Supplying the credentials with the username `editor` and password `editor` will display the edit page form.
- `http://localhost:6543/add_page/SomePageName` invokes the `add_page` view for a page. If the page already exists, then it redirects the user to the `edit_page` view for the page object. It is executable by either the `editor` or `basic` user. If a different user (or the anonymous user) invokes it, then a login form will be displayed. Supplying the credentials with either the username `editor` and password `editor`, or username `basic` and password `basic`, will display the edit page form.
- `http://localhost:6543/SomePageName/edit_page` invokes the `edit_page` view for an existing page, or generates an error if the page does not exist. It is editable by the `basic` user if the page was created by that user in the previous step. If, instead, the page was created by the `editor` user, then the login page should be shown for the `basic` user.
- After logging in (as a result of hitting an edit or add page and submitting the login form with the `editor` credentials), we’ll see a “Logout” link in the upper right hand corner. When we click it, we’re logged out, redirected back to the front page, and a “Login” link is shown in the upper right hand corner.

Adding authorization

In the last chapter we built *authentication* into our wiki. We also went one step further and used the `request.user` object to perform some explicit *authorization* checks. This is fine for a lot of applications, but Pyramid provides some facilities for cleaning this up and decoupling the constraints from the view function itself.

We will implement access control with the following steps:

- Update the *authentication policy* to break down the *userid* into a list of *principals* (`security.py`).
- Define an *authorization policy* for mapping users, resources and permissions (`security.py`).
- Add new *resource* definitions that will be used as the *context* for the wiki pages (`routes.py`).
- Add an *ACL* to each resource (`routes.py`).
- Replace the inline checks on the views with *permission* declarations (`views/default.py`).

Add user principals

A *principal* is a level of abstraction on top of the raw *userid* that describes the user in terms of its capabilities, roles, or other identifiers that are easier to generalize. The permissions are then written against the principals without focusing on the exact user involved.

Pyramid defines two builtin principals used in every application: `pyramid.security.Everyone` and `pyramid.security.Authenticated`. On top of these we have already mentioned the required principals for this application in the original design. The user has two possible roles: `editor` or `basic`. These will be prefixed by the string `role:` to avoid clashing with any other types of principals.

Open the file `tutorial/security.py` and edit it as follows:

```

1  from pyramid.authentication import AuthTktAuthenticationPolicy
2  from pyramid.authorization import ACLAuthorizationPolicy
3  from pyramid.security import (
4      Authenticated,
5      Everyone,
6  )
7
8  from .models import User
9
10
11 class MyAuthenticationPolicy(AuthTktAuthenticationPolicy):
12     def authenticated_userid(self, request):
13         user = request.user
14         if user is not None:
15             return user.id
16
17     def effective_principals(self, request):
18         principals = [Everyone]
19         user = request.user
20         if user is not None:
21             principals.append(Authenticated)
22             principals.append(str(user.id))
23             principals.append('role:' + user.role)
24         return principals
25
26 def get_user(request):
27     user_id = request.unauthenticated_userid
28     if user_id is not None:
29         user = request.dbsession.query(User).get(user_id)
30         return user
31
32 def includeme(config):

```

```
33     settings = config.get_settings()
34     authn_policy = MyAuthenticationPolicy(
35         settings['auth.secret'],
36         hashalg='sha512',
37     )
38     config.set_authentication_policy(authn_policy)
39     config.set_authorization_policy(ACLAuthorizationPolicy())
40     config.add_request_method(get_user, 'user', reify=True)
```

Only the highlighted lines need to be added.

Note that the role comes from the `User` object. We also add the `user.id` as a principal for when we want to allow that exact user to edit pages which they have created.

Add the authorization policy

We already added the *authorization policy* in the previous chapter because Pyramid requires one when adding an *authentication policy*. However, it was not used anywhere, so we'll mention it now.

In the file `tutorial/security.py`, notice the following lines:

```
38     config.set_authentication_policy(authn_policy)
39     config.set_authorization_policy(ACLAuthorizationPolicy())
40     config.add_request_method(get_user, 'user', reify=True)
```

We're using the `pyramid.authorization.ACLAuthorizationPolicy`, which will suffice for most applications. It uses the *context* to define the mapping between a *principal* and *permission* for the current request via the `__acl__`.


Add resources and ACLs

Resources are the hidden gem of Pyramid. You've made it!

Every URL in a web application represents a *resource* (the “R” in Uniform Resource Locator). Often the resource is something in your data model, but it could also be an abstraction over many models.

Our wiki has two resources:

1. A `NewPage`. Represents a potential `Page` that does not exist. Any logged-in user, having either role of `basic` or `editor`, can create pages.
2. A `PageResource`. Represents a `Page` that is to be viewed or edited. `editor` users, as well as the original creator of the `Page`, may edit the `PageResource`. Anyone may view it.

 The wiki data model is simple enough that the `PageResource` is mostly redundant with our `models.Page` SQLAlchemy class. It is completely valid to combine these into one class. However, for this tutorial, they are explicitly separated to make clear the distinction between the parts about which Pyramid cares versus application-defined objects.

There are many ways to define these resources, and they can even be grouped into collections with a hierarchy. However, we're keeping it simple here!

Open the file `tutorial/routes.py` and edit the following lines:

```

1  from pyramid.httpexceptions import (
2      HTTPNotFound,
3      HTTPFound,
4  )
5  from pyramid.security import (
6      Allow,
7      Everyone,
8  )
9
10 from .models import Page
11
12 def includeme(config):
13     config.add_static_view('static', 'static', cache_max_age=3600)
14     config.add_route('view_wiki', '/')
15     config.add_route('login', '/login')
16     config.add_route('logout', '/logout')
17     config.add_route('view_page', '/{pagename}', factory=page_factory)
18     config.add_route('add_page', '/add_page/{pagename}',
19                     factory=new_page_factory)
20     config.add_route('edit_page', '/{pagename}/edit_page',
21                     factory=page_factory)
22
23 def new_page_factory(request):
24     pagename = request.matchdict['pagename']
25     if request.dbsession.query(Page).filter_by(name=pagename).count() > 0:
26         next_url = request.route_url('edit_page', pagename=pagename)
27         raise HTTPFound(location=next_url)

```

```
28     return NewPage(pagename)
29
30 class NewPage(object):
31     def __init__(self, pagename):
32         self.pagename = pagename
33
34     def __acl__(self):
35         return [
36             (Allow, 'role:editor', 'create'),
37             (Allow, 'role:basic', 'create'),
38         ]
39
40 def page_factory(request):
41     pagename = request.matchdict['pagename']
42     page = request.dbsession.query(Page).filter_by(name=pagename).first()
43     if page is None:
44         raise HTTPNotFound
45     return PageResource(page)
46
47 class PageResource(object):
48     def __init__(self, page):
49         self.page = page
50
51     def __acl__(self):
52         return [
53             (Allow, Everyone, 'view'),
54             (Allow, 'role:editor', 'edit'),
55             (Allow, str(self.page.creator_id), 'edit'),
56         ]
```

The highlighted lines need to be edited or added.

The NewPage class has an `__acl__` on it that returns a list of mappings from *principal* to *permission*. This defines *who* can do *what* with that *resource*. In our case we want to allow only those users with the principals of either `role:editor` or `role:basic` to have the `create` permission:

```
30 class NewPage(object):
31     def __init__(self, pagename):
32         self.pagename = pagename
33
34     def __acl__(self):
35         return [
36             (Allow, 'role:editor', 'create'),
37             (Allow, 'role:basic', 'create'),
38         ]
```

The `NewPage` is loaded as the *context* of the `add_page` route by declaring a factory on the route:

```
18 config.add_route('add_page', '/add_page/{pagename}',
19                 factory=new_page_factory)
```

The `PageResource` class defines the *ACL* for a `Page`. It uses an actual `Page` object to determine *who* can do *what* to the page.

```
47 class PageResource(object):
48     def __init__(self, page):
49         self.page = page
50
51     def __acl__(self):
52         return [
53             (Allow, Everyone, 'view'),
54             (Allow, 'role:editor', 'edit'),
55             (Allow, str(self.page.creator_id), 'edit'),
56         ]
```

The `PageResource` is loaded as the *context* of the `view_page` and `edit_page` routes by declaring a factory on the routes:

```
17 config.add_route('view_page', '/{pagename}', factory=page_factory)
18 config.add_route('add_page', '/add_page/{pagename}',
19                 factory=new_page_factory)
20 config.add_route('edit_page', '/{pagename}/edit_page',
21                 factory=page_factory)
```

Add view permissions

At this point we've modified our application to load the `PageResource`, including the actual `Page` model in the `page_factory`. The `PageResource` is now the *context* for all `view_page` and `edit_page` views. Similarly the `NewPage` will be the context for the `add_page` view.

Open the file `tutorial/views/default.py`.

First, you can drop a few imports that are no longer necessary:

```
5 from pyramid.httpexceptions import HTTPFound
6 from pyramid.view import view_config
7
```


Edit the `view_page` view to declare the `view` permission, and remove the explicit checks within the view:

```
18 @view_config(route_name='view_page', renderer='../templates/view.jinja2',
19              permission='view')
20 def view_page(request):
21     page = request.context.page
22
23     def add_link(match):
```

The work of loading the page has already been done in the factory, so we can just pull the `page` object out of the `PageResource`, loaded as `request.context`. Our factory also guarantees we will have a `Page`, as it raises the `HTTPNotFound` exception if no `Page` exists, again simplifying the view logic.

Edit the `edit_page` view to declare the `edit` permission:

```
38 @view_config(route_name='edit_page', renderer='../templates/edit.jinja2',
39              permission='edit')
40 def edit_page(request):
41     page = request.context.page
42     if 'form.submitted' in request.params:
```

Edit the `add_page` view to declare the `create` permission:

```
52 @view_config(route_name='add_page', renderer='../templates/edit.jinja2',
53              permission='create')
54 def add_page(request):
55     pagename = request.context.pagename
56     if 'form.submitted' in request.params:
```

Note the `pagename` here is pulled off of the context instead of `request.matchdict`. The factory has done a lot of work for us to hide the actual route pattern.

The ACLs defined on each *resource* are used by the *authorization policy* to determine if any *principal* is allowed to have some *permission*. If this check fails (for example, the user is not logged in) then an `HTTPForbidden` exception will be raised automatically. Thus we're able to drop those exceptions and checks from the views themselves. Rather we've defined them in terms of operations on a resource.

The final `tutorial/views/default.py` should look like the following:

```

1 from pyramid.compat import escape
2 import re
3 from docutils.core import publish_parts
4
5 from pyramid.httpexceptions import HTTPFound
6 from pyramid.view import view_config
7
8 from ..models import Page
9
10 # regular expression used to find WikiWords
11 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+)")
12
13 @view_config(route_name='view_wiki')
14 def view_wiki(request):
15     next_url = request.route_url('view_page', pagename='FrontPage')
16     return HTTPFound(location=next_url)
17
18 @view_config(route_name='view_page', renderer='../templates/view.jinja2',
19             permission='view')
20 def view_page(request):
21     page = request.context.page
22
23     def add_link(match):
24         word = match.group(1)
25         exists = request.dbsession.query(Page).filter_by(name=word).all()
26         if exists:
27             view_url = request.route_url('view_page', pagename=word)
28             return '<a href="%s">%s</a>' % (view_url, escape(word))
29         else:
30             add_url = request.route_url('add_page', pagename=word)
31             return '<a href="%s">%s</a>' % (add_url, escape(word))
32
33     content = publish_parts(page.data, writer_name='html')['html_body']
34     content = wikiwords.sub(add_link, content)
35     edit_url = request.route_url('edit_page', pagename=page.name)
36     return dict(page=page, content=content, edit_url=edit_url)
37
38 @view_config(route_name='edit_page', renderer='../templates/edit.jinja2',
39             permission='edit')
40 def edit_page(request):
41     page = request.context.page
42     if 'form.submitted' in request.params:
43         page.data = request.params['body']
44         next_url = request.route_url('view_page', pagename=page.name)
45         return HTTPFound(location=next_url)
46     return dict(

```

```
47         pagename=page.name,
48         pagedata=page.data,
49         save_url=request.route_url('edit_page', pagename=page.name),
50     )
51
52 @view_config(route_name='add_page', renderer='../templates/edit.jinja2',
53             permission='create')
54 def add_page(request):
55     pagename = request.context.pagename
56     if 'form.submitted' in request.params:
57         body = request.params['body']
58         page = Page(name=pagename, data=body)
59         page.creator = request.user
60         request.dbsession.add(page)
61         next_url = request.route_url('view_page', pagename=pagename)
62         return HTTPFound(location=next_url)
63     save_url = request.route_url('add_page', pagename=pagename)
64     return dict(pagename=pagename, pagedata='', save_url=save_url)
```

Viewing the application in a browser

We can finally examine our application in a browser (See *Start the application*). Launch a browser and visit each of the following URLs, checking that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` page object. It is executable by any user.
- `http://localhost:6543/FrontPage` invokes the `view_page` view of the `FrontPage` page object. There is a “Login” link in the upper right corner while the user is not authenticated, else it is a “Logout” link when the user is authenticated.
- `http://localhost:6543/FrontPage/edit_page` invokes the `edit_page` view for the `FrontPage` page object. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, then a login form will be displayed. Supplying the credentials with the username `editor` and password `editor` will display the edit page form.
- `http://localhost:6543/add_page/SomePageName` invokes the `add_page` view for a page. If the page already exists, then it redirects the user to the `edit_page` view for the page object. It is executable by either the `editor` or `basic` user. If a different user (or the anonymous user) invokes it, then a login form will be displayed. Supplying the credentials with either the username `editor` and password `editor`, or username `basic` and password `basic`, will display the edit page form.

- `http://localhost:6543/SomePageName/edit_page` invokes the `edit_page` view for an existing page, or generates an error if the page does not exist. It is editable by the `basic` user if the page was created by that user in the previous step. If, instead, the page was created by the `editor` user, then the login page should be shown for the `basic` user.
- After logging in (as a result of hitting an edit or add page and submitting the login form with the `editor` credentials), we'll see a "Logout" link in the upper right hand corner. When we click it, we're logged out, redirected back to the front page, and a "Login" link is shown in the upper right hand corner.

Adding Tests

We will now add tests for the models and views as well as a few functional tests in a new `tests` subpackage. Tests ensure that an application works, and that it continues to work when changes are made in the future.

The file `tests.py` was generated as part of the `alchemy` scaffold, but it is a common practice to put tests into a `tests` subpackage, especially as projects grow in size and complexity. Each module in the test subpackage should contain tests for its corresponding module in our application. Each corresponding pair of modules should have the same names, except the test module should have the prefix `test_`.

Start by deleting `tests.py`, then create a new directory to contain our new tests as well as a new empty file `tests/__init__.py`.



It is very important when refactoring a Python module into a package to be sure to delete the cache files (`.pyc` files or `__pycache__` folders) sitting around! Python will prioritize the cache files before traversing into folders, using the old code, and you will wonder why none of your changes are working!

Test the views

We'll create a new `tests/test_views.py` file, adding a `BaseTest` class used as the base for other test classes. Next we'll add tests for each view function we previously added to our application. We'll add four test classes: `ViewWikiTests`, `ViewPageTests`, `AddPageTests`, and `EditPageTests`. These test the `view_wiki`, `view_page`, `add_page`, and `edit_page` views.

Functional tests

We'll test the whole application, covering security aspects that are not tested in the unit tests, like logging in, logging out, checking that the `basic` user cannot edit pages that it didn't create but the `editor` user can, and so on.

View the results of all our edits to `tests` subpackage

Open `tutorial/tests/test_views.py`, and edit it such that it appears as follows:

```
1 import unittest
2 import transaction
3
4 from pyramid import testing
5
6
7 def dummy_request(dbsession):
8     return testing.DummyRequest(dbsession=dbsession)
9
10
11 class BaseTest(unittest.TestCase):
12     def setUp(self):
13         from ..models import get_tm_session
14         self.config = testing.setUp(settings={
15             'sqlalchemy.url': 'sqlite:///memory:'
16         })
17         self.config.include('..models')
18         self.config.include('..routes')
19
20         session_factory = self.config.registry['dbsession_factory']
21         self.session = get_tm_session(session_factory, transaction.manager)
22
23         self.init_database()
24
25     def init_database(self):
26         from ..models.meta import Base
27         session_factory = self.config.registry['dbsession_factory']
28         engine = session_factory.kw['bind']
29         Base.metadata.create_all(engine)
30
31     def tearDown(self):
32         testing.tearDown()
33         transaction.abort()
```

```

34
35     def makeUser(self, name, role, password='dummy'):
36         from ..models import User
37         user = User(name=name, role=role)
38         user.set_password(password)
39         return user
40
41     def makePage(self, name, data, creator):
42         from ..models import Page
43         return Page(name=name, data=data, creator=creator)
44
45
46 class ViewWikiTests(unittest.TestCase):
47     def setUp(self):
48         self.config = testing.setUp()
49         self.config.include('..routes')
50
51     def tearDown(self):
52         testing.tearDown()
53
54     def _callFUT(self, request):
55         from tutorial.views.default import view_wiki
56         return view_wiki(request)
57
58     def test_it(self):
59         request = testing.DummyRequest()
60         response = self._callFUT(request)
61         self.assertEqual(response.location, 'http://example.com/FrontPage')
62
63
64 class ViewPageTests(BaseTest):
65     def _callFUT(self, request):
66         from tutorial.views.default import view_page
67         return view_page(request)
68
69     def test_it(self):
70         from ..routes import PageResource
71
72         # add a page to the db
73         user = self.makeUser('foo', 'editor')
74         page = self.makePage('IDoExist', 'Hello CruelWorld IDoExist', user)
75         self.session.add_all([page, user])
76
77         # create a request asking for the page we've created
78         request = dummy_request(self.session)
79         request.context = PageResource(page)

```

```
80
81     # call the view we're testing and check its behavior
82     info = self._callFUT(request)
83     self.assertEqual(info['page'], page)
84     self.assertEqual(
85         info['content'],
86         '<div class="document">\n'
87         '<p>Hello <a href="http://example.com/add_page/CruelWorld">'
88         'CruelWorld</a> '
89         '<a href="http://example.com/IDoExist">'
90         'IDoExist</a>'
91         '</p>\n</div>\n')
92     self.assertEqual(info['edit_url'],
93                     'http://example.com/IDoExist/edit_page')
94
95
96 class AddPageTests(BaseTest):
97     def _callFUT(self, request):
98         from tutorial.views.default import add_page
99         return add_page(request)
100
101     def test_it_pageexists(self):
102         from ..models import Page
103         from ..routes import NewPage
104         request = testing.DummyRequest({'form.submitted': True,
105                                         'body': 'Hello yo!'},
106                                         dbsession=self.session)
107         request.user = self.makeUser('foo', 'editor')
108         request.context = NewPage('AnotherPage')
109         self._callFUT(request)
110         pagecount = self.session.query(Page).filter_by(name='AnotherPage').
↪count()
111         self.assertGreater(pagecount, 0)
112
113     def test_it_notsubmitted(self):
114         from ..routes import NewPage
115         request = dummy_request(self.session)
116         request.user = self.makeUser('foo', 'editor')
117         request.context = NewPage('AnotherPage')
118         info = self._callFUT(request)
119         self.assertEqual(info['pagedata'], '')
120         self.assertEqual(info['save_url'],
121                         'http://example.com/add_page/AnotherPage')
122
123     def test_it_submitted(self):
124         from ..models import Page
```

```

125     from ..routes import NewPage
126     request = testing.DummyRequest({'form.submitted': True,
127                                     'body': 'Hello yo!'},
128                                     dbsession=self.session)
129     request.user = self.makeUser('foo', 'editor')
130     request.context = NewPage('AnotherPage')
131     self._callFUT(request)
132     page = self.session.query(Page).filter_by(name='AnotherPage').one()
133     self.assertEqual(page.data, 'Hello yo!')
134
135
136 class EditPageTests(BaseTest):
137     def _callFUT(self, request):
138         from tutorial.views.default import edit_page
139         return edit_page(request)
140
141     def makeContext(self, page):
142         from ..routes import PageResource
143         return PageResource(page)
144
145     def test_it_notsubmitted(self):
146         user = self.makeUser('foo', 'editor')
147         page = self.makePage('abc', 'hello', user)
148         self.session.add_all([page, user])
149
150         request = dummy_request(self.session)
151         request.context = self.makeContext(page)
152         info = self._callFUT(request)
153         self.assertEqual(info['pagename'], 'abc')
154         self.assertEqual(info['save_url'],
155                          'http://example.com/abc/edit_page')
156
157     def test_it_submitted(self):
158         user = self.makeUser('foo', 'editor')
159         page = self.makePage('abc', 'hello', user)
160         self.session.add_all([page, user])
161
162         request = testing.DummyRequest({'form.submitted': True,
163                                         'body': 'Hello yo!'},
164                                         dbsession=self.session)
165         request.context = self.makeContext(page)
166         response = self._callFUT(request)
167         self.assertEqual(response.location, 'http://example.com/abc')
168         self.assertEqual(page.data, 'Hello yo!')

```

Open `tutorial/tests/test_functional.py`, and edit it such that it appears as follows:


```
1 import transaction
2 import unittest
3 import webtest
4
5
6 class FunctionalTests(unittest.TestCase):
7
8     basic_login = (
9         '/login?login=basic&password=basic'
10        '&next=FrontPage&form.submitted=Login')
11     basic_wrong_login = (
12         '/login?login=basic&password=incorrect'
13        '&next=FrontPage&form.submitted=Login')
14     editor_login = (
15         '/login?login=editor&password=editor'
16        '&next=FrontPage&form.submitted=Login')
17
18     @classmethod
19     def setUpClass(cls):
20         from tutorial.models.meta import Base
21         from tutorial.models import (
22             User,
23             Page,
24             get_tm_session,
25         )
26         from tutorial import main
27
28         settings = {
29             'sqlalchemy.url': 'sqlite://',
30             'auth.secret': 'seekrit',
31         }
32         app = main({}, **settings)
33         cls.testapp = webtest.TestApp(app)
34
35         session_factory = app.registry['dbsession_factory']
36         cls.engine = session_factory.kw['bind']
37         Base.metadata.create_all(bind=cls.engine)
38
39         with transaction.manager:
40             dbsession = get_tm_session(session_factory, transaction.
41 ↪manager)
42             editor = User(name='editor', role='editor')
43             editor.set_password('editor')
44             basic = User(name='basic', role='basic')
45             basic.set_password('basic')
46             page1 = Page(name='FrontPage', data='This is the front page')
```

```

46         page1.creator = editor
47         page2 = Page(name='BackPage', data='This is the back page')
48         page2.creator = basic
49         dbsession.add_all([basic, editor, page1, page2])
50
51     @classmethod
52     def tearDownClass(cls):
53         from tutorial.models.meta import Base
54         Base.metadata.drop_all(bind=cls.engine)
55
56     def test_root(self):
57         res = self.testapp.get('/', status=302)
58         self.assertEqual(res.location, 'http://localhost/FrontPage')
59
60     def test_FrontPage(self):
61         res = self.testapp.get('/FrontPage', status=200)
62         self.assertTrue(b'FrontPage' in res.body)
63
64     def test_unexisting_page(self):
65         self.testapp.get('/SomePage', status=404)
66
67     def test_successful_login(self):
68         res = self.testapp.get(self.basic_login, status=302)
69         self.assertEqual(res.location, 'http://localhost/FrontPage')
70
71     def test_failed_login(self):
72         res = self.testapp.get(self.basic_wrong_login, status=200)
73         self.assertTrue(b'login' in res.body)
74
75     def test_logout_link_present_when_logged_in(self):
76         self.testapp.get(self.basic_login, status=302)
77         res = self.testapp.get('/FrontPage', status=200)
78         self.assertTrue(b'Logout' in res.body)
79
80     def test_logout_link_not_present_after_logged_out(self):
81         self.testapp.get(self.basic_login, status=302)
82         self.testapp.get('/FrontPage', status=200)
83         res = self.testapp.get('/logout', status=302)
84         self.assertTrue(b'Logout' not in res.body)
85
86     def test_anonymous_user_cannot_edit(self):
87         res = self.testapp.get('/FrontPage/edit_page', status=302).follow()
88         self.assertTrue(b'Login' in res.body)
89
90     def test_anonymous_user_cannot_add(self):
91         res = self.testapp.get('/add_page/NewPage', status=302).follow()

```

```
92         self.assertTrue(b'Login' in res.body)
93
94     def test_basic_user_cannot_edit_front(self):
95         self.testapp.get(self.basic_login, status=302)
96         res = self.testapp.get('/FrontPage/edit_page', status=302).follow()
97         self.assertTrue(b'Login' in res.body)
98
99     def test_basic_user_can_edit_back(self):
100         self.testapp.get(self.basic_login, status=302)
101         res = self.testapp.get('/BackPage/edit_page', status=200)
102         self.assertTrue(b'Editing' in res.body)
103
104     def test_basic_user_can_add(self):
105         self.testapp.get(self.basic_login, status=302)
106         res = self.testapp.get('/add_page/NewPage', status=200)
107         self.assertTrue(b'Editing' in res.body)
108
109     def test_editors_member_user_can_edit(self):
110         self.testapp.get(self.editor_login, status=302)
111         res = self.testapp.get('/FrontPage/edit_page', status=200)
112         self.assertTrue(b'Editing' in res.body)
113
114     def test_editors_member_user_can_add(self):
115         self.testapp.get(self.editor_login, status=302)
116         res = self.testapp.get('/add_page/NewPage', status=200)
117         self.assertTrue(b'Editing' in res.body)
118
119     def test_editors_member_user_can_view(self):
120         self.testapp.get(self.editor_login, status=302)
121         res = self.testapp.get('/FrontPage', status=200)
122         self.assertTrue(b'FrontPage' in res.body)
```



We're utilizing the excellent WebTest package to do functional testing of the application. This is defined in the `tests_require` section of our `setup.py`. Any other dependencies needed only for testing purposes can be added there and will be installed automatically when running `setup.py test`.

Running the tests

We can run these tests similarly to how we did in *Run the tests*:

On UNIX:


```
$ $VENV/bin/py.test -q
```

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\py.test -q
```

The expected result should look like the following:

```
.....
22 passed, 1 pytest-warnings in 5.81 seconds
```

 If you use Python 3 during this tutorial, you will see deprecation warnings in the output, which we will choose to ignore. In making this tutorial run on both Python 2 and 3, the authors prioritized simplicity and focus for the learner over accommodating warnings. In your own app or as extra credit, you may choose to either drop Python 2 support or hack your code to work without warnings on both Python 2 and 3.

Distributing Your Application

Once your application works properly, you can create a “tarball” from it by using the `setup.py sdist` command. The following commands assume your current working directory contains the `tutorial` package and the `setup.py` file.

On UNIX:

```
$ $VENV/bin/python setup.py sdist
```

On Windows:

```
c:\pyramidtut> %VENV%\Scripts\python setup.py sdist
```

The output of such a command will be something like:

```
running sdist
# .. more output ..
creating dist
Creating tar archive
removing 'tutorial-0.0' (and everything under it)
```

Note that this command creates a tarball in the “dist” subdirectory named `tutorial-0.0.tar.gz`. You can send this file to your friends to show them your cool new application. They should be able to install it by pointing the `easy_install` command directly at it. Or you can upload it to PyPI and share it with the rest of the world, where it can be downloaded via `easy_install` remotely like any other package people download from PyPI.

ZODB + Traversal Wiki Tutorial

This tutorial introduces a *ZODB* and *traversal*-based Pyramid application to a developer familiar with Python. It will be most familiar to developers with previous *Zope* experience. When finished, the developer will have created a basic Wiki application with authentication.

For cut and paste purposes, the source code for all stages of this tutorial can be browsed on GitHub at `docs/tutorials/wiki/src`, which corresponds to the same location if you have Pyramid sources.

Background

This version of the Pyramid wiki tutorial presents a Pyramid application that uses technologies which will be familiar to someone with *Zope* experience. It uses *ZODB* as a persistence mechanism and *traversal* to map URLs to code. It can also be followed by people without any prior Python web framework experience.

To code along with this tutorial, the developer will need a UNIX machine with development tools (Mac OS X with XCode, any Linux or BSD variant, etc.) *or* a Windows system of any kind.



This tutorial has been written for Python 2. It is unlikely to work without modification under Python 3.

Have fun!

Design

Following is a quick overview of the design of our wiki application, to help us understand the changes that we will be making as we work through the tutorial.

Overall

We choose to use *reStructuredText* markup in the wiki text. Translation from `reStructuredText` to HTML is provided by the widely used `docutils` Python module. We will add this module in the dependency list on the project `setup.py` file.

Models

The root resource named `Wiki` will be a mapping of wiki page names to page resources. The page resources will be instances of a *Page* class and they store the text content.

URLs like `/PageName` will be traversed using `Wiki[PageName] => page`, and the context that results is the page resource of an existing page.

To add a page to the wiki, a new instance of the page resource is created and its name and reference are added to the `Wiki` mapping.

A page named `FrontPage` containing the text *This is the front page*, will be created when the storage is initialized, and will be used as the wiki home page.

Views

There will be three views to handle the normal operations of adding, editing, and viewing wiki pages, plus one view for the wiki front page. Two templates will be used, one for viewing, and one for both adding and editing wiki pages.

The default templating systems in Pyramid are *Chameleon* and *Mako*. *Chameleon* is a variant of *ZPT*, which is an XML-based templating language. *Mako* is a non-XML-based templating language. Because we had to pick one, we chose *Chameleon* for this tutorial.

Security

We'll eventually be adding security to our application. The components we'll use to do this are below.

- `USERS`, a dictionary mapping *userids* to their corresponding passwords.
- `GROUPS`, a dictionary mapping *userids* to a list of groups to which they belong.
- `groupfinder`, an *authorization callback* that looks up `USERS` and `GROUPS`. It will be provided in a new `security.py` file.
- An *ACL* is attached to the root *resource*. Each row below details an *ACE*:

Action	Principal	Permission
Allow	Everyone	View
Allow	group:editors	Edit

- Permission declarations are added to the views to assert the security policies as each request is handled.

Two additional views and one template will handle the login and logout tasks.

Summary

The URL, context, actions, template and permission associated to each view are listed in the following table:

URL	View	Context	Action	Template	Permission
/	view_wiki	Wiki	Redirect to /FrontPage		
/PageName	view_page ¹	Page	Display existing page ²	view.pt	view
/PageName/edit_page	edit_page	Page	Display edit form with existing content. If the form was submitted, redirect to /PageName	edit.pt	edit
/add_page/PageName	add_page	Wiki	Create the page <i>PageName</i> in storage, display the edit form without content. If the form was submitted, redirect to /PageName	edit.pt	edit
/login	login	Wiki, Forbidden ³	Display login form. If the form was submitted, authenticate. <ul style="list-style-type: none"> • If authentication succeeds, redirect to the page that we came from. • If authentication fails, display login form with “login 	login.pt	

Installation

Before you begin

This tutorial assumes that you have already followed the steps in *Installing Pyramid*, except **do not create a virtual environment or install Pyramid**. Thereby you will satisfy the following requirements.

- A Python interpreter is installed on your operating system.
- You've satisfied the *Requirements for Installing Packages*.

Create directory to contain the project

We need a workspace for our project files.

On UNIX

```
$ mkdir ~/pyramidtut
```

On Windows

```
c:\> mkdir pyramidtut
```

Create and use a virtual Python environment

Next let's create a virtual environment workspace for our project. We will use the `VENV` environment variable instead of the absolute path of the virtual environment.

On UNIX

² Pyramid will return a default 404 Not Found page if the page *PageName* does not exist yet.

³ `pyramid.exceptions.Forbidden` is reached when a user tries to invoke a view that is not authorized by the authorization policy.

```
$ export VENV=~/.pyramiddtut
$ python3 -m venv $VENV
```

On Windows

```
c:\> set VENV=c:\pyramiddtut
```

Each version of Python uses different paths, so you will need to adjust the path to the command for your Python version.

Python 2.7:

```
c:\> c:\Python27\Scripts\virtualenv %VENV%
```

Python 3.5:

```
c:\> c:\Python35\Scripts\python -m venv %VENV%
```

Upgrade pip and setuptools in the virtual environment

On UNIX

```
$ $VENV/bin/pip install --upgrade pip setuptools
```

On Windows

```
c:\> %VENV%\Scripts\pip install --upgrade pip setuptools
```

Install Pyramid into the virtual Python environment

On UNIX

```
$ $VENV/bin/pip install "pyramid==1.7.6"
```

On Windows

```
c:\> %VENV%\Scripts\pip install "pyramid==1.7.6"
```

Change directory to your virtual Python environment

Change directory to the `pyramidtut` directory, which is both your workspace and your virtual environment.

On UNIX

```
$ cd pyramidtut
```

On Windows

```
c:\> cd pyramidtut
```

Making a project

Your next step is to create a project. For this tutorial, we will use the *scaffold* named `zodb`, which generates an application that uses *ZODB* and *traversal*.

Pyramid supplies a variety of scaffolds to generate sample projects. We will use `pcreate`, a script that comes with Pyramid, to create our project using a scaffold.

By passing `zodb` into the `pcreate` command, the script creates the files needed to use ZODB. By passing in our application name `tutorial`, the script inserts that application name into all the required files.


The below instructions assume your current working directory is “pyramidtut”.

On UNIX

```
$ $VENV/bin/pcreate -s zodb tutorial
```

On Windows

```
c:\pyramidtut> %VENV%\Scripts\pcreate -s zodb tutorial
```

 If you are using Windows, the zodb scaffold may not deal gracefully with installation into a location that contains spaces in the path. If you experience startup problems, try putting both the virtual environment and the project into directories that do not contain spaces in their paths.

Installing the project in development mode

In order to do development on the project easily, you must “register” the project as a development egg in your workspace using the `pip install -e .` command. In order to do so, change directory to the tutorial directory that you created in *Making a project*, and run the `pip install -e .` command using the virtual environment Python interpreter.

On UNIX

```
$ cd tutorial
$ $VENV/bin/pip install -e .
```

On Windows

```
c:\pyramidtut> cd tutorial
c:\pyramidtut\tutorial> %VENV%\Scripts\pip install -e .
```

The console will show `pip` checking for packages and installing missing packages. Success executing this command will show a line like the following:

```
Successfully installed BTrees-4.2.0 Chameleon-2.24 Mako-1.0.4 \
MarkupSafe-0.23 Pygments-2.1.3 ZConfig-3.1.0 ZEO-4.2.0b1 ZODB-4.2.0 \
ZODB3-3.11.0 mock-2.0.0 pbr-1.8.1 persistent-4.1.1 pyramid-chameleon-0.3 \
pyramid-debugtoolbar-2.4.2 pyramid-mako-1.0.2 pyramid-tm-0.12.1 \
pyramid-zodbconn-0.7 six-1.10.0 transaction-1.4.4 tutorial waitress-0.8.10_
↪\
zc.lockfile-1.1.0 zdaemon-4.1.0 zodbpickle-0.6.0 zodburi-2.0
```

Install testing requirements

In order to run tests, we need to install the testing requirements. This is done through our project's `setup.py` file, in the `tests_require` and `extras_require` stanzas, and by issuing the command below for your operating system.

```
22 tests_require = [
23     'WebTest >= 1.3.1', # py3 compat
24     'pytest', # includes virtualenv
25     'pytest-cov',
26 ]
```

```
45     extras_require={
46         'testing': tests_require,
47     },
```

On UNIX

```
$ $VENV/bin/pip install -e ".[testing]"
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\pip install -e ".[testing]"
```

Run the tests

After you’ve installed the project in development mode as well as the testing requirements, you may run the tests for the project. The following commands provide options to `py.test` that specify the module for which its tests shall be run, and to run `py.test` in quiet mode.

On UNIX

```
$ $VENV/bin/py.test -q
```

On Windows

```
c:\pyramiddtut\tutorial> %VENV%\Scripts\py.test -q
```

For a successful test run, you should see output that ends like this:

```
.  
1 passed in 0.24 seconds
```

Expose test coverage information

You can run the `py.test` command to see test coverage information. This runs the tests in the same way that `py.test` does, but provides additional “coverage” information, exposing which lines of your project are covered by the tests.

We’ve already installed the `pytest-cov` package into our virtual environment, so we can run the tests with coverage.

On UNIX

```
$ $VENV/bin/py.test --cov --cov-report=term-missing
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\py.test --cov \
--cov-report=term-missing
```

If successful, you will see output something like this:

```
===== test session starts =====
platform Python 3.5.1, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: /Users/stevepiercy/projects/pyramidtut/tutorial, inifile:
plugins: cov-2.2.1
collected 1 items

tutorial/tests.py .

----- coverage: platform Python 3.5.1 -----
Name                               Stmts   Miss  Cover   Missing
-----
tutorial/__init__.py                12      7    42%    7-8, 14-18
tutorial/models.py                  10      6    40%    9-14
tutorial/tests.py                   12      0   100%
tutorial/views.py                   4      0   100%
-----
TOTAL                               38     13    66%

===== 1 passed in 0.31 seconds =====
```

Our package doesn't quite have 100% test coverage.

Test and coverage scaffold defaults

Scaffolds include configuration defaults for `py.test` and test coverage. These configuration files are `pytest.ini` and `.coveragerc`, located at the root of your package. Without these defaults, we would need to specify the path to the module on which we want to run tests and coverage.

On UNIX

```
$ %VENV%/bin/py.test --cov=tutorial tutorial/tests.py -q
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\py.test --cov=tutorial \
--cov-report=term-missing tutorial\tests.py -q
```

`py.test` follows conventions for Python test discovery, and the configuration defaults from the scaffold tell `py.test` where to find the module on which we want to run tests and coverage.

See also:

See `py.test`'s documentation for Usage and Invocations or invoke `py.test -h` to see its full set of options.

Start the application


Start the application.

On UNIX

```
$ %VENV/bin/pserve development.ini --reload
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\pserve development.ini --reload
```

 Your OS firewall, if any, may pop up a dialog asking for authorization to allow python to accept incoming network connections.

If successful, you will see something like this on your console:

```
Starting subprocess with file monitor
Starting server in PID 82349.
serving on http://127.0.0.1:6543
```

This means the server is ready to accept requests.

Visit the application in a browser


In a browser, visit `http://localhost:6543/`. You will see the generated application's default page.

One thing you'll notice is the “debug toolbar” icon on right hand side of the page. You can read more about the purpose of the icon at *The Debug Toolbar*. It allows you to get information about your application while you develop.

Decisions the `zodb` scaffold has made for you

Creating a project using the `zodb` scaffold makes the following assumptions:

- You are willing to use *ZODB* as persistent storage.
- You are willing to use *traversal* to map URLs to code.
- You want to use `pyramid_zodbconn`, `pyramid_tm`, and the transaction packages to manage connections and transactions with *ZODB*.
- You want to use `pyramid_chameleon` to render your templates. Different templating engines can be used, but we had to choose one to make this tutorial. See *Available Add-On Template System Bindings* for some options.

 Pyramid supports any persistent storage mechanism (e.g., an SQL database or filesystem files). It also supports an additional mechanism to map URLs to code (*URL dispatch*). However, for the purposes of this tutorial, we'll only be using *traversal* and *ZODB*.

Basic Layout

The starter files generated by the `zodb` scaffold are very basic, but they provide a good orientation for the high-level patterns common to most *traversal*-based (and *ZODB*-based) Pyramid projects.

Application configuration with `__init__.py`

A directory on disk can be turned into a Python *package* by containing an `__init__.py` file. Even if empty, this marks a directory as a Python package. We use `__init__.py` both as a marker, indicating the directory in which it's contained is a package, and to contain application configuration code.

When you run the application using the `pserve` command using the `development.ini` generated configuration file, the application configuration points at a setuptools *entry point* described as `egg:tutorial`. In our application, because the application's `setup.py` file says so, this entry point happens to be the main function within the file named `__init__.py`.

Open `tutorial/__init__.py`. It should already contain the following:

```

1  from pyramid.config import Configurator
2  from pyramid_zodbconn import get_connection
3  from .models import appmaker
4
5
6  def root_factory(request):
7      conn = get_connection(request)
8      return appmaker(conn.root())
9
10
11 def main(global_config, **settings):
12     """ This function returns a Pyramid WSGI application.
13     """
14     config = Configurator(root_factory=root_factory, settings=settings)
15     config.include('pyramid_chameleon')
16     config.add_static_view('static', 'static', cache_max_age=3600)
17     config.scan()
18     return config.make_wsgi_app()
```

1. Lines 1-3. Perform some dependency imports.
2. Lines 6-8. Define a *root factory* for our Pyramid application.
3. Line 11. `__init__.py` defines a function named `main`.
4. Line 14. We construct a *Configurator* with a root factory and the settings keywords parsed by *PasteDeploy*. The root factory is named `root_factory`.
5. Line 15. Include support for the *Chameleon* template rendering bindings, allowing us to use the `.pt` templates.

6. *Line 16.* Register a “static view”, which answers requests whose URL paths start with `/static`, using the `pyramid.config.Configurator.add_static_view()` method. This statement registers a view that will serve up static assets, such as CSS and image files, for us, in this case, at `http://localhost:6543/static/` and below. The first argument is the “name” `static`, which indicates that the URL path prefix of the view will be `/static`. The second argument of this tag is the “path”, which is a relative *asset specification*, so it finds the resources it should serve within the `static` directory inside the `tutorial` package. Alternatively the scaffold could have used an *absolute* asset specification as the path (`tutorial:static`).
7. *Line 17.* Perform a *scan*. A scan will find *configuration decoration*, such as view configuration decorators (e.g., `@view_config`) in the source code of the `tutorial` package and will take actions based on these decorators. We don’t pass any arguments to `scan()`, which implies that the scan should take place in the current package (in this case, `tutorial`). The scaffold could have equivalently said `config.scan('tutorial')`, but it chose to omit the package name argument.
8. *Line 18.* Use the `pyramid.config.Configurator.make_wsgi_app()` method to return a WSGI application.

Resources and models with `models.py`

Pyramid uses the word *resource* to describe objects arranged hierarchically in a *resource tree*. This tree is consulted by *traversal* to map URLs to code. In this application, the resource tree represents the site structure, but it *also* represents the *domain model* of the application, because each resource is a node stored persistently in a *ZODB* database. The `models.py` file is where the `zodb` scaffold put the classes that implement our resource objects, each of which also happens to be a domain model object.

Here is the source for `models.py`:

```
1 from persistent.mapping import PersistentMapping
2
3
4 class MyModel(PersistentMapping):
5     __parent__ = __name__ = None
6
7
8 def appmaker(zodb_root):
9     if 'app_root' not in zodb_root:
10         app_root = MyModel()
11         zodb_root['app_root'] = app_root
12         import transaction
13         transaction.commit()
14     return zodb_root['app_root']
```

1. *Lines 4-5.* The `MyModel resource` class is implemented here. Instances of this class are capable of being persisted in *ZODB* because the class inherits from the `persistent.mapping.PersistentMapping` class. The `__parent__` and `__name__` are important parts of the *traversal* protocol. By default, have these as `None` indicating that this is the *root* object.
2. *Lines 8-14.* `appmaker` is used to return the *application root* object. It is called on *every request* to the Pyramid application. It also performs bootstrapping by *creating* an application root (inside the *ZODB* root object) if one does not already exist. It is used by the `root_factory` we've defined in our `__init__.py`.

Bootstrapping is done by first seeing if the database has the persistent application root. If not, we make an instance, store it, and commit the transaction. We then return the application root object.

Views With `views.py`

Our scaffold generated a default `views.py` on our behalf. It contains a single view, which is used to render the page shown when you visit the URL `http://localhost:6543/`.

Here is the source for `views.py`:

```

1 from pyramid.view import view_config
2 from .models import MyModel
3
4
5 @view_config(context=MyModel, renderer='templates/mytemplate.pt')
6 def my_view(request):
7     return {'project': 'tutorial'}
```

Let's try to understand the components in this module:

1. *Lines 1-2.* Perform some dependency imports.
2. *Line 5.* Use the `pyramid.view.view_config()` *configuration decoration* to perform a *view configuration* registration. This view configuration registration will be activated when the application is started. It will be activated by virtue of it being found as the result of a *scan* (when Line 14 of `__init__.py` is run).

The `@view_config` decorator accepts a number of keyword arguments. We use two keyword arguments here: `context` and `renderer`.

The `context` argument signifies that the decorated view callable should only be run when *traversal* finds the `tutorial.models.MyModel resource` to be the *context* of a request. In English,

this means that when the URL `/` is visited, because `MyModel` is the root model, this view callable will be invoked.

The `renderer` argument names an *asset specification* of `templates/mytemplate.pt`. This asset specification points at a *Chameleon* template which lives in the `mytemplate.pt` file within the `templates` directory of the tutorial package. And indeed if you look in the `templates` directory of this package, you'll see a `mytemplate.pt` template file, which renders the default home page of the generated project. This asset specification is *relative* (to the `view.py`'s current package). Alternatively we could have used the absolute asset specification `tutorial:templates/mytemplate.pt`, but chose to use the relative version.

Since this call to `@view_config` doesn't pass a `name` argument, the `my_view` function which it decorates represents the "default" view callable used when the context is of the type `MyModel`.

3. *Lines 6-7.* We define a *view callable* named `my_view`, which we decorated in the step above. This view callable is a *function* we write generated by the `zodb` scaffold that is given a `request` and which returns a dictionary. The `mytemplate.pt` *renderer* named by the asset specification in the step above will convert this dictionary to a *response* on our behalf.

The function returns the dictionary `{ 'project': 'tutorial' }`. This dictionary is used by the template named by the `mytemplate.pt` asset specification to fill in certain values on the page.

Configuration in `development.ini`

The `development.ini` (in the tutorial *project* directory, as opposed to the tutorial *package* directory) looks like this:

```
###
# app configuration
# http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
# ↪environment.html
###

[app:main]
use = egg:tutorial

pyramid.reload_templates = true
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.debug_routematch = false
pyramid.default_locale_name = en
pyramid.includes =
```

```
pyramid_debugtoolbar
pyramid_zodbconn
pyramid_tm

tm.attempts = 3
zodbconn.uri = file://%(here)s/Data.fs?connection_cache_size=20000

# By default, the toolbar only appears for clients from IP addresses
# '127.0.0.1' and '::1'.
# debugtoolbar.hosts = 127.0.0.1 ::1

###
# wsgi server configuration
###

[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 6543

###
# logging configuration
# http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
# ↪ logging.html
###

[loggers]
keys = root, tutorial

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

[logger_tutorial]
level = DEBUG
handlers =
qualname = tutorial

[handler_console]
class = StreamHandler
```

```
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(asctime)s %(levelname)-5.5s [% (name)s:% (lineno)s] [
↳ %(threadName)s] %(message)s
```

Note the existence of a `[app:main]` section which specifies our WSGI application. Our ZODB database settings are specified as the `zodbconn.uri` setting within this section. This value, and the other values within this section, are passed as `**settings` to the main function we defined in `__init__.py` when the server is started via `pserve`.

Defining the Domain Model

The first change we'll make to our stock `pcreate`-generated application will be to define two *resource* constructors, one representing a wiki page, and another representing the wiki as a mapping of wiki page names to page objects. We'll do this inside our `models.py` file.

Because we're using *ZODB* to represent our *resource tree*, each of these resource constructors represents a *domain model* object, so we'll call these constructors "model constructors". Both our `Page` and `Wiki` constructors will be class objects. A single instance of the "Wiki" class will serve as a container for "Page" objects, which will be instances of the "Page" class.

Delete the database

In the next step, we're going to remove the `MyModel` Python model class from our `models.py` file. Since this class is referred to within our persistent storage (represented on disk as a file named `Data.fs`), we'll have strange things happen the next time we want to visit the application in a browser. Remove the `Data.fs` from the `tutorial` directory before proceeding any further. It's always fine to do this as long as you don't care about the content of the database; the database itself will be recreated as necessary.

Edit `models.py`



There is nothing special about the filename `models.py`. A project may have many models throughout its codebase in arbitrarily named files. Files implementing models often have `model` in their

filenames or they may live in a Python subpackage of your application package named `models`, but this is only by convention.

Open `tutorial/models.py` file and edit it to look like the following:

```

1 from persistent import Persistent
2 from persistent.mapping import PersistentMapping
3
4 class Wiki(PersistentMapping):
5     __name__ = None
6     __parent__ = None
7
8 class Page(Persistent):
9     def __init__(self, data):
10         self.data = data
11
12 def appmaker(zodb_root):
13     if 'app_root' not in zodb_root:
14         app_root = Wiki()
15         frontpage = Page('This is the front page')
16         app_root['FrontPage'] = frontpage
17         frontpage.__name__ = 'FrontPage'
18         frontpage.__parent__ = app_root
19         zodb_root['app_root'] = app_root
20         import transaction
21         transaction.commit()
22     return zodb_root['app_root']

```

The first thing we want to do is remove the `MyModel` class from the generated `models.py` file. The `MyModel` class is only a sample and we’re not going to use it.

Then, we’ll add a `Wiki` class. We want it to inherit from the `persistent.mapping.PersistentMapping` class because it provides mapping behavior, and it makes sure that our `Wiki` page is stored as a “first-class” persistent object in our ZODB database.

Our `Wiki` class should have two attributes set to `None` at class scope: `__parent__` and `__name__`. If a model has a `__parent__` attribute of `None` in a traversal-based Pyramid application, it means that it’s the *root* model. The `__name__` of the root model is also always `None`.

Then we’ll add a `Page` class. This class should inherit from the `persistent.Persistent` class. We’ll also give it an `__init__` method that accepts a single parameter named `data`. This parameter will contain the *reStructuredText* body representing the wiki page content. Note that `Page` objects don’t have an initial `__name__` or `__parent__` attribute. All objects in a traversal graph must have a `__name__`

and a `__parent__` attribute. We don't specify these here because both `__name__` and `__parent__` will be set by a *view* function when a *Page* is added to our Wiki mapping.

As a last step, we want to change the `appmaker` function in our `models.py` file so that the *root resource* of our application is a *Wiki* instance. We'll also slot a single page object (the front page) into the *Wiki* within the `appmaker`. This will provide *traversal* a *resource tree* to work against when it attempts to resolve URLs to resources.

View the application in a browser


We can't. At this point, our system is in a “non-runnable” state; we'll need to change view-related files in the next chapter to be able to start the application successfully. If you try to start the application (See *Start the application*), you'll wind up with a Python traceback on your console that ends with this exception:

```
ImportError: cannot import name MyModel
```

This will also happen if you attempt to run the tests.

Defining Views

A *view callable* in a *traversal*-based Pyramid application is typically a simple Python function that accepts two parameters: *context* and *request*. A view callable is assumed to return a *response* object.

 A Pyramid view can also be defined as callable which accepts *only* a *request* argument. You'll see this one-argument pattern used in other Pyramid tutorials and applications. Either calling convention will work in any Pyramid application; the calling conventions can be used interchangeably as necessary. In *traversal* based applications, URLs are mapped to a context *resource*, and since our *resource tree* also represents our application's “domain model”, we're often interested in the context because it represents the persistent storage of our application. For this reason, in this tutorial we define views as callables that accept `context` in the callable argument list. If you do need the `context` within a view function that only takes the request as a single argument, you can obtain it via `request.context`.

We're going to define several *view callable* functions, then wire them into Pyramid using some *view configuration*.

Declaring Dependencies in Our `setup.py` File

The view code in our application will depend on a package which is not a dependency of the original “tutorial” application. The original “tutorial” application was generated by the `pcreate` command; it doesn’t know about our custom application requirements.

We need to add a dependency on the `docutils` package to our tutorial package’s `setup.py` file by assigning this dependency to the `requires` parameter in the `setup()` function.

Open `setup.py` and edit it to look like the following:

```

1  import os
2
3  from setuptools import setup, find_packages
4
5  here = os.path.abspath(os.path.dirname(__file__))
6  with open(os.path.join(here, 'README.txt')) as f:
7      README = f.read()
8  with open(os.path.join(here, 'CHANGES.txt')) as f:
9      CHANGES = f.read()
10
11  requires = [
12      'pyramid',
13      'pyramid_chameleon',
14      'pyramid_debugtoolbar',
15      'pyramid_tm',
16      'pyramid_zodbconn',
17      'transaction',
18      'ZODB3',
19      'waitress',
20      'docutils',
21  ]
22
23  tests_require = [
24      'WebTest >= 1.3.1', # py3 compat
25      'pytest', # includes virtualenv
26      'pytest-cov',
27  ]
28
29  setup(name='tutorial',
30        version='0.0',
31        description='tutorial',
32        long_description=README + '\n\n' + CHANGES,
33        classifiers=[
34            "Programming Language :: Python",

```

```
35         "Framework :: Pyramid",
36         "Topic :: Internet :: WWW/HTTP",
37         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
38     ],
39     author='',
40     author_email='',
41     url='',
42     keywords='web pylons pyramid',
43     packages=find_packages(),
44     include_package_data=True,
45     zip_safe=False,
46     extras_require={
47         'testing': tests_require,
48     },
49     install_requires=requires,
50     entry_points="""\
51     [paste.app_factory]
52     main = tutorial:main
53     """,
54 )
```

Only the highlighted line needs to be added.

Running `pip install -e .`

Since a new software dependency was added, you will need to run `pip install -e .` again inside the root of the `tutorial` package to obtain and register the newly added dependency distribution.

Make sure your current working directory is the root of the project (the directory in which `setup.py` lives) and execute the following command.

On UNIX:

```
$ cd tutorial
$ $ENV/bin/pip install -e .
```

On Windows:

```
c:\pyramiddtut> cd tutorial
c:\pyramiddtut\tutorial> %ENV%\Scripts\pip install -e .
```

Success executing this command will end with a line to the console something like:

Successfully installed docutils-0.12 tutorial-0.0

Adding view functions in `views.py`

It's time for a major change. Open `tutorial/views.py` and edit it to look like the following:

```

1  from docutils.core import publish_parts
2  import re
3
4  from pyramid.httpexceptions import HTTPFound
5  from pyramid.view import view_config
6
7  from .models import Page
8
9  # regular expression used to find WikiWords
10 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+) ")
11
12 @view_config(context='.models.Wiki')
13 def view_wiki(context, request):
14     return HTTPFound(location=request.resource_url(context, 'FrontPage'))
15
16 @view_config(context='.models.Page', renderer='templates/view.pt')
17 def view_page(context, request):
18     wiki = context.__parent__
19
20     def check(match):
21         word = match.group(1)
22         if word in wiki:
23             page = wiki[word]
24             view_url = request.resource_url(page)
25             return '<a href="%s">%s</a>' % (view_url, word)
26         else:
27             add_url = request.application_url + '/add_page/' + word
28             return '<a href="%s">%s</a>' % (add_url, word)
29
30     content = publish_parts(context.data, writer_name='html')['html_body']
31     content = wikiwords.sub(check, content)
32     edit_url = request.resource_url(context, 'edit_page')
33     return dict(page = context, content = content, edit_url = edit_url)
34
35 @view_config(name='add_page', context='.models.Wiki',
36             renderer='templates/edit.pt')
37 def add_page(context, request):

```

```
38     pagename = request.subpath[0]
39     if 'form.submitted' in request.params:
40         body = request.params['body']
41         page = Page(body)
42         page.__name__ = pagename
43         page.__parent__ = context
44         context[pagename] = page
45         return HTTPFound(location = request.resource_url(page))
46     save_url = request.resource_url(context, 'add_page', pagename)
47     page = Page('')
48     page.__name__ = pagename
49     page.__parent__ = context
50     return dict(page = page, save_url = save_url)
51
52 @view_config(name='edit_page', context='.models.Page',
53             renderer='templates/edit.pt')
54 def edit_page(context, request):
55     if 'form.submitted' in request.params:
56         context.data = request.params['body']
57         return HTTPFound(location = request.resource_url(context))
58
59     return dict(page=context,
60               save_url=request.resource_url(context, 'edit_page'))
```

We added some imports and created a regular expression to find “WikiWords”.

We got rid of the `my_view` view function and its decorator that was added when we originally rendered the `zodb` scaffold. It was only an example and isn’t relevant to our application.

Then we added four *view callable* functions to our `views.py` module:

- `view_wiki()` - Displays the wiki itself. It will answer on the root URL.
- `view_page()` - Displays an individual page.
- `add_page()` - Allows the user to add a page.
- `edit_page()` - Allows the user to edit a page.

We’ll describe each one briefly in the following sections.




There is nothing special about the filename `views.py`. A project may have many view callables throughout its codebase in arbitrarily named files. Files implementing view callables often have `view` in their filenames (or may live in a Python subpackage of your application package named `views`), but this is only by convention.

The `view_wiki` view function

Following is the code for the `view_wiki` view function and its decorator:

```
12 @view_config(context='.models.Wiki')
13 def view_wiki(context, request):
14     return HTTPFound(location=request.resource_url(context, 'FrontPage'))
```

 In our code, we use an *import* that is *relative* to our package named `tutorial`, meaning we can omit the name of the package in the `import` and `context` statements. In our narrative, however, we refer to a *class* and thus we use the *absolute* form, meaning that the name of the package is included.

`view_wiki()` is the *default view* that gets called when a request is made to the root URL of our wiki. It always redirects to an URL which represents the path to our “FrontPage”.

We provide it with a `@view_config` decorator which names the class `tutorial.models.Wiki` as its context. This means that when a Wiki resource is the context and no *view name* exists in the request, then this view will be used. The view configuration associated with `view_wiki` does not use a *renderer* because the view callable always returns a *response* object rather than a dictionary. No *renderer* is necessary when a view returns a response object.

The `view_wiki` view callable always redirects to the URL of a Page resource named “FrontPage”. To do so, it returns an instance of the `pyramid.httpexceptions.HTTPFound` class (instances of which implement the `pyramid.interfaces.IResponse` interface, like `pyramid.response.Response` does). It uses the `pyramid.request.Request.route_url()` API to construct an URL to the FrontPage page resource (i.e., `http://localhost:6543/FrontPage`), and uses it as the “location” of the `HTTPFound` response, forming an HTTP redirect.

The `view_page` view function

Here is the code for the `view_page` view function and its decorator:

```
16 @view_config(context='.models.Page', renderer='templates/view.pt')
17 def view_page(context, request):
18     wiki = context.__parent__
19
20     def check(match):
21         word = match.group(1)
```

```
22     if word in wiki:
23         page = wiki[word]
24         view_url = request.resource_url(page)
25         return '<a href="%s">%s</a>' % (view_url, word)
26     else:
27         add_url = request.application_url + '/add_page/' + word
28         return '<a href="%s">%s</a>' % (add_url, word)
29
30     content = publish_parts(context.data, writer_name='html')['html_body']
31     content = wikiwords.sub(check, content)
32     edit_url = request.resource_url(context, 'edit_page')
33     return dict(page = context, content = content, edit_url = edit_url)
```

The `view_page` function is configured to respond as the default view of a `Page` resource. We provide it with a `@view_config` decorator which names the class `tutorial.models.Page` as its context. This means that when a `Page` resource is the context, and no *view name* exists in the request, this view will be used. We inform Pyramid this view will use the `templates/view.pt` template file as a renderer.

The `view_page` function generates the *reStructuredText* body of a page (stored as the `data` attribute of the context passed to the view; the context will be a `Page` resource) as HTML. Then it substitutes an HTML anchor for each *WikiWord* reference in the rendered HTML using a compiled regular expression.

The curried function named `check` is used as the first argument to `wikiwords.sub`, indicating that it should be called to provide a value for each *WikiWord* match found in the content. If the wiki (our page's `__parent__`) already contains a page with the matched *WikiWord* name, the `check` function generates a view link to be used as the substitution value and returns it. If the wiki does not already contain a page with the matched *WikiWord* name, the function generates an “add” link as the substitution value and returns it.

As a result, the `content` variable is now a fully formed bit of HTML containing various view and add links for *WikiWords* based on the content of our current page resource.

We then generate an edit URL because it's easier to do here than in the template, and we wrap up a number of arguments in a dictionary and return it.

The arguments we wrap into a dictionary include `page`, `content`, and `edit_url`. As a result, the *template* associated with this view callable (via `renderer=` in its configuration) will be able to use these names to perform various rendering tasks. The template associated with this view callable will be a template which lives in `templates/view.pt`.

Note the contrast between this view callable and the `view_wiki` view callable. In the `view_wiki` view callable, we unconditionally return a *response* object. In the `view_page` view callable, we return a *dictionary*. It is *always* fine to return a *response* object from a Pyramid view. Returning a dictionary is allowed only when there is a *renderer* associated with the view callable in the view configuration.

The add_page view function

Here is the code for the add_page view function and its decorator:

```

35 @view_config(name='add_page', context='.models.Wiki',
36             renderer='templates/edit.pt')
37 def add_page(context, request):
38     pagename = request.subpath[0]
39     if 'form.submitted' in request.params:
40         body = request.params['body']
41         page = Page(body)
42         page.__name__ = pagename
43         page.__parent__ = context
44         context[pagename] = page
45         return HTTPFound(location = request.resource_url(page))
46     save_url = request.resource_url(context, 'add_page', pagename)
47     page = Page('')
48     page.__name__ = pagename
49     page.__parent__ = context
50     return dict(page = page, save_url = save_url)

```

The add_page function is configured to respond when the context resource is a Wiki and the *view name* is add_page. We provide it with a @view_config decorator which names the string add_page as its *view name* (via name=), the class tutorial.models.Wiki as its context, and the renderer named templates/edit.pt. This means that when a Wiki resource is the context, and a *view name* named add_page exists as the result of traversal, this view will be used. We inform Pyramid this view will use the templates/edit.pt template file as a renderer. We share the same template between add and edit views, thus edit.pt instead of add.pt.

The add_page function will be invoked when a user clicks on a WikiWord which isn't yet represented as a page in the system. The check function within the view_page view generates URLs to this view. It also acts as a handler for the form that is generated when we want to add a page resource. The context of the add_page view is always a Wiki resource (*not* a Page resource).

The request *subpath* in Pyramid is the sequence of names that are found *after* the *view name* in the URL segments given in the PATH_INFO of the WSGI request as the result of *traversal*. If our add view is invoked via, e.g., http://localhost:6543/add_page/SomeName, the *subpath* will be a tuple: ('SomeName',).

The add view takes the zeroth element of the subpath (the wiki page name), and aliases it to the name attribute in order to know the name of the page we're trying to add.

If the view rendering is *not* a result of a form submission (if the expression 'form.submitted' in request.params is False), the view renders a template. To do so, it generates a "save url" which

the template uses as the form post URL during rendering. We're lazy here, so we're trying to use the same template (`templates/edit.pt`) for the add view as well as the page edit view. To do so, we create a dummy Page resource object in order to satisfy the edit form's desire to have *some* page object exposed as page, and we'll render the template to a response.

If the view rendering *is* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `True`), we grab the page body from the form data, create a Page object using the name in the subpath and the page body, and save it into “our context” (the Wiki) using the `__setitem__` method of the context. We then redirect back to the `view_page` view (the default view for a page) for the newly created page.

The `edit_page` view function

Here is the code for the `edit_page` view function and its decorator:

```
52 @view_config(name='edit_page', context='.models.Page',
53             renderer='templates/edit.pt')
54 def edit_page(context, request):
55     if 'form.submitted' in request.params:
56         context.data = request.params['body']
57         return HTTPFound(location = request.resource_url(context))
58
59     return dict(page=context,
60               save_url=request.resource_url(context, 'edit_page'))
```

The `edit_page` function is configured to respond when the context is a Page resource and the *view name* is `edit_page`. We provide it with a `@view_config` decorator which names the string `edit_page` as its *view name* (via `name=`), the class `tutorial.models.Page` as its context, and the renderer named `templates/edit.pt`. This means that when a Page resource is the context, and a *view name* exists as the result of traversal named `edit_page`, this view will be used. We inform Pyramid this view will use the `templates/edit.pt` template file as a renderer.

The `edit_page` function will be invoked when a user clicks the “Edit this Page” button on the view form. It renders an edit form but it also acts as the form post view callable for the form it renders. The context of the `edit_page` view will *always* be a Page resource (never a Wiki resource).

If the view execution is *not* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `False`), the view simply renders the edit form, passing the page resource, and a `save_url` which will be used as the action of the generated form.

If the view execution *is* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `True`), the view grabs the `body` element of the request parameter and sets it as the `data` attribute of the page context. It then redirects to the default view of the context (the page), which will always be the `view_page` view.

Adding templates

The `view_page`, `add_page` and `edit_page` views that we've added reference a *template*. Each template is a *Chameleon ZPT* template. These templates will live in the `templates` directory of our tutorial package. Chameleon templates must have a `.pt` extension to be recognized as such.

The `view.pt` template

Create `tutorial/templates/view.pt` and add the following content:

```

1  <!DOCTYPE html>
2  <html lang="{request.locale_name}">
3    <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta name="description" content="pyramid web application">
8      <meta name="author" content="Pylons Project">
9      <link rel="shortcut icon" href="{request.static_url('tutorial:static/
    ↳ pyramid-16x16.png')}">
10
11     <title>${page.__name__} - Pyramid tutorial wiki (based on
12     TurboGears 20-Minute Wiki)</title>
13
14     <!-- Bootstrap core CSS -->
15     <link href="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/css/
    ↳ bootstrap.min.css" rel="stylesheet">
16
17     <!-- Custom styles for this scaffold -->
18     <link href="{request.static_url('tutorial:static/theme.css')}" rel=
    ↳ "stylesheet">
19
20     <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media_
    ↳ queries -->
21     <!--[if lt IE 9]>
22       <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></
    ↳ script>
23       <script src="//oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js">
    ↳ </script>
24     <![endif]-->
25   </head>
26   <body>
27

```

```
28     <div class="starter-template">
29         <div class="container">
30             <div class="row">
31                 <div class="col-md-2">
32                     
34                 </div>
35                 <div class="col-md-10">
36                     <div class="content">
37                         <div tal:replace="structure content">
38                             Page text goes here.
39                         </div>
40                         <p>
41                             <a tal:attributes="href edit_url" href="">
42                                 Edit this page
43                             </a>
44                         </p>
45                         <p>
46                             Viewing <strong><span tal:replace="page.__name__">
47                                 Page Name Goes Here</span></strong>
48                         </p>
49                         <p>You can return to the
50                             <a href="{request.application_url}">FrontPage</a>.
51                         </p>
52                     </div>
53                 </div>
54             <div class="row">
55                 <div class="copyright">
56                     Copyright &copy; Pylons Project
57                 </div>
58             </div>
59         </div>
60     </div>
61
62     <!-- Bootstrap core JavaScript
63     ===== -->
64     <!-- Placed at the end of the document so the pages load faster -->
65     <script src="//oss.maxcdn.com/libs/jquery/1.10.2/jquery.min.js"></
66 ↪script>
67     <script src="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/js/
68 ↪bootstrap.min.js"></script>
69 </body>
70 </html>
```

This template is used by `view_page()` for displaying a single wiki page. It includes:

- A `div` element that is replaced with the `content` value provided by the view (lines 36-38). `content` contains HTML, so the `structure` keyword is used to prevent escaping it (i.e., changing “>” to “>”, etc.)
- A link that points at the “edit” URL which invokes the `edit_page` view for the page being viewed (lines 40-42).

The `edit.pt` template

Create `tutorial/templates/edit.pt` and add the following content:

```

1  <!DOCTYPE html>
2  <html lang="{request.locale_name}">
3    <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta name="description" content="pyramid web application">
8      <meta name="author" content="Pylons Project">
9      <link rel="shortcut icon" href="{request.static_url('tutorial:static/
    ↪pyramid-16x16.png')}">
10
11     <title>{page.__name__} - Pyramid tutorial wiki (based on
12     TurboGears 20-Minute Wiki)</title>
13
14     <!-- Bootstrap core CSS -->
15     <link href="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/css/
    ↪bootstrap.min.css" rel="stylesheet">
16
17     <!-- Custom styles for this scaffold -->
18     <link href="{request.static_url('tutorial:static/theme.css')}" rel=
    ↪"stylesheet">
19
20     <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media_
    ↪queries -->
21     <!--[if lt IE 9]>
22       <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></
    ↪script>
23       <script src="//oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js">
    ↪</script>
24     <![endif]-->
25   </head>

```


```
26 <body>
27
28 <div class="starter-template">
29 <div class="container">
30 <div class="row">
31 <div class="col-md-2">
32 
33 </div>
34 <div class="col-md-10">
35 <div class="content">
36 <p>
37     Editing <strong><span tal:replace="page.__name__">
38     Page Name Goes Here</span></strong>
39 </p>
40 <p>You can return to the
41 <a href="{request.application_url}">FrontPage</a>.
42 </p>
43 <form action="{save_url}" method="post">
44 <div class="form-group">
45 <textarea class="form-control" name="body" tal:content=
↪ "page.data" rows="10" cols="60"></textarea>
46 </div>
47 <div class="form-group">
48 <button type="submit" name="form.submitted" value="Save"
↪ class="btn btn-default">Save</button>
49 </div>
50 </form>
51 </div>
52 </div>
53 </div>
54 <div class="row">
55 <div class="copyright">
56     Copyright &copy; Pylons Project
57 </div>
58 </div>
59 </div>
60 </div>
61
62
63 <!-- Bootstrap core JavaScript
64 ===== -->
65 <!-- Placed at the end of the document so the pages load faster -->
66 <script src="//oss.maxcdn.com/libs/jquery/1.10.2/jquery.min.js"></
↪ script>
67 <script src="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/js/
↪ bootstrap.min.js"></script>
```

```
68 | </body>
69 | </html>
```

This template is used by `add_page()` and `edit_page()` for adding and editing a wiki page. It displays a page containing a form that includes:

- A 10 row by 60 column `textarea` field named `body` that is filled with any existing page data when it is rendered (line 45).
- A submit button that has the name `form.submitted` (line 48).

The form POSTs back to the `save_url` argument supplied by the view (line 43). The view will use the `body` and `form.submitted` values.

 Our templates use a `request` object that none of our tutorial views return in their dictionary. `request` is one of several names that are available “by default” in a template when a template renderer is used. See *System Values Used During Rendering* for information about other names that are available by default when a template is used as a renderer.

Static assets

Our templates name static assets, including CSS and images. We don’t need to create these files within our package’s `static` directory because they were provided at the time we created the project.

As an example, the CSS file will be accessed via `http://localhost:6543/static/theme.css` by virtue of the call to the `add_static_view` directive we’ve made in the `__init__.py` file. Any number and type of static assets can be placed in this directory (or subdirectories) and are just referred to by URL or by using the convenience method `static_url`, e.g., `request.static_url('<package>:static/foo.css')` within templates.

Viewing the application in a browser

We can finally examine our application in a browser (See *Start the application*). Launch a browser and visit each of the following URLs, checking that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` resource.

- `http://localhost:6543/FrontPage/` invokes the `view_page` view of the front page resource. This is because it's the *default view* (a view without a name) for Page resources.
- `http://localhost:6543/FrontPage/edit_page` invokes the edit view for the `FrontPage` Page resource.
- `http://localhost:6543/add_page/SomePageName` invokes the add view for a Page.
- To generate an error, visit `http://localhost:6543/add_page` which will generate an `IndexError: tuple index out of range` error. You'll see an interactive traceback facility provided by `pyramid_debugtoolbar`.

Adding authorization

Pyramid provides facilities for *authentication* and *authorization*. We'll make use of both features to provide security to our application. Our application currently allows anyone with access to the server to view, edit, and add pages to our wiki. We'll change that to allow only people who are members of a *group* named `group:editors` to add and edit wiki pages, but we'll continue allowing anyone with access to the server to view pages.

We will also add a login page and a logout link on all the pages. The login page will be shown when a user is denied access to any of the views that require permission, instead of a default "403 Forbidden" page.

We will implement the access control with the following steps:

- Add users and groups (`security.py`, a new module).
- Add an *ACL* (`models.py`).
- Add an *authentication policy* and an *authorization policy* (`__init__.py`).
- Add *permission* declarations to the `edit_page` and `add_page` views (`views.py`).

Then we will add the login and logout feature:

- Add login and logout views (`views.py`).
- Add a login template (`login.pt`).
- Make the existing views return a `logged_in` flag to the renderer (`views.py`).
- Add a "Logout" link to be shown when logged in and viewing or editing a page (`view.pt`, `edit.pt`).

Access control

Add users and groups

Create a new `tutorial/security.py` module with the following content:

```

1  USERS = {'editor': 'editor',
2          'viewer': 'viewer'}
3  GROUPS = {'editor': ['group:editors']}
4
5  def groupfinder(userid, request):
6      if userid in USERS:
7          return GROUPS.get(userid, [])

```

The `groupfinder` function accepts a `userid` and a `request` and returns one of these values:

- If the `userid` exists in the system, it will return a sequence of group identifiers (or an empty sequence if the user isn't a member of any groups).
- If the `userid` *does not* exist in the system, it will return `None`.

For example, `groupfinder('editor', request)` returns `['group:editor']`, `groupfinder('viewer', request)` returns `[]`, and `groupfinder('admin', request)` returns `None`. We will use `groupfinder()` as an *authentication policy* “callback” that will provide the *principal* or principals for a user.

In a production system, user and group data will most often come from a database, but here we use “dummy” data to represent user and groups sources.

Add an ACL

Open `tutorial/models.py` and add the following import statement at the head:

```

1  from pyramid.security import (
2      Allow,
3      Everyone,
4      )

```

Add the following lines to the `Wiki` class:


```
9 class Wiki(PersistentMapping):
10     __name__ = None
11     __parent__ = None
12     __acl__ = [ (Allow, Everyone, 'view'),
13                 (Allow, 'group:editors', 'edit') ]
```

We import *Allow*, an action that means that permission is allowed, and *Everyone*, a special *principal* that is associated to all requests. Both are used in the *ACE* entries that make up the ACL.

The ACL is a list that needs to be named `__acl__` and be an attribute of a class. We define an *ACL* with two *ACE* entries: the first entry allows any user the *view* permission. The second entry allows the `group:editors` principal the *edit* permission.

The `Wiki` class that contains the ACL is the *resource* constructor for the *root* resource, which is a `Wiki` instance. The ACL is provided to each view in the *context* of the request as the `context` attribute.

It's only happenstance that we're assigning this ACL at class scope. An ACL can be attached to an object *instance* too; this is how “row level security” can be achieved in Pyramid applications. We actually need only *one* ACL for the entire system, however, because our security requirements are simple, so this feature is not demonstrated. See *Assigning ACLs to Your Resource Objects* for more information about what an *ACL* represents.

Add authentication and authorization policies

Open `tutorial/__init__.py` and add the highlighted import statements:

```
1 from pyramid.config import Configurator
2 from pyramid_zodbconn import get_connection
3
4 from pyramid.authentication import AuthTktAuthenticationPolicy
5 from pyramid.authorization import ACLAuthorizationPolicy
6
7 from .models import appmaker
8 from .security import groupfinder
```

Now add those policies to the configuration:

```

18     authn_policy = AuthTktAuthenticationPolicy(
19         'sosecret', callback=groupfinder, hashalg='sha512')
20     authz_policy = ACLAuthorizationPolicy()
21     config = Configurator(root_factory=root_factory, settings=settings)
22     config.set_authentication_policy(authn_policy)
23     config.set_authorization_policy(authz_policy)

```

Only the highlighted lines need to be added.

We are enabling an `AuthTktAuthenticationPolicy`, which is based in an auth ticket that may be included in the request. We are also enabling an `ACLAuthorizationPolicy`, which uses an ACL to determine the *allow* or *deny* outcome for a view.

Note that the `pyramid.authentication.AuthTktAuthenticationPolicy` constructor accepts two arguments: `secret` and `callback`. `secret` is a string representing an encryption key used by the “authentication ticket” machinery represented by this policy: it is required. The `callback` is the `groupfinder()` function that we created before.

Add permission declarations

Open `tutorial/views.py` and add a `permission='edit'` parameter to the `@view_config` decorators for `add_page()` and `edit_page()`:

```

@view_config(name='add_page', context='.models.Wiki',
             renderer='templates/edit.pt',
             permission='edit')

```

```

@view_config(name='edit_page', context='.models.Page',
             renderer='templates/edit.pt',
             permission='edit')

```

Only the highlighted lines, along with their preceding commas, need to be edited and added.

The result is that only users who possess the `edit` permission at the time of the request may invoke those two views.

Add a `permission='view'` parameter to the `@view_config` decorator for `view_wiki()` and `view_page()` as follows:

```
@view_config(context='.models.Wiki',  
             permission='view')
```

```
@view_config(context='.models.Page', renderer='templates/view.pt',  
             permission='view')
```

Only the highlighted lines, along with their preceding commas, need to be edited and added.

This allows anyone to invoke these two views.

We are done with the changes needed to control access. The changes that follow will add the login and logout feature.

Login, logout

Add login and logout views

We'll add a `login` view which renders a login form and processes the post from the login form, checking credentials.

We'll also add a `logout` view callable to our application and provide a link to it. This view will clear the credentials of the logged in user and redirect back to the front page.

Add the following import statements to the head of `tutorial/views.py`:

```
from pyramid.view import (  
    view_config,  
    forbidden_view_config,  
)  
  
from pyramid.security import (  
    remember,  
    forget,  
)  
  
from .security import USERS
```

All the highlighted lines need to be added or edited.

`forbidden_view_config()` will be used to customize the default 403 Forbidden page. `remember()` and `forget()` help to create and expire an auth ticket cookie.

Now add the login and logout views at the end of the file:

```

82 @view_config(context='.models.Wiki', name='login',
83             renderer='templates/login.pt')
84 @forbidden_view_config(renderer='templates/login.pt')
85 def login(request):
86     login_url = request.resource_url(request.context, 'login')
87     referrer = request.url
88     if referrer == login_url:
89         referrer = '/' # never use the login form itself as came_from
90     came_from = request.params.get('came_from', referrer)
91     message = ''
92     login = ''
93     password = ''
94     if 'form.submitted' in request.params:
95         login = request.params['login']
96         password = request.params['password']
97         if USERS.get(login) == password:
98             headers = remember(request, login)
99             return HTTPFound(location=came_from,
100                             headers=headers)
101         message = 'Failed login'
102
103     return dict(
104         message=message,
105         url=request.application_url + '/login',
106         came_from=came_from,
107         login=login,
108         password=password,
109     )
110
111
112 @view_config(context='.models.Wiki', name='logout')
113 def logout(request):
114     headers = forget(request)
115     return HTTPFound(location=request.resource_url(request.context),
116                     headers=headers)

```

`login()` has two decorators:

- a `@view_config` decorator which associates it with the login route and makes it visible when we visit `/login`,

- a `@forbidden_view_config` decorator which turns it into a *forbidden view*. `login()` will be invoked when a user tries to execute a view callable for which they lack authorization. For example, if a user has not logged in and tries to add or edit a Wiki page, they will be shown the login form before being allowed to continue.

The order of these two *view configuration* decorators is unimportant.

`logout()` is decorated with a `@view_config` decorator which associates it with the `logout` route. It will be invoked when we visit `/logout`.

Add the `login.pt` Template

Create `tutorial/templates/login.pt` with the following content:

```
<!DOCTYPE html>
<html lang="{request.locale_name}">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="pyramid web application">
    <meta name="author" content="Pylons Project">
    <link rel="shortcut icon" href="{request.static_url('tutorial:static/
↪pyramid-16x16.png')}">

    <title>Login - Pyramid tutorial wiki (based on
    TurboGears 20-Minute Wiki)</title>

    <!-- Bootstrap core CSS -->
    <link href="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/css/
↪bootstrap.min.css" rel="stylesheet">

    <!-- Custom styles for this scaffold -->
    <link href="{request.static_url('tutorial:static/theme.css')}" rel=
↪"stylesheet">

    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media_
↪queries -->
    <!--[if lt IE 9]>
      <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></
↪script>
      <script src="//oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js">
↪</script>
    <![endif]-->
```

```

</head>
<body>

  <div class="starter-template">
    <div class="container">
      <div class="row">
        <div class="col-md-2">
          
        </div>
        <div class="col-md-10">
          <div class="content">
            <p>
              <strong>
                Login
              </strong><br>
              <span tal:replace="message"></span>
            </p>
            <form action="{url}" method="post">
              <input type="hidden" name="came_from" value="{came_from}">
              <div class="form-group">
                <label for="login">Username</label>
                <input type="text" name="login" value="{login}">
              </div>
              <div class="form-group">
                <label for="password">Password</label>
                <input type="password" name="password" value="{password}
↪ ">
              </div>
              <div class="form-group">
                <button type="submit" name="form.submitted" value="Log In
↪ " class="btn btn-default">Log In</button>
              </div>
            </form>
          </div>
        </div>
      </div>
      <div class="copyright">
        Copyright &copy; Pylons Project
      </div>
    </div>
  </div>
</div>

```

```
<!-- Bootstrap core JavaScript
===== -->
<!-- Placed at the end of the document so the pages load faster -->
<script src="//oss.maxcdn.com/libs/jquery/1.10.2/jquery.min.js"></
↪script>
<script src="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/js/
↪bootstrap.min.js"></script>
</body>
</html>
```

The above template is referenced in the login view that we just added in `views.py`.

Return a `logged_in` flag to the renderer

Open `tutorial/views.py` again. Add a `logged_in` parameter to the return value of `view_page()`, `add_page()`, and `edit_page()` as follows:

```
return dict(page=context, content=content, edit_url=edit_url,
            logged_in=request.authenticated_userid)
```

```
return dict(page=page, save_url=save_url,
            logged_in=request.authenticated_userid)
```

```
return dict(page=context,
            save_url=request.resource_url(context, 'edit_page'),
            logged_in=request.authenticated_userid)
```

Only the highlighted lines need to be added or edited.

The `pyramid.request.Request.authenticated_userid()` will be `None` if the user is not authenticated, or a `userid` if the user is authenticated.

Add a “Logout” link when logged in

Open `tutorial/templates/edit.pt` and `tutorial/templates/view.pt` and add the following code as indicated by the highlighted lines.

```

<div class="col-md-10">
  <div class="content">
    <p tal:condition="logged_in" class="pull-right">
      <a href="{request.application_url}/logout">Logout</a>
    </p>

```

The attribute `tal:condition="logged_in"` will make the element be included when `logged_in` is any user id. The link will invoke the logout view. The above element will not be included if `logged_in` is `None`, such as when a user is not authenticated.

Reviewing our changes

Our tutorial/`__init__.py` will look like this when we're done:

```

1  from pyramid.config import Configurator
2  from pyramid_zodbconn import get_connection
3
4  from pyramid.authentication import AuthTktAuthenticationPolicy
5  from pyramid.authorization import ACLAuthorizationPolicy
6
7  from .models import appmaker
8  from .security import groupfinder
9
10 def root_factory(request):
11     conn = get_connection(request)
12     return appmaker(conn.root())
13
14
15 def main(global_config, **settings):
16     """ This function returns a Pyramid WSGI application.
17     """
18     authn_policy = AuthTktAuthenticationPolicy(
19         'sosecret', callback=groupfinder, hashalg='sha512')
20     authz_policy = ACLAuthorizationPolicy()
21     config = Configurator(root_factory=root_factory, settings=settings)
22     config.set_authentication_policy(authn_policy)
23     config.set_authorization_policy(authz_policy)
24     config.include('pyramid_chameleon')
25     config.add_static_view('static', 'static', cache_max_age=3600)
26     config.scan()
27     return config.make_wsgi_app()

```


Only the highlighted lines need to be added or edited.

Our tutorial/models.py will look like this when we're done:

```
1 from persistent import Persistent
2 from persistent.mapping import PersistentMapping
3
4 from pyramid.security import (
5     Allow,
6     Everyone,
7 )
8
9 class Wiki(PersistentMapping):
10     __name__ = None
11     __parent__ = None
12     __acl__ = [ (Allow, Everyone, 'view'),
13                 (Allow, 'group:editors', 'edit') ]
14
15 class Page(Persistent):
16     def __init__(self, data):
17         self.data = data
18
19 def appmaker(zodb_root):
20     if 'app_root' not in zodb_root:
21         app_root = Wiki()
22         frontpage = Page('This is the front page')
23         app_root['FrontPage'] = frontpage
24         frontpage.__name__ = 'FrontPage'
25         frontpage.__parent__ = app_root
26         zodb_root['app_root'] = app_root
27         import transaction
28         transaction.commit()
29     return zodb_root['app_root']
```

Only the highlighted lines need to be added or edited.

Our tutorial/views.py will look like this when we're done:

```
1 from docutils.core import publish_parts
2 import re
3
4 from pyramid.httpexceptions import HTTPFound
5
6 from pyramid.view import (
7     view_config,
```

```

8         forbidden_view_config,
9     )
10
11 from pyramid.security import (
12     remember,
13     forget,
14 )
15
16
17 from .security import USERS
18 from .models import Page
19
20 # regular expression used to find WikiWords
21 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+) ")
22
23 @view_config(context='.models.Wiki',
24             permission='view')
25 def view_wiki(context, request):
26     return HTTPFound(location=request.resource_url(context, 'FrontPage'))
27
28 @view_config(context='.models.Page', renderer='templates/view.pt',
29             permission='view')
30 def view_page(context, request):
31     wiki = context.__parent__
32
33     def check(match):
34         word = match.group(1)
35         if word in wiki:
36             page = wiki[word]
37             view_url = request.resource_url(page)
38             return '<a href="%s">%s</a>' % (view_url, word)
39         else:
40             add_url = request.application_url + '/add_page/' + word
41             return '<a href="%s">%s</a>' % (add_url, word)
42
43     content = publish_parts(context.data, writer_name='html')['html_body']
44     content = wikiwords.sub(check, content)
45     edit_url = request.resource_url(context, 'edit_page')
46
47     return dict(page=context, content=content, edit_url=edit_url,
48               logged_in=request.authenticated_userid)
49
50 @view_config(name='add_page', context='.models.Wiki',
51             renderer='templates/edit.pt',
52             permission='edit')
53 def add_page(context, request):

```

```
54     pagename = request.subpath[0]
55     if 'form.submitted' in request.params:
56         body = request.params['body']
57         page = Page(body)
58         page.__name__ = pagename
59         page.__parent__ = context
60         context[pagename] = page
61         return HTTPFound(location=request.resource_url(page))
62     save_url = request.resource_url(context, 'add_page', pagename)
63     page = Page('')
64     page.__name__ = pagename
65     page.__parent__ = context
66
67     return dict(page=page, save_url=save_url,
68                 logged_in=request.authenticated_userid)
69
70 @view_config(name='edit_page', context='.models.Page',
71             renderer='templates/edit.pt',
72             permission='edit')
73 def edit_page(context, request):
74     if 'form.submitted' in request.params:
75         context.data = request.params['body']
76         return HTTPFound(location=request.resource_url(context))
77
78     return dict(page=context,
79                 save_url=request.resource_url(context, 'edit_page'),
80                 logged_in=request.authenticated_userid)
81
82 @view_config(context='.models.Wiki', name='login',
83             renderer='templates/login.pt')
84 @forbidden_view_config(renderer='templates/login.pt')
85 def login(request):
86     login_url = request.resource_url(request.context, 'login')
87     referrer = request.url
88     if referrer == login_url:
89         referrer = '/' # never use the login form itself as came_from
90     came_from = request.params.get('came_from', referrer)
91     message = ''
92     login = ''
93     password = ''
94     if 'form.submitted' in request.params:
95         login = request.params['login']
96         password = request.params['password']
97         if USERS.get(login) == password:
98             headers = remember(request, login)
99             return HTTPFound(location=came_from,
```

```

100             headers=headers)
101     message = 'Failed login'
102
103     return dict(
104         message=message,
105         url=request.application_url + '/login',
106         came_from=came_from,
107         login=login,
108         password=password,
109     )
110
111
112 @view_config(context='.models.Wiki', name='logout')
113 def logout(request):
114     headers = forget(request)
115     return HTTPFound(location=request.resource_url(request.context),
116                     headers=headers)

```

Only the highlighted lines need to be added or edited.

Our tutorial/templates/edit.pt template will look like this when we're done:

```

1  <!DOCTYPE html>
2  <html lang="${request.locale_name}">
3      <head>
4          <meta charset="utf-8">
5          <meta http-equiv="X-UA-Compatible" content="IE=edge">
6          <meta name="viewport" content="width=device-width, initial-scale=1.0">
7          <meta name="description" content="pyramid web application">
8          <meta name="author" content="Pylons Project">
9          <link rel="shortcut icon" href="${request.static_url('tutorial:static/
    ↳ pyramid-16x16.png')}">
10
11      <title>${page.__name__} - Pyramid tutorial wiki (based on
12      TurboGears 20-Minute Wiki)</title>
13
14      <!-- Bootstrap core CSS -->
15      <link href="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/css/
    ↳ bootstrap.min.css" rel="stylesheet">
16
17      <!-- Custom styles for this scaffold -->
18      <link href="${request.static_url('tutorial:static/theme.css')}" rel=
    ↳ "stylesheet">
19
20      <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media_
    ↳ queries -->

```

```
21     <!--[if lt IE 9]>
22     <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></
↪script>
23     <script src="//oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js">
↪</script>
24     <![endif]-->
25 </head>
26 <body>
27
28 <div class="starter-template">
29 <div class="container">
30 <div class="row">
31 <div class="col-md-2">
32 
33 </div>
34 <div class="col-md-10">
35 <div class="content">
36 <p tal:condition="logged_in" class="pull-right">
37 <a href="{request.application_url}/logout">Logout</a>
38 </p>
39 <p>
40     Editing <strong><span tal:replace="page.__name__">
41     Page Name Goes Here</span></strong>
42 </p>
43 <p>You can return to the
44 <a href="{request.application_url}">FrontPage</a>.
45 </p>
46 <form action="{save_url}" method="post">
47 <div class="form-group">
48 <textarea class="form-control" name="body" tal:content=
↪"page.data" rows="10" cols="60"></textarea>
49 </div>
50 <div class="form-group">
51 <button type="submit" name="form.submitted" value="Save"
↪class="btn btn-default">Save</button>
52 </div>
53 </form>
54 </div>
55 </div>
56 </div>
57 <div class="row">
58 <div class="copyright">
59     Copyright &copy; Pylons Project
60 </div>
61 </div>
```

```

62     </div>
63 </div>
64
65
66     <!-- Bootstrap core JavaScript
67     ===== -->
68     <!-- Placed at the end of the document so the pages load faster -->
69     <script src="//oss.maxcdn.com/libs/jquery/1.10.2/jquery.min.js"></
↪script>
70     <script src="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/js/
↪bootstrap.min.js"></script>
71 </body>
72 </html>

```

Only the highlighted lines need to be added or edited.

Our tutorial/templates/view.pt template will look like this when we're done:

```

1  <!DOCTYPE html>
2  <html lang="{request.locale_name}">
3    <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta name="description" content="pyramid web application">
8      <meta name="author" content="Pylons Project">
9      <link rel="shortcut icon" href="{request.static_url('tutorial:static/
↪pyramid-16x16.png')}">
10
11      <title>${page.__name__} - Pyramid tutorial wiki (based on
12      TurboGears 20-Minute Wiki)</title>
13
14      <!-- Bootstrap core CSS -->
15      <link href="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/css/
↪bootstrap.min.css" rel="stylesheet">
16
17      <!-- Custom styles for this scaffold -->
18      <link href="{request.static_url('tutorial:static/theme.css')}" rel=
↪"stylesheet">
19
20      <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media_
↪queries -->
21      <!--[if lt IE 9]>
22          <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></
↪script>
23          <script src="//oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js">
↪</script>

```

```
24     <![endif]-->
25 </head>
26 <body>
27
28     <div class="starter-template">
29         <div class="container">
30             <div class="row">
31                 <div class="col-md-2">
32                     
34                 </div>
35                 <div class="col-md-10">
36                     <div class="content">
37                         <p tal:condition="logged_in" class="pull-right">
38                             <a href="{request.application_url}/logout">Logout</a>
39                         </p>
40                         <div tal:replace="structure content">
41                             Page text goes here.
42                         </div>
43                         <p>
44                             <a tal:attributes="href edit_url" href="">
45                                 Edit this page
46                             </a>
47                         </p>
48                         <p>
49                             Viewing <strong><span tal:replace="page.__name__">
50                                 Page Name Goes Here</span></strong>
51                         </p>
52                         <p>You can return to the
53                             <a href="{request.application_url}">FrontPage</a>.
54                         </p>
55                     </div>
56                 </div>
57             <div class="row">
58                 <div class="copyright">
59                     Copyright &copy; Pylons Project
60                 </div>
61             </div>
62         </div>
63     </div>
64
65     <!-- Bootstrap core JavaScript
66     ===== -->
67
68     <!-- Placed at the end of the document so the pages load faster -->
```

```
69     <script src="//oss.maxcdn.com/libs/jquery/1.10.2/jquery.min.js"></  
    ↪script>  
70     <script src="//oss.maxcdn.com/libs/twitter-bootstrap/3.0.3/js/  
    ↪bootstrap.min.js"></script>  
71     </body>  
72 </html>
```

Only the highlighted lines need to be added or edited.

Viewing the application in a browser

We can finally examine our application in a browser (See *Start the application*). Launch a browser and visit each of the following URLs, checking that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` Page resource. It is executable by any user.
- `http://localhost:6543/FrontPage` invokes the `view_page` view of the `FrontPage` Page resource. This is because it's the *default view* (a view without a name) for Page resources. It is executable by any user.
- `http://localhost:6543/FrontPage/edit_page` invokes the edit view for the `FrontPage` object. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, a login form will be displayed. Supplying the credentials with the username `editor`, password `editor` will display the edit page form.
- `http://localhost:6543/add_page/SomePageName` invokes the add view for a page. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, a login form will be displayed. Supplying the credentials with the username `editor`, password `editor` will display the edit page form.
- After logging in (as a result of hitting an edit or add page and submitting the login form with the `editor` credentials), we'll see a Logout link in the upper right hand corner. When we click it, we're logged out, and redirected back to the front page.

Adding Tests

We will now add tests for the models and the views and a few functional tests in `tests.py`. Tests ensure that an application works, and that it continues to work when changes are made in the future.

Test the models

We write tests for the model classes and the appmaker. Changing `tests.py`, we'll write a separate test class for each model class, and we'll write a test class for the appmaker.

To do so, we'll retain the `tutorial.tests.ViewTests` class that was generated as part of the `zodb` scaffold. We'll add three test classes: one for the `Page` model named `PageModelTests`, one for the `Wiki` model named `WikiModelTests`, and one for the appmaker named `AppmakerTests`.

Test the views

We'll modify our `tests.py` file, adding tests for each view function we added previously. As a result, we'll *delete* the `ViewTests` class that the `zodb` scaffold provided, and add four other test classes: `ViewWikiTests`, `ViewPageTests`, `AddPageTests`, and `EditPageTests`. These test the `view_wiki`, `view_page`, `add_page`, and `edit_page` views.

Functional tests

We'll test the whole application, covering security aspects that are not tested in the unit tests, like logging in, logging out, checking that the `viewer` user cannot add or edit pages, but the `editor` user can, and so on.

View the results of all our edits to `tests.py`

Open the `tutorial/tests.py` module, and edit it such that it appears as follows:

```
1 import unittest
2
3 from pyramid import testing
4
5 class PageModelTests(unittest.TestCase):
6
7     def _getTargetClass(self):
8         from .models import Page
9         return Page
10
11     def _makeOne(self, data=u'some data'):
```

```

12         return self._getTargetClass()(data=data)
13
14     def test_constructor(self):
15         instance = self._makeOne()
16         self.assertEqual(instance.data, u'some data')
17
18 class WikiModelTests(unittest.TestCase):
19
20     def _getTargetClass(self):
21         from .models import Wiki
22         return Wiki
23
24     def _makeOne(self):
25         return self._getTargetClass>()
26
27     def test_it(self):
28         wiki = self._makeOne()
29         self.assertEqual(wiki.__parent__, None)
30         self.assertEqual(wiki.__name__, None)
31
32 class AppmakerTests(unittest.TestCase):
33
34     def _callFUT(self, zodb_root):
35         from .models import appmaker
36         return appmaker(zodb_root)
37
38     def test_it(self):
39         root = {}
40         self._callFUT(root)
41         self.assertEqual(root['app_root']['FrontPage'].data,
42                          'This is the front page')
43
44 class ViewWikiTests(unittest.TestCase):
45     def test_it(self):
46         from .views import view_wiki
47         context = testing.DummyResource()
48         request = testing.DummyRequest()
49         response = view_wiki(context, request)
50         self.assertEqual(response.location, 'http://example.com/FrontPage')
51
52 class ViewPageTests(unittest.TestCase):
53     def _callFUT(self, context, request):
54         from .views import view_page
55         return view_page(context, request)
56
57     def test_it(self):

```

```
58     wiki = testing.DummyResource()
59     wiki['IDoExist'] = testing.DummyResource()
60     context = testing.DummyResource(data='Hello CruelWorld IDoExist')
61     context.__parent__ = wiki
62     context.__name__ = 'thepage'
63     request = testing.DummyRequest()
64     info = self._callFUT(context, request)
65     self.assertEqual(info['page'], context)
66     self.assertEqual(
67         info['content'],
68         '<div class="document">\n'
69         '<p>Hello <a href="http://example.com/add_page/CruelWorld">'
70         'CruelWorld</a> '
71         '<a href="http://example.com/IDoExist/">'
72         'IDoExist</a>'
73         '</p>\n</div>\n')
74     self.assertEqual(info['edit_url'],
75                     'http://example.com/thepage/edit_page')
76
77
78 class AddPageTests(unittest.TestCase):
79     def _callFUT(self, context, request):
80         from .views import add_page
81         return add_page(context, request)
82
83     def test_it_notsubmitted(self):
84         context = testing.DummyResource()
85         request = testing.DummyRequest()
86         request.subpath = ['AnotherPage']
87         info = self._callFUT(context, request)
88         self.assertEqual(info['page'].data, '')
89         self.assertEqual(
90             info['save_url'],
91             request.resource_url(context, 'add_page', 'AnotherPage'))
92
93     def test_it_submitted(self):
94         context = testing.DummyResource()
95         request = testing.DummyRequest({'form.submitted': True,
96                                         'body': 'Hello yo!'})
97         request.subpath = ['AnotherPage']
98         self._callFUT(context, request)
99         page = context['AnotherPage']
100         self.assertEqual(page.data, 'Hello yo!')
101         self.assertEqual(page.__name__, 'AnotherPage')
102         self.assertEqual(page.__parent__, context)
```

```

104 class EditPageTests(unittest.TestCase):
105     def _callFUT(self, context, request):
106         from .views import edit_page
107         return edit_page(context, request)
108
109     def test_it_notsubmitted(self):
110         context = testing.DummyResource()
111         request = testing.DummyRequest()
112         info = self._callFUT(context, request)
113         self.assertEqual(info['page'], context)
114         self.assertEqual(info['save_url'],
115                          request.resource_url(context, 'edit_page'))
116
117     def test_it_submitted(self):
118         context = testing.DummyResource()
119         request = testing.DummyRequest({'form.submitted':True,
120                                         'body':'Hello yo!'})
121         response = self._callFUT(context, request)
122         self.assertEqual(response.location, 'http://example.com/')
123         self.assertEqual(context.data, 'Hello yo!')
124
125 class FunctionalTests(unittest.TestCase):
126
127     viewer_login = '/login?login=viewer&password=viewer' \
128                  '&came_from=FrontPage&form.submitted=Login'
129     viewer_wrong_login = '/login?login=viewer&password=incorrect' \
130                        '&came_from=FrontPage&form.submitted=Login'
131     editor_login = '/login?login=editor&password=editor' \
132                  '&came_from=FrontPage&form.submitted=Login'
133
134     def setUp(self):
135         import tempfile
136         import os.path
137         from . import main
138         self.tmpdir = tempfile.mkdtemp()
139
140         dbpath = os.path.join( self.tmpdir, 'test.db')
141         uri = 'file://' + dbpath
142         settings = { 'zodbconn.uri' : uri ,
143                    'pyramid.includes': ['pyramid_zodbconn', 'pyramid_tm
144     ↪'] }
145
146         app = main({}, **settings)
147         self.db = app.registry._zodb_databases['']
148         from webtest import TestApp
149         self.testapp = TestApp(app)

```

```
149
150 def tearDown(self):
151     import shutil
152     self.db.close()
153     shutil.rmtree( self.tmpdir )
154
155 def test_root(self):
156     res = self.testapp.get('/', status=302)
157     self.assertEqual(res.location, 'http://localhost/FrontPage')
158
159 def test_FrontPage(self):
160     res = self.testapp.get('/FrontPage', status=200)
161     self.assertTrue(b'FrontPage' in res.body)
162
163 def test_unexisting_page(self):
164     res = self.testapp.get('/SomePage', status=404)
165     self.assertTrue(b'Not Found' in res.body)
166
167 def test_referrer_is_login(self):
168     res = self.testapp.get('/login', status=200)
169     self.assertTrue(b'name="came_from" value="/" in res.body)
170
171 def test_successful_log_in(self):
172     res = self.testapp.get( self.viewer_login, status=302)
173     self.assertEqual(res.location, 'http://localhost/FrontPage')
174
175 def test_failed_log_in(self):
176     res = self.testapp.get( self.viewer_wrong_login, status=200)
177     self.assertTrue(b'login' in res.body)
178
179 def test_logout_link_present_when_logged_in(self):
180     res = self.testapp.get( self.viewer_login, status=302)
181     res = self.testapp.get('/FrontPage', status=200)
182     self.assertTrue(b'Logout' in res.body)
183
184 def test_logout_link_not_present_after_logged_out(self):
185     res = self.testapp.get( self.viewer_login, status=302)
186     res = self.testapp.get('/FrontPage', status=200)
187     res = self.testapp.get('/logout', status=302)
188     self.assertTrue(b'Logout' not in res.body)
189
190 def test_anonymous_user_cannot_edit(self):
191     res = self.testapp.get('/FrontPage/edit_page', status=200)
192     self.assertTrue(b'Login' in res.body)
193
194 def test_anonymous_user_cannot_add(self):
```

```

195         res = self.testapp.get('/add_page/NewPage', status=200)
196         self.assertTrue(b'Login' in res.body)
197
198     def test_viewer_user_cannot_edit(self):
199         res = self.testapp.get(self.viewer_login, status=302)
200         res = self.testapp.get('/FrontPage/edit_page', status=200)
201         self.assertTrue(b'Login' in res.body)
202
203     def test_viewer_user_cannot_add(self):
204         res = self.testapp.get(self.viewer_login, status=302)
205         res = self.testapp.get('/add_page/NewPage', status=200)
206         self.assertTrue(b'Login' in res.body)
207
208     def test_editors_member_user_can_edit(self):
209         res = self.testapp.get(self.editor_login, status=302)
210         res = self.testapp.get('/FrontPage/edit_page', status=200)
211         self.assertTrue(b'Editing' in res.body)
212
213     def test_editors_member_user_can_add(self):
214         res = self.testapp.get(self.editor_login, status=302)
215         res = self.testapp.get('/add_page/NewPage', status=200)
216         self.assertTrue(b'Editing' in res.body)
217
218     def test_editors_member_user_can_view(self):
219         res = self.testapp.get(self.editor_login, status=302)
220         res = self.testapp.get('/FrontPage', status=200)
221         self.assertTrue(b'FrontPage' in res.body)

```

Running the tests

We can run these tests by using `py.test` similarly to how we did in *Run the tests*. Courtesy of the scaffold, our testing dependencies have already been satisfied and `py.test` and coverage have already been configured, so we can jump right to running tests.

On UNIX:

```
$ $VENV/bin/py.test -q
```

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\py.test -q
```

The expected result should look like the following:

```
.....
24 passed in 2.46 seconds
```

Distributing Your Application

Once your application works properly, you can create a “tarball” from it by using the `setup.py sdist` command. The following commands assume your current working directory is the `tutorial` package we’ve created and that the parent directory of the `tutorial` package is a virtual environment representing a Pyramid environment.

On UNIX:

```
$ $VENV/bin/python setup.py sdist
```

On Windows:

```
c:\pyramidtut> %VENV%\Scripts\python setup.py sdist
```

The output of such a command will be something like:


```
running sdist
# more output
creating dist
Creating tar archive
removing 'tutorial-0.0' (and everything under it)
```

Note that this command creates a tarball in the “dist” subdirectory named `tutorial-0.0.tar.gz`. You can send this file to your friends to show them your cool new application. They should be able to install it by pointing the `pip install .` command directly at it. Or you can upload it to PyPI and share it with the rest of the world, where it can be downloaded via `pip install` remotely like any other package people download from PyPI.

Running a Pyramid Application under `mod_wsgi`

`mod_wsgi` is an Apache module developed by Graham Dumpleton. It allows *WSGI* programs to be served using the Apache web server.

This guide will outline broad steps that can be used to get a Pyramid application running under Apache via `mod_wsgi`. This particular tutorial was developed under Apple's Mac OS X platform (Snow Leopard, on a 32-bit Mac), but the instructions should be largely the same for all systems, delta specific path information for commands and files.

 Unfortunately these instructions almost certainly won't work for deploying a Pyramid application on a Windows system using `mod_wsgi`. If you have experience with Pyramid and `mod_wsgi` on Windows systems, please help us document this experience by submitting documentation to the Pylons-devel maillist.

1. The tutorial assumes you have Apache already installed on your system. If you do not, install Apache 2.X for your platform in whatever manner makes sense.
2. It is also assumed that you have satisfied the *Requirements for Installing Packages*.
3. Once you have Apache installed, install `mod_wsgi`. Use the (excellent) installation instructions for your platform into your system's Apache installation.
4. Create a *virtual environment* which we'll use to install our application.

```
$ cd ~  
$ mkdir modwsgi  
$ cd modwsgi  
$ python3 -m venv env
```

5. Install Pyramid into the newly created virtual environment:

```
$ cd ~/modwsgi/env  
$ $ENV/bin/pip install "pyramid==1.7.6"
```

6. Create and install your Pyramid application. For the purposes of this tutorial, we'll just be using the `pyramid_starter` application as a baseline application. Substitute your existing Pyramid application as necessary if you already have one.


```
$ cd ~/modwsgi/env
$ $VENV/bin/pcreate -s starter myapp
$ cd myapp
$ $VENV/bin/pip install -e .
```

7. Within the virtual environment directory (`~/modwsgi/env`), create a script named `pyramid.wsgi`. Give it these contents:

```
from pyramid.paster import get_app, setup_logging
ini_path = '/Users/chrism/modwsgi/env/myapp/production.ini'
setup_logging(ini_path)
application = get_app(ini_path, 'main')
```

The first argument to `get_app` is the project configuration file name. It's best to use the `production.ini` file provided by your scaffold, as it contains settings appropriate for production. The second is the name of the section within the `.ini` file that should be loaded by `mod_wsgi`. The assignment to the name `application` is important: `mod_wsgi` requires finding such an assignment when it opens the file.

The call to `setup_logging` initializes the standard library's *logging* module to allow logging within your application. See *Logging Configuration*.

There is no need to make the `pyramid.wsgi` script executable. However, you'll need to make sure that *two* users have access to change into the `~/modwsgi/env` directory: your current user (mine is `chrism` and the user that Apache will run as often named `apache` or `httpd`). Make sure both of these users can “cd” into that directory.

8. Edit your Apache configuration and add some stuff. I happened to create a file named `/etc/apache2/other/modwsgi.conf` on my own system while installing Apache, so this stuff went in there.

```
# Use only 1 Python sub-interpreter. Multiple sub-interpreters
# play badly with C extensions. See
# http://stackoverflow.com/a/10558360/209039
WSGIApplicationGroup %{GLOBAL}
WSGIPassAuthorization On
WSGIDaemonProcess pyramid user=chrism group=staff threads=4 \
    python-path=/Users/chrism/modwsgi/env/lib/python2.7/site-packages
WSGIScriptAlias /myapp /Users/chrism/modwsgi/env/pyramid.wsgi

<Directory /Users/chrism/modwsgi/env>
    WSGIProcessGroup pyramid
```

```
Order allow,deny
Allow from all
</Directory>
```

9. Restart Apache

```
$ sudo /usr/sbin/apachectl restart
```

10. Visit `http://localhost/myapp` in a browser. You should see the sample application rendered in your browser.

`mod_wsgi` has many knobs and a great variety of deployment modes. This is just one representation of how you might use it to serve up a Pyramid application. See the `mod_wsgi` configuration documentation for more in-depth configuration information.

Narrative Documentation

Pyramid Introduction

Pyramid is a general, open source, Python web application development *framework*. Its primary goal is to make it easier for a Python developer to create web applications.

Frameworks vs. Libraries

A *framework* differs from a *library* in one very important way: library code is always *called* by code that you write, while a framework always *calls* code that you write. Using a set of libraries to create an application is usually easier than using a framework initially, because you can choose to cede control to library code you have not authored very selectively. But when you use a framework, you are required to cede a greater portion of control to code you have not authored: code that resides in the framework itself. You needn't use a framework at all to create a web application using Python. A rich set of libraries already exists for the platform. In practice, however, using a framework to create an application is often more practical than rolling your own via a set of libraries if the framework provides a set of facilities that fits your application requirements.

Pyramid attempts to follow these design and engineering principles:

Simplicity Pyramid takes a *“pay only for what you eat”* approach. You can get results even if you have only a partial understanding of Pyramid. It doesn’t force you to use any particular technology to produce an application, and we try to keep the core set of concepts that you need to understand to a minimum.

Minimalism Pyramid tries to solve only the fundamental problems of creating a web application: the mapping of URLs to code, templating, security, and serving static assets. We consider these to be the core activities that are common to nearly all web applications.

Documentation Pyramid’s minimalism means that it is easier for us to maintain complete and up-to-date documentation. It is our goal that no aspect of Pyramid is undocumented.

Speed Pyramid is designed to provide noticeably fast execution for common tasks such as templating and simple response generation.

Reliability Pyramid is developed conservatively and tested exhaustively. Where Pyramid source code is concerned, our motto is: “If it ain’t tested, it’s broke”.

Openness As with Python, the Pyramid software is distributed under a permissive open source license.

What makes Pyramid unique

Understandably, people don’t usually want to hear about squishy engineering principles; they want to hear about concrete stuff that solves their problems. With that in mind, what would make someone want to use Pyramid instead of one of the many other web frameworks available today? What makes Pyramid unique?

This is a hard question to answer because there are lots of excellent choices, and it’s actually quite hard to make a wrong choice, particularly in the Python web framework market. But one reasonable answer is this: you can write very small applications in Pyramid without needing to know a lot. “What?” you say. “That can’t possibly be a unique feature. Lots of other web frameworks let you do that!” Well, you’re right. But unlike many other systems, you can also write very large applications in Pyramid if you learn a little more about it. Pyramid will allow you to become productive quickly, and will grow with you. It won’t hold you back when your application is small, and it won’t get in your way when your application becomes large. “Well that’s fine,” you say. “Lots of other frameworks let me write large apps, too.” Absolutely. But other Python web frameworks don’t seamlessly let you do both. They seem to fall into two non-overlapping categories: frameworks for “small apps” and frameworks for “big apps”. The “small app” frameworks typically sacrifice “big app” features, and vice versa.

We don’t think it’s a universally reasonable suggestion to write “small apps” in a “small framework” and “big apps” in a “big framework”. You can’t really know to what size every application will eventually grow. We don’t really want to have to rewrite a previously small application in another framework when

it gets “too big”. We believe the current binary distinction between frameworks for small and large applications is just false. A well-designed framework should be able to be good at both. Pyramid strives to be that kind of framework.

To this end, Pyramid provides a set of features that combined are unique amongst Python web frameworks. Lots of other frameworks contain some combination of these features. Pyramid of course actually stole many of them from those other frameworks. But Pyramid is the only one that has all of them in one place, documented appropriately, and useful *à la carte* without necessarily paying for the entire banquet. These are detailed below.

Single-file applications

You can write a Pyramid application that lives entirely in one Python file, not unlike existing Python microframeworks. This is beneficial for one-off prototyping, bug reproduction, and very small applications. These applications are easy to understand because all the information about the application lives in a single place, and you can deploy them without needing to understand much about Python distributions and packaging. Pyramid isn’t really marketed as a microframework, but it allows you to do almost everything that frameworks that are marketed as “micro” offer in very similar ways.

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.response import Response

def hello_world(request):
    return Response('Hello %(name)s!' % request.matchdict)

if __name__ == '__main__':
    config = Configurator()
    config.add_route('hello', '/hello/{name}')
    config.add_view(hello_world, route_name='hello')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

See also:

See also *Creating Your First Pyramid Application*.

Decorator-based configuration

If you like the idea of framework configuration statements living next to the code it configures, so you don't have to constantly switch between files to refer to framework configuration when adding new code, you can use Pyramid decorators to localize the configuration. For example:

```
from pyramid.view import view_config
from pyramid.response import Response

@view_config(route_name='fred')
def fred_view(request):
    return Response('fred')
```

However, unlike some other systems, using decorators for Pyramid configuration does not make your application difficult to extend, test, or reuse. The `view_config` decorator, for example, does not actually *change* the input or output of the function it decorates, so testing it is a “WYSIWYG” operation. You don't need to understand the framework to test your own code. You just behave as if the decorator is not there. You can also instruct Pyramid to ignore some decorators, or use completely imperative configuration instead of decorators to add views. Pyramid decorators are inert instead of eager. You detect and activate them with a *scan*.

Example: *Adding View Configuration Using the @view_config Decorator.*

URL generation

Pyramid is capable of generating URLs for resources, routes, and static assets. Its URL generation APIs are easy to use and flexible. If you use Pyramid's various APIs for generating URLs, you can change your configuration around arbitrarily without fear of breaking a link on one of your web pages.

Example: *Generating Route URLs.*

Static file serving

Pyramid is perfectly willing to serve static files itself. It won't make you use some external web server to do that. You can even serve more than one set of static files in a single Pyramid web application (e.g., `/static` and `/static2`). You can optionally place your files on an external web server and ask Pyramid to help you generate URLs to those files. This lets you use Pyramid's internal file serving while doing development, and a faster static file server in production, without changing any code.

Example: *Serving Static Assets.*

Fully interactive development

When developing a Pyramid application, several interactive features are available. Pyramid can automatically utilize changed templates when rendering pages and automatically restart the application to incorporate changed Python code. Plain old `print()` calls used for debugging can display to a console.

Pyramid's debug toolbar comes activated when you use a Pyramid scaffold to render a project. This toolbar overlays your application in the browser, and allows you access to framework data, such as the routes configured, the last renderings performed, the current set of packages installed, SQLAlchemy queries run, logging data, and various other facts. When an exception occurs, you can use its interactive debugger to poke around right in your browser to try to determine the cause of the exception. It's handy.

Example: *The Debug Toolbar*.

Debugging settings

Pyramid has debugging settings that allow you to print Pyramid runtime information to the console when things aren't behaving as you're expecting. For example, you can turn on `debug_notfound`, which prints an informative message to the console every time a URL does not match any view. You can turn on `debug_authorization`, which lets you know why a view execution was allowed or denied by printing a message to the console. These features are useful for those WTF moments.

There are also a number of commands that you can invoke within a Pyramid environment that allow you to introspect the configuration of your system. `proutes` shows all configured routes for an application in the order they'll be evaluated for matching. `pviews` shows all configured views for any given URL. These are also WTF-crushers in some circumstances.

Examples: *Debugging View Authorization Failures* and *Command-Line Pyramid*.

Add-ons

Pyramid has an extensive set of add-ons held to the same quality standards as the Pyramid core itself. Add-ons are packages which provide functionality that the Pyramid core doesn't. Add-on packages already exist which let you easily send email, let you use the Jinja2 templating system, let you use XML-RPC or JSON-RPC, let you integrate with jQuery Mobile, etc.

Examples: <https://trypyramid.com/resources-extending-pyramid.html>

Class-based and function-based views

Pyramid has a structured, unified concept of a *view callable*. View callables can be functions, methods of classes, or even instances. When you add a new view callable, you can choose to make it a function or a method of a class. In either case Pyramid treats it largely the same way. You can change your mind later and move code between methods of classes and functions. A collection of similar view callables can be attached to a single class as methods, if that floats your boat, and they can share initialization code as necessary. All kinds of views are easy to understand and use, and operate similarly. There is no phony distinction between them. They can be used for the same purposes.

Here's a view callable defined as a function:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(route_name='aview')
5 def aview(request):
6     return Response('one')
```

Here's a few views defined as methods of a class instead:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 class AView(object):
5     def __init__(self, request):
6         self.request = request
7
8     @view_config(route_name='view_one')
9     def view_one(self):
10         return Response('one')
11
12     @view_config(route_name='view_two')
13     def view_two(self):
14         return Response('two')
```

See also:

See also *@view_config Placement*.

Asset specifications

Asset specifications are strings that contain both a Python package name and a file or directory name, e.g., `MyPackage:static/index.html`. Use of these specifications is omnipresent in Pyramid. An asset specification can refer to a template, a translation directory, or any other package-bound static resource. This makes a system built on Pyramid extensible because you don't have to rely on globals ("*the* static directory") or lookup schemes ("*the* ordered set of template directories") to address your files. You can move files around as necessary, and include other packages that may not share your system's templates or static files without encountering conflicts.

Because asset specifications are used heavily in Pyramid, we've also provided a way to allow users to override assets. Say you love a system that someone else has created with Pyramid but you just need to change "that one template" to make it all better. No need to fork the application. Just override the asset specification for that template with your own inside a wrapper, and you're good to go.

Examples: *Understanding Asset Specifications* and *Overriding Assets*.

Extensible templating

Pyramid has a structured API that allows for pluggability of "renderers". Templating systems such as Mako, Genshi, Chameleon, and Jinja2 can be treated as renderers. Renderer bindings for all of these templating systems already exist for use in Pyramid. But if you'd rather use another, it's not a big deal. Just copy the code from an existing renderer package, and plug in your favorite templating system. You'll then be able to use that templating system from within Pyramid just as you'd use one of the "built-in" templating systems.

Pyramid does not make you use a single templating system exclusively. You can use multiple templating systems, even in the same project.

Example: *Using Templates Directly*.

Rendered views can return dictionaries

If you use a *renderer*, you don't have to return a special kind of "webby" `Response` object from a view. Instead you can return a dictionary, and Pyramid will take care of converting that dictionary to a `Response` using a template on your behalf. This makes the view easier to test, because you don't have to parse HTML in your tests. Instead just make an assertion that the view returns "the right stuff" in the dictionary. You can write "real" unit tests instead of functionally testing all of your views.

For example, instead of returning a `Response` object from a `render_to_response` call:


```
1 from pyramid.renderers import render_to_response
2
3 def myview(request):
4     return render_to_response('myapp:templates/mytemplate.pt', {'a':1},
5                               request=request)
```

You can return a Python dictionary:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='myapp:templates/mytemplate.pt')
4 def myview(request):
5     return {'a':1}
```

When this view callable is called by Pyramid, the `{'a':1}` dictionary will be rendered to a response on your behalf. The string passed as `renderer=` above is an *asset specification*. It is in the form `packagename:directoryname/filename.ext`. In this case, it refers to the `mytemplate.pt` file in the `templates` directory within the `myapp` Python package. Asset specifications are omnipresent in Pyramid. See *Asset specifications* for more information.

Example: *Renderers*.

Event system

Pyramid emits *events* during its request processing lifecycle. You can subscribe any number of listeners to these events. For example, to be notified of a new request, you can subscribe to the `NewRequest` event. To be notified that a template is about to be rendered, you can subscribe to the `BeforeRender` event, and so forth. Using an event publishing system as a framework notification feature instead of hardcoded hook points tends to make systems based on that framework less brittle.

You can also use Pyramid's event system to send your *own* events. For example, if you'd like to create a system that is itself a framework, and may want to notify subscribers that a document has just been indexed, you can create your own event type (`DocumentIndexed` perhaps) and send the event via Pyramid. Users of this framework can then subscribe to your event like they'd subscribe to the events that are normally sent by Pyramid itself.

Example: *Using Events* and *Event Types*.

Built-in internationalization

Pyramid ships with internationalization-related features in its core: localization, pluralization, and creating message catalogs from source files and templates. Pyramid allows for a plurality of message catalogs via the use of translation domains. You can create a system that has its own translations without conflict with other translations in other domains.

Example: *Internationalization and Localization*.

HTTP caching

Pyramid provides an easy way to associate views with HTTP caching policies. You can just tell Pyramid to configure your view with an `http_cache` statement, and it will take care of the rest:

```
@view_config(http_cache=3600) # 60 minutes
def myview(request): ....
```

Pyramid will add appropriate `Cache-Control` and `Expires` headers to responses generated when this view is invoked.

See the `add_view()` method's `http_cache` documentation for more information.

Sessions

Pyramid has built-in HTTP sessioning. This allows you to associate data with otherwise anonymous users between requests. Lots of systems do this. But Pyramid also allows you to plug in your own sessioning system by creating some code that adheres to a documented interface. Currently there is a binding package for the third-party Redis sessioning system that does exactly this. But if you have a specialized need (perhaps you want to store your session data in MongoDB), you can. You can even switch between implementations without changing your application code.

Example: *Sessions*.

Speed

The Pyramid core is, as far as we can tell, at least marginally faster than any other existing Python web framework. It has been engineered from the ground up for speed. It only does as much work as absolutely necessary when you ask it to get a job done. Extraneous function calls and suboptimal algorithms in its core codepaths are avoided. It is feasible to get, for example, between 3500 and 4000 requests per second from a simple Pyramid view on commodity dual-core laptop hardware and an appropriate WSGI server (*mod_wsgi* or gunicorn). In any case, performance statistics are largely useless without requirements and goals, but if you need speed, Pyramid will almost certainly never be your application’s bottleneck; at least no more than Python will be a bottleneck.

Example: <http://blog.curiassolutions.com/pages/the-great-web-framework-shootout.html>

Exception views

Exceptions happen. Rather than deal with exceptions that might present themselves to a user in production in an ad-hoc way, Pyramid allows you to register an *exception view*. Exception views are like regular Pyramid views, but they’re only invoked when an exception “bubbles up” to Pyramid itself. For example, you might register an exception view for the `Exception` exception, which will catch *all* exceptions, and present a pretty “well, this is embarrassing” page. Or you might choose to register an exception view for only specific kinds of application-specific exceptions, such as an exception that happens when a file is not found, or an exception that happens when an action cannot be performed because the user doesn’t have permission to do something. In the former case, you can show a pretty “Not Found” page; in the latter case you might show a login form.

Example: *Custom Exception Views*.

No singletons

Pyramid is written in such a way that it requires your application to have exactly zero “singleton” data structures. Or put another way, Pyramid doesn’t require you to construct any “mutable globals”. Or put even another different way, an import of a Pyramid application needn’t have any “import-time side effects”. This is esoteric-sounding, but if you’ve ever tried to cope with parameterizing a Django `settings.py` file for multiple installations of the same application, or if you’ve ever needed to monkey-patch some framework fixture so that it behaves properly for your use case, or if you’ve ever wanted to deploy your system using an asynchronous server, you’ll end up appreciating this feature. It just won’t be a problem. You can even run multiple copies of a similar but not identically configured Pyramid application within the same Python process. This is good for shared hosting environments, where RAM is at a premium.

View predicates and many views per route

Unlike many other systems, Pyramid allows you to associate more than one view per route. For example, you can create a route with the pattern `/items` and when the route is matched, you can shuffle off the request to one view if the request method is GET, another view if the request method is POST, etc. A system known as “view predicates” allows for this. Request method matching is the most basic thing you can do with a view predicate. You can also associate views with other request parameters, such as the elements in the query string, the Accept header, whether the request is an XHR request or not, and lots of other things. This feature allows you to keep your individual views clean. They won’t need much conditional logic, so they’ll be easier to test.

Example: *View Configuration Parameters*.

Transaction management

Pyramid’s *scaffold* system renders projects that include a *transaction management* system, stolen from Zope. When you use this transaction management system, you cease being responsible for committing your data anymore. Instead Pyramid takes care of committing: it commits at the end of a request or aborts if there’s an exception. Why is that a good thing? Having a centralized place for transaction management is a great thing. If, instead of managing your transactions in a centralized place, you sprinkle `session.commit` calls in your application logic itself, you can wind up in a bad place. Wherever you manually commit data to your database, it’s likely that some of your other code is going to run *after* your commit. If that code goes on to do other important things after that commit, and an error happens in the later code, you can easily wind up with inconsistent data if you’re not extremely careful. Some data will have been written to the database that probably should not have. Having a centralized commit point saves you from needing to think about this; it’s great for lazy people who also care about data integrity. Either the request completes successfully, and all changes are committed, or it does not, and all changes are aborted.

Pyramid’s transaction management system allows you to synchronize commits between multiple databases. It also allows you to do things like conditionally send email if a transaction commits, but otherwise keep quiet.

Example: *SQLAlchemy + URL dispatch wiki tutorial* (note the lack of commit statements anywhere in application code).

Configuration conflict detection

When a system is small, it's reasonably easy to keep it all in your head. But when systems grow large, you may have hundreds or thousands of configuration statements which add a view, add a route, and so forth.

Pyramid's configuration system keeps track of your configuration statements. If you accidentally add two that are identical, or Pyramid can't make sense out of what it would mean to have both statements active at the same time, it will complain loudly at startup time. It's not dumb though. It will automatically resolve conflicting configuration statements on its own if you use the configuration `include()` system. "More local" statements are preferred over "less local" ones. This allows you to intelligently factor large systems into smaller ones.

Example: *Conflict Detection*.

Configuration extensibility

Unlike other systems, Pyramid provides a structured "include" mechanism (see `include()`) that allows you to combine applications from multiple Python packages. All the configuration statements that can be performed in your "main" Pyramid application can also be performed by included packages, including the addition of views, routes, subscribers, and even authentication and authorization policies. You can even extend or override an existing application by including another application's configuration in your own, overriding or adding new views and routes to it. This has the potential to allow you to create a big application out of many other smaller ones. For example, if you want to reuse an existing application that already has a bunch of routes, you can just use the `include` statement with a `route_prefix`. The new application will live within your application at an URL prefix. It's not a big deal, and requires little up-front engineering effort.

For example:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     config = Configurator()
5     config.include('pyramid_jinja2')
6     config.include('pyramid_exclog')
7     config.include('some.other.guys.package', route_prefix='/someotherguy')
```

See also:

See also *Including Configuration from External Sources* and *Rules for Building an Extensible Application*.

Flexible authentication and authorization

Pyramid includes a flexible, pluggable authentication and authorization system. No matter where your user data is stored, or what scheme you'd like to use to permit your users to access your data, you can use a predefined Pyramid plugpoint to plug in your custom authentication and authorization code. If you want to change these schemes later, you can just change it in one place rather than everywhere in your code. It also ships with prebuilt well-tested authentication and authorization schemes out of the box. But what if you don't want to use Pyramid's built-in system? You don't have to. You can just write your own bespoke security code as you would in any other system.

Example: *Enabling an Authorization Policy*.

Traversal

Traversal is a concept stolen from *Zope*. It allows you to create a tree of resources, each of which can be addressed by one or more URLs. Each of those resources can have one or more *views* associated with it. If your data isn't naturally treelike, or you're unwilling to create a treelike representation of your data, you aren't going to find traversal very useful. However, traversal is absolutely fantastic for sites that need to be arbitrarily extensible. It's a lot easier to add a node to a tree than it is to shoehorn a route into an ordered list of other routes, or to create another entire instance of an application to service a department and glue code to allow disparate apps to share data. It's a great fit for sites that naturally lend themselves to changing departmental hierarchies, such as content management systems and document management systems. Traversal also lends itself well to systems that require very granular security ("Bob can edit *this* document" as opposed to "Bob can edit documents").

Examples: *Hello Traversal World* and *Much Ado About Traversal*.

Tweens

Pyramid has a sort of internal WSGI-middleware-ish pipeline that can be hooked by arbitrary add-ons named "tweens". The debug toolbar is a "tween", and the `pyramid_tm` transaction manager is also. Tweens are more useful than WSGI *middleware* in some circumstances because they run in the context of Pyramid itself, meaning you have access to templates and other renderers, a "real" request object, and other niceties.

Example: *Registering Tweens*.

View response adapters

A lot is made of the aesthetics of what *kinds* of objects you’re allowed to return from view callables in various frameworks. In a previous section in this document, we showed you that, if you use a *renderer*, you can usually return a dictionary from a view callable instead of a full-on *Response* object. But some frameworks allow you to return strings or tuples from view callables. When frameworks allow for this, code looks slightly prettier, because fewer imports need to be done, and there is less code. For example, compare this:

```
1 def aview(request):
2     return "Hello world!"
```

To this:

```
1 from pyramid.response import Response
2
3 def aview(request):
4     return Response("Hello world!")
```

The former is “prettier”, right?

Out of the box, if you define the former view callable (the one that simply returns a string) in Pyramid, when it is executed, Pyramid will raise an exception. This is because “explicit is better than implicit”, in most cases, and by default Pyramid wants you to return a *Response* object from a view callable. This is because there’s usually a heck of a lot more to a response object than just its body. But if you’re the kind of person who values such aesthetics, we have an easy way to allow for this sort of thing:

```
1 from pyramid.config import Configurator
2 from pyramid.response import Response
3
4 def string_response_adapter(s):
5     response = Response(s)
6     response.content_type = 'text/html'
7     return response
8
9 if __name__ == '__main__':
10     config = Configurator()
11     config.add_response_adapter(string_response_adapter, basestring)
```

Do that once in your Pyramid application at startup. Now you can return strings from any of your view callables, e.g.:

```
1 def helloview(request):
2     return "Hello world!"
3
4 def goodbyeview(request):
5     return "Goodbye world!"
```

Oh noes! What if you want to indicate a custom content type? And a custom status code? No fear:

```
1 from pyramid.config import Configurator
2
3 def tuple_response_adapter(val):
4     status_int, content_type, body = val
5     response = Response(body)
6     response.content_type = content_type
7     response.status_int = status_int
8     return response
9
10 def string_response_adapter(body):
11     response = Response(body)
12     response.content_type = 'text/html'
13     response.status_int = 200
14     return response
15
16 if __name__ == '__main__':
17     config = Configurator()
18     config.add_response_adapter(string_response_adapter, basestring)
19     config.add_response_adapter(tuple_response_adapter, tuple)
```

Once this is done, both of these view callables will work:

```
1 def aview(request):
2     return "Hello world!"
3
4 def anotherview(request):
5     return (403, 'text/plain', "Forbidden")
```

Pyramid defaults to explicit behavior, because it's the most generally useful, but provides hooks that allow you to adapt the framework to localized aesthetic desires.

See also:

See also *Changing How Pyramid Treats View Responses*.

“Global” response object

“Constructing these response objects in my view callables is such a chore! And I’m way too lazy to register a response adapter, as per the prior section,” you say. Fine. Be that way:

```
1 def aview(request):
2     response = request.response
3     response.body = 'Hello world!'
4     response.content_type = 'text/plain'
5     return response
```

See also:

See also *Varying Attributes of Rendered Responses*.

Automating repetitive configuration

Does Pyramid’s configurator allow you to do something, but you’re a little adventurous and just want it a little less verbose? Or you’d like to offer up some handy configuration feature to other Pyramid users without requiring that we change Pyramid? You can extend Pyramid’s *Configurator* with your own directives. For example, let’s say you find yourself calling `pyramid.config.Configurator.add_view()` repetitively. Usually you can take the boring away by using existing shortcuts, but let’s say that this is a case where there is no such shortcut:

```
1 from pyramid.config import Configurator
2
3 config = Configurator()
4 config.add_route('xhr_route', '/xhr/{id}')
5 config.add_view('my.package.GET_view', route_name='xhr_route',
6                 xhr=True, permission='view', request_method='GET')
7 config.add_view('my.package.POST_view', route_name='xhr_route',
8                 xhr=True, permission='view', request_method='POST')
9 config.add_view('my.package.HEAD_view', route_name='xhr_route',
10                xhr=True, permission='view', request_method='HEAD')
```

Pretty tedious right? You can add a directive to the Pyramid configurator to automate some of the tedium away:

```

1 from pyramid.config import Configurator
2
3 def add_protected_xhr_views(config, module):
4     module = config.maybe_dotted(module)
5     for method in ('GET', 'POST', 'HEAD'):
6         view = getattr(module, 'xhr_%s_view' % method, None)
7         if view is not None:
8             config.add_view(view, route_name='xhr_route', xhr=True,
9                             permission='view', request_method=method)
10
11 config = Configurator()
12 config.add_directive('add_protected_xhr_views', add_protected_xhr_views)

```

Once that's done, you can call the directive you've just added as a method of the Configurator object:

```

1 config.add_route('xhr_route', '/xhr/{id}')
2 config.add_protected_xhr_views('my.package')

```

Your previously repetitive configuration lines have now morphed into one line.

You can share your configuration code with others this way, too, by packaging it up and calling `add_directive()` from within a function called when another user uses the `include()` method against your code.

See also:

See also *Adding Methods to the Configurator via add_directive*.

Programmatic introspection

If you're building a large system that other users may plug code into, it's useful to be able to get an enumeration of what code they plugged in *at application runtime*. For example, you might want to show them a set of tabs at the top of the screen based on an enumeration of views they registered.

This is possible using Pyramid's *introspector*.

Here's an example of using Pyramid's introspector from within a view callable:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='bar')
5 def show_current_route_pattern(request):
6     introspector = request.registry.introspector
7     route_name = request.matched_route.name
8     route_intr = introspector.get('routes', route_name)
9     return Response(str(route_intr['pattern']))
```

See also:

See also *Pyramid Configuration Introspection*.

Python 3 compatibility

Pyramid and most of its add-ons are Python 3 compatible. If you develop a Pyramid application today, you won't need to worry that five years from now you'll be backwatered because there are language features you'd like to use but your framework doesn't support newer Python versions.

Testing

Every release of Pyramid has 100% statement coverage via unit and integration tests, as measured by the coverage tool available on PyPI. It also has greater than 95% decision/condition coverage as measured by the `instrumental` tool available on PyPI. It is automatically tested by Travis, and Jenkins on Python 2.7, Python 3.3, Python 3.4, Python 3.5, PyPy, and PyPy3 after each commit to its GitHub repository. Official Pyramid add-ons are held to a similar testing standard. We still find bugs in Pyramid and its official add-ons, but we've noticed we find a lot more of them while working on other projects that don't have a good testing regime.

Travis: <https://travis-ci.org/Pylons/pyramid> Jenkins: <http://jenkins.pylonsproject.org/job/pyramid/>

Support

It's our goal that no Pyramid question go unanswered. Whether you ask a question on IRC, on the Pylons-discuss mailing list, or on StackOverflow, you're likely to get a reasonably prompt response. We don't tolerate "support trolls" or other people who seem to get their rocks off by berating fellow users in our various official support channels. We try to keep it well-lit and new-user-friendly.

Example: Visit <irc://freenode.net#pyramid> (the `#pyramid` channel on irc.freenode.net in an IRC client) or the pylons-discuss maillist at <https://groups.google.com/forum/#!forum/pylons-discuss>.

Documentation

It's a constant struggle, but we try to maintain a balance between completeness and new-user-friendliness in the official narrative Pyramid documentation (concrete suggestions for improvement are always appreciated, by the way). We also maintain a “cookbook” of recipes, which are usually demonstrations of common integration scenarios too specific to add to the official narrative docs. In any case, the Pyramid documentation is comprehensive.

Example: The Pyramid Community Cookbook.

What Is The Pylons Project?

Pyramid is a member of the collection of software published under the Pylons Project. Pylons software is written by a loose-knit community of contributors. The Pylons Project website includes details about how Pyramid relates to the Pylons Project.

Pyramid and Other Web Frameworks

The first release of Pyramid's predecessor (named `repoze.bfg`) was made in July of 2008. At the end of 2010, we changed the name of `repoze.bfg` to Pyramid. It was merged into the Pylons project as Pyramid in November of that year.

Pyramid was inspired by *Zope*, *Pylons* (version 1.0), and *Django*. As a result, Pyramid borrows several concepts and features from each, combining them into a unique web framework.

Many features of Pyramid trace their origins back to *Zope*. Like *Zope* applications, Pyramid applications can be easily extended. If you obey certain constraints, the application you produce can be reused, modified, re-integrated, or extended by third-party developers without forking the original application. The concepts of *traversal* and declarative security in Pyramid were pioneered first in *Zope*.

The Pyramid concept of *URL dispatch* is inspired by the *Routes* system used by *Pylons* version 1.0. Like *Pylons* version 1.0, Pyramid is mostly policy-free. It makes no assertions about which database you should use. Pyramid no longer has built-in templating facilities as of version 1.5a2, but instead officially supports bindings for templating languages, including Chameleon, Jinja2, and Mako. In essence, it only supplies a mechanism to map URLs to *view* code, along with a set of conventions for calling those views. You are free to use third-party components that fit your needs in your applications.

The concept of *view* is used by Pyramid mostly as it would be by Django. Pyramid has a documentation culture more like Django's than like Zope's.

Like *Pylons* version 1.0, but unlike *Zope*, a Pyramid application developer may use completely imperative code to perform common framework configuration tasks such as adding a view or a route. In *Zope*, *ZCML* is typically required for similar purposes. In *Grok*, a *Zope*-based web framework, *decorator* objects and class-level declarations are used for this purpose. Out of the box, Pyramid supports imperative and decorator-based configuration. *ZCML* may be used via an add-on package named `pyramid_zcml`.

Also unlike *Zope* and other “full-stack” frameworks such as *Django*, Pyramid makes no assumptions about which persistence mechanisms you should use to build an application. *Zope* applications are typically reliant on *ZODB*. Pyramid allows you to build *ZODB* applications, but it has no reliance on the *ZODB* software. Likewise, *Django* tends to assume that you want to store your application’s data in a relational database. Pyramid makes no such assumption, allowing you to use a relational database, and neither encouraging nor discouraging the decision.

Other Python web frameworks advertise themselves as members of a class of web frameworks named model-view-controller frameworks. Insofar as this term has been claimed to represent a class of web frameworks, Pyramid also generally fits into this class.

You Say Pyramid is MVC, but Where’s the Controller?

The Pyramid authors believe that the MVC pattern just doesn’t really fit the web very well. In a Pyramid application, there is a resource tree which represents the site structure, and views which tend to present the data stored in the resource tree and a user-defined “domain model”. However, no facility provided by *the framework* actually necessarily maps to the concept of a “controller” or “model”. So if you had to give it some acronym, I guess you’d say Pyramid is actually an “RV” framework rather than an “MVC” framework. “MVC”, however, is close enough as a general classification moniker for purposes of comparison with other web frameworks.

Installing Pyramid



This installation guide emphasizes the use of Python 3.4 and greater for simplicity.

Before You Install Pyramid


Install Python version 3.4 or greater for your operating system, and satisfy the *Requirements for Installing Packages*, as described in the following sections.

Python Versions

As of this writing, Pyramid has been tested under Python 2.7, Python 3.3, Python 3.4, Python 3.5, PyPy, and PyPy3. Pyramid does not run under any version of Python before 2.7.

Pyramid is known to run on all popular UNIX-like systems such as Linux, Mac OS X, and FreeBSD, as well as on Windows platforms. It is also known to run on *PyPy* (1.9+).

Pyramid installation does not require the compilation of any C code. However, some Pyramid dependencies may attempt to build binary extensions from C code for performance speed ups. If a compiler or Python headers are unavailable, the dependency will fall back to using pure Python instead.

 If you see any warnings or errors related to failing to compile the binary extensions, in most cases you may safely ignore those errors. If you wish to use the binary extensions, please verify that you have a functioning compiler and the Python header files installed for your operating system.

For Mac OS X Users

Python comes pre-installed on Mac OS X, but due to Apple's release cycle, it is often out of date. Unless you have a need for a specific earlier version, it is recommended to install the latest 3.x version of Python.

You can install the latest version of Python for Mac OS X from the binaries on python.org.

Alternatively, you can use the *homebrew* package manager.

```
# for python 3.x
$ brew install python3
```

If you use an installer for your Python, then you can skip to the section *Installing Pyramid on a UNIX System*.

If You Don't Yet Have a Python Interpreter (UNIX)

If your system doesn't have a Python interpreter, and you're on UNIX, you can either install Python using your operating system's package manager *or* you can install Python from source fairly easily on any UNIX system that has development tools.

See also:

See the official Python documentation [Using Python on Unix platforms](#) for full details.

If You Don't Yet Have a Python Interpreter (Windows)

If your Windows system doesn't have a Python interpreter, you'll need to install it by downloading a Python 3.x-series interpreter executable from python.org's download section (the files labeled "Windows Installer"). Once you've downloaded it, double click on the executable, and select appropriate options during the installation process. To standardize this documentation, we used the GUI installer and selected the following options:

- **Screen 1: Install Python 3.x.x (32- or 64-bit)**
 - Check "Install launcher for all users (recommended)"
 - Check "Add Python 3.x to PATH"
 - Click "Customize installation"
- **Screen 2: Optional Features**
 - Check all options
 - Click "Next"
- **Screen 3: Advanced Options**
 - Check all options
 - Customize install location: "C:\Python3x", where "x" is the minor version of Python
 - Click "Next"

You might also need to download and install the Python for Windows extensions.

See also:

See the official Python documentation [Using Python on Windows](#) for full details.

See also:

Download and install the Python for Windows extensions. Carefully read the README.txt file at the end of the list of builds, and follow its directions. Make sure you get the proper 32- or 64-bit build and Python version.

See also:

Python launcher for Windows provides a command `py` that allows users to run any installed version of Python.



After you install Python on Windows, you might need to add the `c:\Python3x` directory to your environment's `Path`, where `x` is the minor version of installed Python, in order to make it possible to invoke Python from a command prompt by typing `python`. To do so, right click `My Computer`, select `Properties -> Advanced Tab -> Environment Variables`, and add that directory to the end of the `Path` environment variable.

See also:

See [Configuring Python \(on Windows\)](#) for full details.

Requirements for Installing Packages

Use *pip* for installing packages and `python3 -m venv env` for creating a virtual environment. A virtual environment is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.

See also:

See the Python Packaging Authority's (PyPA) documentation [Requirements for Installing Packages](#) for full details.

Installing Pyramid on a UNIX System

After installing Python as described previously in *For Mac OS X Users* or *If You Don't Yet Have a Python Interpreter (UNIX)*, and satisfying the *Requirements for Installing Packages*, you can now install Pyramid.

1. Make a *virtual environment* workspace:

```
$ export VENV=~/.env
$ python3 -m venv $VENV
```

You can either follow the use of the environment variable `$VENV`, or replace it with the root directory of the virtual environment. If you choose the former approach, ensure that `$VENV` is an absolute path. In the latter case, the `export` command can be skipped.

2. (Optional) Consider using `$VENV/bin/activate` to make your shell environment wired to use the virtual environment.

3. Use `pip` to get Pyramid and its direct dependencies installed:

```
$ $VENV/bin/pip install "pyramid==1.7.6"
```



Why use `$VENV/bin/pip` instead of `source bin/activate`, then `pip`?

`$VENV/bin/pip` clearly specifies that `pip` is run from within the virtual environment and not at the system level.

`activate` drops turds into the user's shell environment, leaving them vulnerable to executing commands in the wrong context. `deactivate` might not correctly restore previous shell environment variables.

Although using `source bin/activate`, then `pip`, requires fewer key strokes to issue commands once invoked, there are other things to consider. Michael F. Lamb (datagrok) presents a summary in Virtualenv's `bin/activate` is Doing It Wrong.

Ultimately we prefer to keep things clear and simple, so we use `$VENV/bin/pip`.

Installing Pyramid on a Windows System

After installing Python as described previously in *If You Don't Yet Have a Python Interpreter (Windows)*, and satisfying the *Requirements for Installing Packages*, you can now install Pyramid.

1. Make a *virtual environment* workspace:

```
c:\> set VENV=c:\env
# replace "x" with your minor version of Python 3
c:\> c:\Python3x\python -m venv %VENV%
c:\> cd %VENV%
```

You can either follow the use of the environment variable `%VENV%`, or replace it with the root directory of the virtual environment. If you choose the former approach, ensure that `%VENV%` is an absolute path. In the latter case, the `set` command can be skipped.

2. (Optional) Consider using `%VENV%\Scripts\activate.bat` to make your shell environment wired to use the virtual environment.
3. Use `pip` to get Pyramid and its direct dependencies installed:

```
c:\> %VENV%\Scripts\pip install "pyramid==1.7.6"
```



See the note above for *Why use `$VENV/bin/pip` instead of `source bin/activate`, then `pip`.*

What Gets Installed

When you install Pyramid, various libraries such as WebOb, PasteDeploy, and others are installed.

Additionally, as chronicled in *Creating a Pyramid Project*, scaffolds will be registered, which make it easy to start a new Pyramid project.

Creating Your First Pyramid Application

In this chapter, we will walk through the creation of a tiny Pyramid application. After we're finished creating the application, we'll explain in more detail how it works. It assumes you already have Pyramid installed. If you do not, head over to the *Installing Pyramid* section.

Hello World

Here's one of the very simplest Pyramid applications:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5
6 def hello_world(request):
7     return Response('Hello %(name)s!' % request.matchdict)
8
9 if __name__ == '__main__':
10     config = Configurator()
11     config.add_route('hello', '/hello/{name}')
12     config.add_view(hello_world, route_name='hello')
13     app = config.make_wsgi_app()
14     server = make_server('0.0.0.0', 8080, app)
15     server.serve_forever()
16
```

When this code is inserted into a Python script named `helloworld.py` and executed by a Python interpreter which has the Pyramid software installed, an HTTP server is started on TCP port 8080.

On UNIX:

```
$ $VENV/bin/python helloworld.py
```

On Windows:

```
c:\> %VENV%\Scripts\python helloworld.py
```

This command will not return and nothing will be printed to the console. When port 8080 is visited by a browser on the URL `/hello/world`, the server will simply serve up the text “Hello world!”. If your application is running on your local system, using `http://localhost:8080/hello/world` in a browser will show this result.

Each time you visit a URL served by the application in a browser, a logging line will be emitted to the console displaying the hostname, the date, the request method and path, and some additional information. This output is done by the `wsgiref` server we’ve used to serve this application. It logs an “access log” in Apache combined logging format to the console.

Press `Ctrl-C` (or `Ctrl-Break` on Windows) to stop the application.

Now that we have a rudimentary understanding of what the application does, let’s examine it piece by piece.

Imports

The above `helloworld.py` script uses the following set of import statements:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
```

The script imports the `Configurator` class from the `pyramid.config` module. An instance of the `Configurator` class is later used to configure your Pyramid application.

Like many other Python web frameworks, Pyramid uses the *WSGI* protocol to connect an application and a web server together. The `wsgiref` server is used in this example as a WSGI server for convenience, as it is shipped within the Python standard library.

The script also imports the `pyramid.response.Response` class for later use. An instance of this class will be used to create a web response.

View Callable Declarations

The above script, beneath its set of imports, defines a function named `hello_world`.

```
1 def hello_world(request):  
2     return Response('Hello %(name)s!' % request.matchdict)
```

The function accepts a single argument (`request`) and it returns an instance of the `pyramid.response.Response` class. The single argument to the class' constructor is a string computed from parameters matched from the URL. This value becomes the body of the response.

This function is known as a *view callable*. A view callable accepts a single argument, `request`. It is expected to return a *response* object. A view callable doesn't need to be a function; it can be represented via another type of object, like a class or an instance, but for our purposes here, a function serves us well.

A view callable is always called with a *request* object. A request object is a representation of an HTTP request sent to Pyramid via the active *WSGI* server.

A view callable is required to return a *response* object because a response object has all the information necessary to formulate an actual HTTP response; this object is then converted to text by the *WSGI* server which called Pyramid and it is sent back to the requesting browser. To return a response, each view callable creates an instance of the *Response* class. In the `hello_world` function, a string is passed as the body to the response.

Application Configuration

In the above script, the following code represents the *configuration* of this simple application. The application is configured using the previously defined imports and function definitions, placed within the confines of an `if` statement:

```
1 if __name__ == '__main__':  
2     config = Configurator()  
3     config.add_route('hello', '/hello/{name}')  
4     config.add_view(hello_world, route_name='hello')  
5     app = config.make_wsgi_app()  
6     server = make_server('0.0.0.0', 8080, app)  
7     server.serve_forever()
```

Let's break this down piece by piece.

Configurator Construction

```
1 if __name__ == '__main__':  
2     config = Configurator()
```

The `if __name__ == '__main__':` line in the code sample above represents a Python idiom: the code inside this `if` clause is not invoked unless the script containing this code is run directly from the operating system command line. For example, if the file named `helloworld.py` contains the entire script body, the code within the `if` statement will only be invoked when `python helloworld.py` is executed from the command line.

Using the `if` clause is necessary—or at least best practice—because code in a Python `.py` file may be eventually imported via the Python `import` statement by another `.py` file. `.py` files that are imported by other `.py` files are referred to as *modules*. By using the `if __name__ == '__main__':` idiom, the script above is indicating that it does not want the code within the `if` statement to execute if this module is imported from another; the code within the `if` block should only be run during a direct script execution.

The `config = Configurator()` line above creates an instance of the *Configurator* class. The resulting `config` object represents an API which the script uses to configure this particular Pyramid application. Methods called on the *Configurator* will cause registrations to be made in an *application registry* associated with the application.

Adding Configuration

```
1 config.add_route('hello', '/hello/{name}')
```

```
2 config.add_view(hello_world, route_name='hello')
```

The first line above calls the `pyramid.config.Configurator.add_route()` method, which registers a *route* to match any URL path that begins with `/hello/` followed by a string.

The second line registers the `hello_world` function as a *view callable* and makes sure that it will be called when the `hello` route is matched.

WSGI Application Creation

```
1 app = config.make_wsgi_app()
```

After configuring views and ending configuration, the script creates a WSGI *application* via the `pyramid.config.Configurator.make_wsgi_app()` method. A call to `make_wsgi_app` implies that all configuration is finished (meaning all method calls to the configurator, which sets up views and various other configuration settings, have been performed). The `make_wsgi_app` method returns a WSGI application object that can be used by any WSGI server to present an application to a requestor. WSGI is a protocol that allows servers to talk to Python applications. We don't discuss WSGI in any depth within this book, but you can learn more about it by reading its documentation.

The Pyramid application object, in particular, is an instance of a class representing a Pyramid *router*. It has a reference to the *application registry* which resulted from method calls to the configurator used to configure it. The *router* consults the registry to obey the policy choices made by a single application. These policy choices were informed by method calls to the *Configurator* made earlier; in our case, the only policy choices made were implied by calls to its `add_view` and `add_route` methods.

WSGI Application Serving

```
1 server = make_server('0.0.0.0', 8080, app)
2 server.serve_forever()
```

Finally, we actually serve the application to requestors by starting up a WSGI server. We happen to use the `wsgiref.make_server` server maker for this purpose. We pass in as the first argument `'0.0.0.0'`, which means “listen on all TCP interfaces”. By default, the HTTP server listens only on the `127.0.0.1` interface, which is problematic if you're running the server on a remote system and you wish to access it with a web browser from a local system. We also specify a TCP port number to listen on, which is 8080, passing it as the second argument. The final argument is the `app` object (a *router*), which is the application we wish to serve. Finally, we call the server's `serve_forever` method, which starts the main loop in which it will wait for requests from the outside world.

When this line is invoked, it causes the server to start listening on TCP port 8080. The server will serve requests forever, or at least until we stop it by killing the process which runs it (usually by pressing `Ctrl-C` or `Ctrl-Break` in the terminal we used to start it).

Conclusion

Our hello world application is one of the simplest possible Pyramid applications, configured “imperatively”. We can see that it's configured imperatively because the full power of Python is available to us as we perform configuration tasks.

References

For more information about the API of a *Configurator* object, see *Configurator*.

For more information about *view configuration*, see *View Configuration*.

Application Configuration

Most people already understand “configuration” as settings that influence the operation of an application. For instance, it’s easy to think of the values in a `.ini` file parsed at application startup time as “configuration”. However, if you’re reasonably open-minded, it’s easy to think of *code* as configuration too. Since Pyramid, like most other web application platforms, is a *framework*, it calls into code that you write (as opposed to a *library*, which is code that exists purely for you to call). The act of plugging application code that you’ve written into Pyramid is also referred to within this documentation as “configuration”; you are configuring Pyramid to call the code that makes up your application.

See also:

For information on `.ini` files for Pyramid applications see the *Startup* chapter.

There are two ways to configure a Pyramid application: *imperative configuration* and *declarative configuration*. Both are described below.

Imperative Configuration

“Imperative configuration” just means configuration done by Python statements, one after the next. Here’s one of the simplest Pyramid applications, configured imperatively:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 if __name__ == '__main__':
9     config = Configurator()
10    config.add_view(hello_world)
11    app = config.make_wsgi_app()
12    server = make_server('0.0.0.0', 8080, app)
13    server.serve_forever()
```

We won't talk much about what this application does yet. Just note that the configuration statements take place underneath the `if __name__ == '__main__':` stanza in the form of method calls on a *Configurator* object (e.g., `config.add_view(...)`). These statements take place one after the other, and are executed in order, so the full power of Python, including conditionals, can be employed in this mode of configuration.

Declarative Configuration

It's sometimes painful to have all configuration done by imperative code, because often the code for a single application may live in many files. If the configuration is centralized in one place, you'll need to have at least two files open at once to see the “big picture”: the file that represents the configuration, and the file that contains the implementation objects referenced by the configuration. To avoid this, Pyramid allows you to insert *configuration decoration* statements very close to code that is referred to by the declaration itself. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(name='hello', request_method='GET')
5 def hello(request):
6     return Response('Hello')
```

The mere existence of configuration decoration doesn't cause any configuration registration to be performed. Before it has any effect on the configuration of a Pyramid application, a configuration decoration within application code must be found through a process known as a *scan*.

For example, the `pyramid.view.view_config` decorator in the code example above adds an attribute to the `hello` function, making it available for a *scan* to find it later.

A *scan* of a *module* or a *package* and its subpackages for decorations happens when the `pyramid.config.Configurator.scan()` method is invoked: scanning implies searching for configuration declarations in a package and its subpackages. For example:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4 from pyramid.view import view_config
5
6 @view_config()
7 def hello(request):
8     return Response('Hello')
```



```
10 if __name__ == '__main__':
11     config = Configurator()
12     config.scan()
13     app = config.make_wsgi_app()
14     server = make_server('0.0.0.0', 8080, app)
15     server.serve_forever()
```

The scanning machinery imports each module and subpackage in a package or module recursively, looking for special attributes attached to objects defined within a module. These special attributes are typically attached to code via the use of a *decorator*. For example, the *view_config* decorator can be attached to a function or instance method.

Once scanning is invoked, and *configuration decoration* is found by the scanner, a set of calls are made to a *Configurator* on your behalf. These calls replace the need to add imperative configuration statements that don't live near the code being configured.

The combination of *configuration decoration* and the invocation of a *scan* is collectively known as *declarative configuration*.

In the example above, the scanner translates the arguments to *view_config* into a call to the *pyramid.config.Configurator.add_view()* method, effectively:

```
config.add_view(hello)
```

Summary

There are two ways to configure a Pyramid application: declaratively and imperatively. You can choose the mode with which you're most comfortable; both are completely equivalent. Examples in this documentation will use both modes interchangeably.

Creating a Pyramid Project

As we saw in *Creating Your First Pyramid Application*, it's possible to create a Pyramid application completely manually. However, it's usually more convenient to use a *scaffold* to generate a basic Pyramid project.

A project is a directory that contains at least one Python *package*. You'll use a scaffold to create a project, and you'll create your application logic within a package that lives inside the project. Even if your application is extremely simple, it is useful to place code that drives the application within a package,

because (1) a package is more easily extended with new code, and (2) an application that lives inside a package can also be distributed more easily than one which does not live within a package.

Pyramid comes with a variety of scaffolds that you can use to generate a project. Each scaffold makes different configuration assumptions about what type of application you're trying to construct.

These scaffolds are rendered using the `pcreate` command that is installed as part of Pyramid.

Scaffolds Included with Pyramid

The convenience scaffolds included with Pyramid differ from each other on a number of axes:

- the persistence mechanism they offer (no persistence mechanism, *ZODB*, or *SQLAlchemy*)
- the mechanism they use to map URLs to code (*traversal* or *URL dispatch*)

The included scaffolds are these:

starter URL mapping via *URL dispatch* and no persistence mechanism

zodb URL mapping via *traversal* and persistence via *ZODB*

alchemy URL mapping via *URL dispatch* and persistence via *SQLAlchemy*

Creating the Project

See also:

See also the output of `pcreate --help`.

In *Installing Pyramid*, you created a virtual Python environment via the `venv` command. To start a Pyramid *project*, use the `pcreate` command installed within the virtual environment. We'll choose the **starter** scaffold for this purpose. When we invoke `pcreate`, it will create a directory that represents our project.

In *Installing Pyramid* we called the virtual environment directory `env`. The following commands assume that our current working directory is the `env` directory.

The below example uses the `pcreate` command to create a project with the **starter** scaffold.

On UNIX:

```
$ $VENV/bin/pcreate -s starter MyProject
```

Or on Windows:

```
c:\> %VENV%\Scripts\pcreate -s starter MyProject
```

As a result of invoking the `pcreate` command, a directory named `MyProject` is created. That directory is a *project* directory. The `setup.py` file in that directory can be used to distribute your application, or install your application for deployment or development.

An `.ini` file named `development.ini` will be created in the project directory. You will use this `.ini` file to configure a server, to run your application, and to debug your application. It contains configuration that enables an interactive debugger and settings optimized for development.

Another `.ini` file named `production.ini` will also be created in the project directory. It contains configuration that disables any interactive debugger (to prevent inappropriate access and disclosure), and turns off a number of debugging settings. You can use this file to put your application into production.

The `MyProject` project directory contains an additional subdirectory named `myproject` (note the case difference) representing a Python *package* which holds very simple Pyramid sample code. This is where you'll edit your application's Python code and templates.

We created this project within an `env` virtual environment directory. However, note that this is not mandatory. The project directory can go more or less anywhere on your filesystem. You don't need to put it in a special "web server" directory, and you don't need to put it within a virtual environment directory. The author uses Linux mainly, and tends to put project directories which he creates within his `~/projects` directory. On Windows, it's a good idea to put project directories within a directory that contains no space characters, so it's wise to *avoid* a path that contains, i.e., `My Documents`. As a result, the author, when he uses Windows, just puts his projects in `C:\projects`.



You'll need to avoid using `pcreate` to create a project with the same name as a Python standard library component. In particular, this means you should avoid using the names `site` or `test`, both of which conflict with Python standard library packages. You should also avoid using the name `pyramid`, which will conflict with Pyramid itself.

Installing your Newly Created Project for Development

To install a newly created project for development, you should `cd` to the newly created project directory and use the Python interpreter from the *virtual environment* you created during *Installing Pyramid* to invoke the command `pip install -e .`, which installs the project in development mode (`-e` is for “editable”) into the current directory (`.`).

The file named `setup.py` will be in the root of the pcreate-generated project directory. The `python` you’re invoking should be the one that lives in the `bin` (or `Scripts` on Windows) directory of your virtual Python environment. Your terminal’s current working directory *must* be the newly created project directory.

On UNIX:

```
$ cd MyProject
$ $VENV/bin/pip install -e .
```

Or on Windows:

```
c:\> cd MyProject
c:\> %VENV%\Scripts\pip install -e .
```

Elided output from a run of this command on UNIX is shown below:

```
$ cd MyProject
$ $VENV/bin/pip install -e .
...
Successfully installed Chameleon-2.24 Mako-1.0.4 MyProject \
pyramid-chameleon-0.3 pyramid-debugtoolbar-2.4.2 pyramid-mako-1.0.2
```

This will install a *distribution* representing your project into the virtual environment interpreter’s library set so it can be found by `import` statements and by other console scripts such as `pserve`, `pshell`, `proutes`, and `pviews`.

Running the Tests for Your Application

To run unit tests for your application, you must first install the testing dependencies.

On UNIX:

```
$ $VENV/bin/pip install -e ".[testing]"
```

On Windows:

```
c:\> %VENV%\Scripts\pip install -e ".[testing]"
```

Once the testing requirements are installed, then you can run the tests using the `py.test` command that was just installed in the `bin` directory of your virtual environment.

On UNIX:

```
$ $VENV/bin/py.test -q
```


On Windows:

```
c:\> %VENV%\Scripts\py.test -q
```

Here's sample output from a test run on UNIX:

```
$ $VENV/bin/py.test -q
..
2 passed in 0.47 seconds
```

The tests themselves are found in the `tests.py` module in your `pcreate` generated project. Within a project generated by the `starter` scaffold, only two sample tests exist.

 The `-q` option is passed to the `py.test` command to limit the output to a stream of dots. If you don't pass `-q`, you'll see verbose test result output (which normally isn't very useful).

Alternatively, if you'd like to see test coverage, pass the `--cov` option to `py.test`:

```
$ $VENV/bin/py.test --cov -q
```

Scaffolds include configuration defaults for `py.test` and test coverage. These configuration files are `pytest.ini` and `.coveragerc`, located at the root of your package. Without these defaults, we would need to specify the path to the module on which we want to run tests and coverage.

```
$ $VENV/bin/py.test --cov=myproject myproject/tests.py -q
```

See also:

See `py.test`'s documentation for Usage and Invocations or invoke `py.test -h` to see its full set of options.

Running the Project Application**See also:**

See also the output of `pserve -help`.

Once a project is installed for development, you can run the application it represents using the `pserve` command against the generated configuration file. In our case, this file is named `development.ini`.

On UNIX:

```
$ $VENV/bin/pserve development.ini
```

On Windows:

```
c:\> %VENV%\Scripts\pserve development.ini
```

Here's sample output from a run of `pserve` on UNIX:

```
$ $VENV/bin/pserve development.ini
Starting server in PID 16208.
serving on http://127.0.0.1:6543
```

Access is restricted such that only a browser running on the same machine as Pyramid will be able to access your Pyramid application. However, if you want to open access to other machines on the same network, then edit the `development.ini` file, and replace the `host` value in the `[server:main]` section, changing it from `127.0.0.1` to `0.0.0.0`. For example:

```
[server:main]
use = egg:waitress#main
host = 0.0.0.0
port = 6543
```

Now when you use `pserve` to start the application, it will respond to requests on *all* IP addresses possessed by your system, not just requests to `localhost`. This is what the `0.0.0.0` in `serving on http://0.0.0.0:6543` means. The server will respond to requests made to `127.0.0.1` and on any external IP address. For example, your system might be configured to have an external IP address `192.168.1.50`. If that's the case, if you use a browser running on the same system as Pyramid, it will be able to access the application via `http://127.0.0.1:6543/` as well as via `http://192.168.1.50:6543/`. However, *other people* on other computers on the same network will also be able to visit your Pyramid application in their browser by visiting `http://192.168.1.50:6543/`.

You can change the port on which the server runs on by changing the same portion of the `development.ini` file. For example, you can change the `port = 6543` line in the `development.ini` file's `[server:main]` section to `port = 8080` to run the server on port 8080 instead of port 6543.

You can shut down a server started this way by pressing `Ctrl-C` (or `Ctrl-Break` on Windows).

The default server used to run your Pyramid application when a project is created from a scaffold is named *Waitress*. This server is what prints the `serving on...` line when you run `pserve`. It's a good idea to use this server during development because it's very simple. It can also be used for light production. Setting your application up under a different server is not advised until you've done some development work under the default server, particularly if you're not yet experienced with Python web development. Python web server setup can be complex, and you should get some confidence that your application works in a default environment before trying to optimize it or make it "more like production". It's awfully easy to get sidetracked trying to set up a non-default server for hours without actually starting to do any development. One of the nice things about Python web servers is that they're largely interchangeable, so if your application works under the default server, it will almost certainly work under any other server in production if you eventually choose to use a different one. Don't worry about it right now.

For more detailed information about the startup process, see *Startup*. For more information about environment variables and configuration file settings that influence startup and runtime behavior, see *Environment Variables and .ini File Settings*.

Reloading Code

During development, it's often useful to run `pserve` using its `--reload` option. When `--reload` is passed to `pserve`, changes to any Python module your project uses will cause the server to restart. This typically makes development easier, as changes to Python code made within a Pyramid application is not put into effect until the server restarts.

For example, on UNIX:

```
$ $VENV/bin/pserve development.ini --reload
Starting subprocess with file monitor
Starting server in PID 16601.
serving on http://127.0.0.1:6543
```

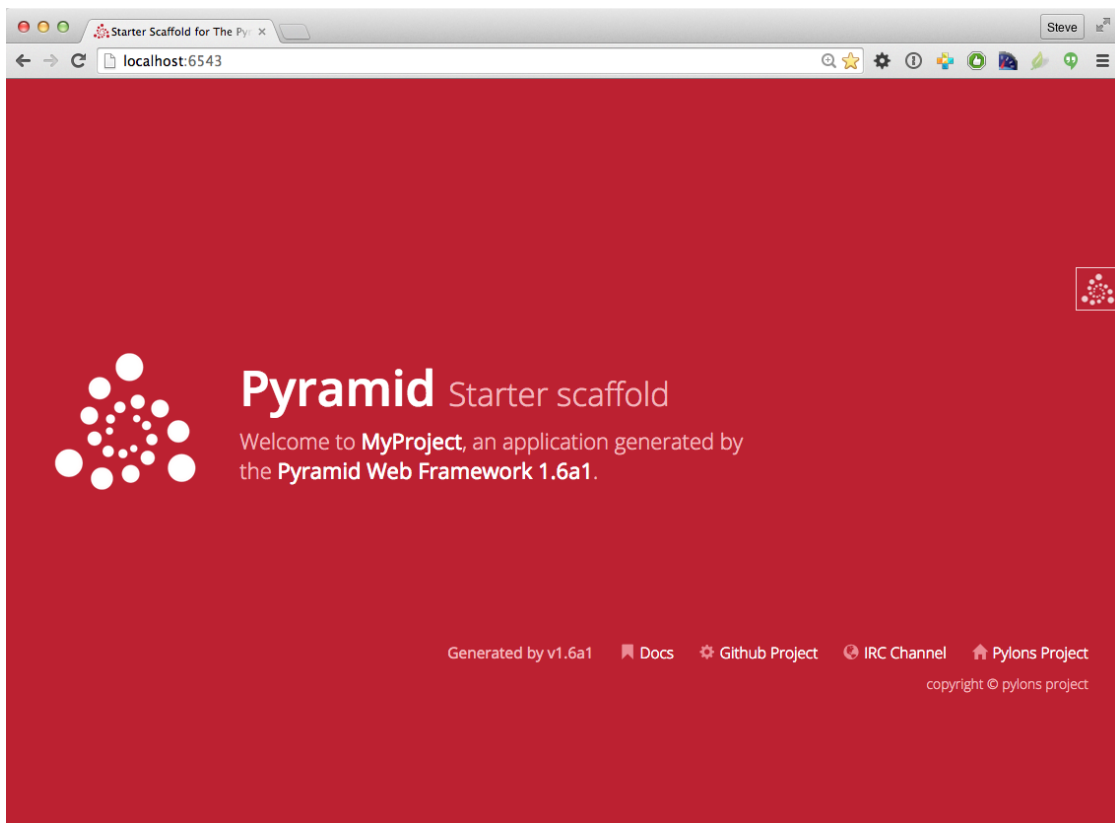
Now if you make a change to any of your project's .py files or .ini files, you'll see the server restart automatically:

```
development.ini changed; reloading...
----- Restarting -----
Starting server in PID 16602.
serving on http://127.0.0.1:6543
```

Changes to template files (such as .pt or .mak files) won't cause the server to restart. Changes to template files don't require a server restart as long as the `pyramid.reload_templates` setting in the `development.ini` file is `true`. Changes made to template files when this setting is `true` will take effect immediately without a server restart.

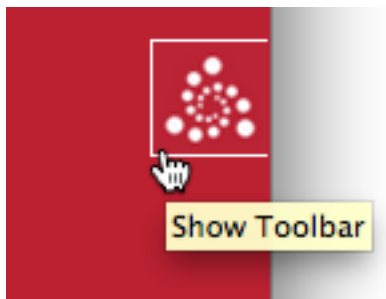
Viewing the Application

Once your application is running via `pserve`, you may visit `http://localhost:6543/` in your browser. You will see something in your browser like what is displayed in the following image:



This is the page shown by default when you visit an unmodified `pcreate` generated starter application in a browser.

The Debug Toolbar



If you click on the Pyramid logo at the top right of the page, a new target window will open to present a debug toolbar that provides various niceties while you're developing. This logo will float above every HTML page served by Pyramid while you develop an application, and allows you to show the toolbar as necessary.

The screenshot shows the Pyramid Debug Toolbar interface. The top bar is red with the Pyramid logo and navigation links: History, Global, and Settings. The left sidebar, titled 'Requests', lists several GET requests to localhost:6543, all with a 200 status. The main panel displays the 'Renderers' tab, showing details for the 'templates/mytemplate.pt' renderer. The 'Rendering Value' is {'project': 'MyProject'}. The 'System Values' section lists various attributes: context (a pyramid.traversal.DefaultRootFactory instance), renderer_info (a pyramid.renderers.RendererHelper object), renderer_name ('templates/mytemplate.pt'), req (a Request object), request (a Request object), and view (a function my_view).

If you don't see the Pyramid logo on the top right of the page, it means you're browsing from a system that does not have debugging access. By default, for security reasons, only a browser originating from localhost (127.0.0.1) can see the debug toolbar. To allow your browser on a remote system to access the server, add a line within the `[app:main]` section of the `development.ini` file in the form `debugtoolbar.hosts = X.X.X.X`. For example, if your Pyramid application is running on a remote system, and you're browsing from a host with the IP address 192.168.1.1, you'd add something like this to enable the toolbar when your system contacts Pyramid:

```
[app:main]
# .. other settings ...
debugtoolbar.hosts = 192.168.1.1
```

For more information about what the debug toolbar allows you to do, see the documentation for `pyramid_debugtoolbar`.

The debug toolbar will not be shown (and all debugging will be turned off) when you use the `production.ini` file instead of the `development.ini` file to run the application.

You can also turn the debug toolbar off by editing `development.ini` and commenting out a line. For example, instead of:

```
1 [app:main]
2 # ... elided configuration
3 pyramid.includes =
4     pyramid_debugtoolbar
```

Put a hash mark at the beginning of the `pyramid_debugtoolbar` line:

```
1 [app:main]
2 # ... elided configuration
3 pyramid.includes =
4 #     pyramid_debugtoolbar
```

Then restart the application to see that the toolbar has been turned off.

Note that if you comment out the `pyramid_debugtoolbar` line, the `#` *must* be in the first column. If you put it anywhere else, and then attempt to restart the application, you'll receive an error that ends something like this:

```
ImportError: No module named #pyramid_debugtoolbar
```

The Project Structure

The starter scaffold generated a *project* (named `MyProject`), which contains a Python *package*. The package is *also* named `myproject`, but it's lowercased; the scaffold generates a project which contains a package that shares its name except for case.

All Pyramid `pcreate`-generated projects share a similar structure. The `MyProject` project we've generated has the following directory structure:

```
MyProject/
|-- CHANGES.txt
|-- development.ini
|-- MANIFEST.in
|-- myproject
|   |-- __init__.py
|   |-- static
|       |-- pyramid-16x16.png
|       |-- pyramid.png
|       |-- theme.css
|       |-- theme.min.css
|   |-- templates
|       |-- mytemplate.pt
|   |-- tests.py
|   |-- views.py
|-- production.ini
|-- README.txt
`-- setup.py
```

The MyProject Project

The `MyProject` *project* directory is the distribution and deployment wrapper for your application. It contains both the `myproject` *package* representing your application as well as files used to describe, run, and test your application.

1. `CHANGES.txt` describes the changes you’ve made to the application. It is conventionally written in *ReStructuredText* format.
2. `README.txt` describes the application in general. It is conventionally written in *ReStructuredText* format.
3. `development.ini` is a *PasteDeploy* configuration file that can be used to execute your application during development.
4. `production.ini` is a *PasteDeploy* configuration file that can be used to execute your application in a production configuration.
5. `MANIFEST.in` is a *distutils* “manifest” file, naming which files should be included in a source distribution of the package when `python setup.py sdist` is run.
6. `setup.py` is the file you’ll use to test and distribute your application. It is a standard *setuptools* `setup.py` file.

development.ini

The `development.ini` file is a *PasteDeploy* configuration file. Its purpose is to specify an application to run when you invoke `pserve`, as well as the deployment settings provided to that application.

The generated `development.ini` file looks like so:

```
1  ###
2  # app configuration
3  # http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
   ↪environment.html
4  ###
5
6  [app:main]
7  use = egg:MyProject
8
9  pyramid.reload_templates = true
10 pyramid.debug_authorization = false
11 pyramid.debug_notfound = false
12 pyramid.debug_routematch = false
13 pyramid.default_locale_name = en
14 pyramid.includes =
15     pyramid_debugtoolbar
16
17 # By default, the toolbar only appears for clients from IP addresses
18 # '127.0.0.1' and ':::1'.
19 # debugtoolbar.hosts = 127.0.0.1 :::1
20
21 ###
22 # wsgi server configuration
23 ###
24
25 [server:main]
26 use = egg:waitress#main
27 host = 127.0.0.1
28 port = 6543
29
30 ###
31 # logging configuration
32 # http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
   ↪logging.html
33 ###
34
35 [loggers]
36 keys = root, myproject
37
```

```

38 [handlers]
39 keys = console
40
41 [formatters]
42 keys = generic
43
44 [logger_root]
45 level = INFO
46 handlers = console
47
48 [logger_myproject]
49 level = DEBUG
50 handlers =
51 qualname = myproject
52
53 [handler_console]
54 class = StreamHandler
55 args = (sys.stderr,)
56 level = NOTSET
57 formatter = generic
58
59 [formatter_generic]
60 format = %(asctime)s %(levelname)-5.5s [% (name)s: %(lineno)s] [
    ↪ %(threadName)s] %(message)s

```

This file contains several sections including `[app:main]`, `[server:main]`, and several other sections related to logging configuration.

The `[app:main]` section represents configuration for your Pyramid application. The `use` setting is the only setting required to be present in the `[app:main]` section. Its default value, `egg:MyProject`, indicates that our `MyProject` project contains the application that should be served. Other settings added to this section are passed as keyword arguments to the function named `main` in our package's `__init__.py` module. You can provide startup-time configuration parameters to your application by adding more settings to this section.

See also:

See *Entry Points and PasteDeploy .ini Files* for more information about the meaning of the `use = egg:MyProject` value in this section.

The `pyramid.reload_templates` setting in the `[app:main]` section is a Pyramid-specific setting which is passed into the framework. If it exists, and its value is `true`, supported template changes will not require an application restart to be detected. See *Automatically Reloading Templates* for more information.



The `pyramid.reload_templates` option should be turned off for production applications, as template rendering is slowed when it is turned on.

The `pyramid.includes` setting in the `[app:main]` section tells Pyramid to “include” configuration from another package. In this case, the line `pyramid.includes = pyramid_debugtoolbar` tells Pyramid to include configuration from the `pyramid_debugtoolbar` package. This turns on a debugging panel in development mode which can be opened by clicking on the Pyramid logo on the top right of the screen. Including the debug toolbar will also make it possible to interactively debug exceptions when an error occurs.

Various other settings may exist in this section having to do with debugging or influencing runtime behavior of a Pyramid application. See *Environment Variables and .ini File Settings* for more information about these settings.

The name `main` in `[app:main]` signifies that this is the default application run by `pserve` when it is invoked against this configuration file. The name `main` is a convention used by `PasteDeploy` signifying that it is the default application.

The `[server:main]` section of the configuration file configures a WSGI server which listens on TCP port 6543. It is configured to listen on localhost only (127.0.0.1). The sections after `# logging configuration` represent Python’s standard library logging module configuration for your application. These sections are passed to the logging module’s config file configuration engine when the `pserve` or `pshell` commands are executed. The default configuration sends application logging output to the standard error output of your terminal. For more information about logging configuration, see *Logging*.

See the *PasteDeploy* documentation for more information about other types of things you can put into this `.ini` file, such as other applications, *middleware*, and alternate *WSGI* server implementations.

production.ini

The `production.ini` file is a *PasteDeploy* configuration file with a purpose much like that of `development.ini`. However, it disables the debug toolbar, and filters all log messages except those above the `WARN` level. It also turns off template development options such that templates are not automatically reloaded when changed, and turns off all debugging options. This file is appropriate to use instead of `development.ini` when you put your application into production.

It’s important to use `production.ini` (and *not* `development.ini`) to benchmark your application and put it into production. `development.ini` configures your system with a debug toolbar that helps development, but the inclusion of this toolbar slows down page rendering times by over an order of magnitude. The debug toolbar is also a potential security risk if you have it configured incorrectly.

MANIFEST.in

The `MANIFEST.in` file is a *distutils* configuration file which specifies the non-Python files that should be included when a *distribution* of your Pyramid project is created when you run `python setup.py sdist`. Due to the information contained in the default `MANIFEST.in`, an sdist of your Pyramid project will include `.txt` files, `.ini` files, `.rst` files, graphics files, and template files, as well as `.py` files. See <https://docs.python.org/2/distutils/sourcedist.html#the-manifest-in-template> for more information about the syntax and usage of `MANIFEST.in`.

Without the presence of a `MANIFEST.in` file or without checking your source code into a version control repository, `setup.py sdist` places only *Python source files* (files ending with a `.py` extension) into tarballs generated by `python setup.py sdist`. This means, for example, if your project was not checked into a `setuptools`-compatible source control system, and your project directory didn't contain a `MANIFEST.in` file that told the sdist machinery to include `*.pt` files, the `myproject/templates/mytemplate.pt` file would not be included in the generated tarball.

Projects generated by Pyramid scaffolds include a default `MANIFEST.in` file. The `MANIFEST.in` file contains declarations which tell it to include files like `*.pt`, `*.css` and `*.js` in the generated tarball. If you include files with extensions other than the files named in the project's `MANIFEST.in` and you don't make use of a `setuptools`-compatible version control system, you'll need to edit the `MANIFEST.in` file and include the statements necessary to include your new files. See <https://docs.python.org/2/distutils/sourcedist.html#principle> for more information about how to do this.

You can also delete `MANIFEST.in` from your project and rely on a `setuptools` feature which simply causes all files checked into a version control system to be put into the generated tarball. To allow this to happen, check all the files that you'd like to be distributed along with your application's Python files into Subversion. After you do this, when you rerun `setup.py sdist`, all files checked into the version control system will be included in the tarball. If you don't use Subversion, and instead use a different version control system, you may need to install a `setuptools` add-on such as `setuptools-git` or `setuptools-hg` for this behavior to work properly.

setup.py

The `setup.py` file is a *setuptools* setup file. It is meant to be used to define requirements for installing dependencies for your package and testing, as well as distributing your application.



`setup.py` is the de facto standard which Python developers use to distribute their reusable code. You can read more about `setup.py` files and their usage in the Python Packaging User Guide and `Setuptools` documentation.

Our generated `setup.py` looks like this:


```
1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 with open(os.path.join(here, 'README.txt')) as f:
7     README = f.read()
8 with open(os.path.join(here, 'CHANGES.txt')) as f:
9     CHANGES = f.read()
10
11 requires = [
12     'pyramid',
13     'pyramid_chameleon',
14     'pyramid_debugtoolbar',
15     'waitress',
16 ]
17
18 tests_require = [
19     'WebTest >= 1.3.1', # py3 compat
20     'pytest', # includes virtualenv
21     'pytest-cov',
22 ]
23
24 setup(name='MyProject',
25       version='0.0',
26       description='MyProject',
27       long_description=README + '\n\n' + CHANGES,
28       classifiers=[
29         "Programming Language :: Python",
30         "Framework :: Pyramid",
31         "Topic :: Internet :: WWW/HTTP",
32         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
33     ],
34     author='',
35     author_email='',
36     url='',
37     keywords='web pyramid pylons',
38     packages=find_packages(),
39     include_package_data=True,
40     zip_safe=False,
41     extras_require={
42         'testing': tests_require,
43     },
44     install_requires=requires,
45     entry_points="""\
46     [paste.app_factory]
```

```
47     main = myproject:main
48     """
49 )
```

The `setup.py` file calls the `setuptools` `setup` function, which does various things depending on the arguments passed to `pip` on the command line.

Within the arguments to this function call, information about your application is kept. While it's beyond the scope of this documentation to explain everything about `setuptools` setup files, we'll provide a whirlwind tour of what exists in this file in this section.

Your application's name can be any string; it is specified in the `name` field. The version number is specified in the `version` value. A short description is provided in the `description` field. The `long_description` is conventionally the content of the `README` and `CHANGES` files appended together. The `classifiers` field is a list of Trove classifiers describing your application. `author` and `author_email` are text fields which probably don't need any description. `url` is a field that should point at your application project's URL (if any). `packages=find_packages()` causes all packages within the project to be found when packaging the application. `include_package_data` will include non-Python files when the application is packaged if those files are checked into version control. `zip_safe=False` indicates that this package is not safe to use as a zipped egg; instead it will always unpack as a directory, which is more convenient. `install_requires` indicates that this package depends on the `pyramid` package. `extras_require` is a Python dictionary that defines what is required to be installed for running tests. We examined `entry_points` in our discussion of the `development.ini` file; this file defines the `main` entry point that represents our project's application.

Usually you only need to think about the contents of the `setup.py` file when distributing your application to other people, when adding Python package dependencies, or when versioning your application for your own use. For fun, you can try this command now:

```
$ $VENV/bin/python setup.py sdist
```

This will create a tarball of your application in a `dist` subdirectory named `MyProject-0.0.tar.gz`. You can send this tarball to other people who want to install and use your application.

The `myproject` Package

The `myproject` *package* lives inside the `MyProject` *project*. It contains:

1. An `__init__.py` file signifies that this is a Python *package*. It also contains code that helps users run the application, including a `main` function which is used as an entry point for commands such as `pserve`, `pshell`, `pviews`, and others.

2. A `templates` directory, which contains *Chameleon* (or other types of) templates.
3. A `tests.py` module, which contains unit test code for the application.
4. A `views.py` module, which contains view code for the application.

These are purely conventions established by the scaffold. Pyramid doesn't insist that you name things in any particular way. However, it's generally a good idea to follow Pyramid standards for naming, so that other Pyramid developers can get up to speed quickly on your code when you need help.

`__init__.py`

We need a small Python module that configures our application and which advertises an entry point for use by our *PasteDeploy* `.ini` file. This is the file named `__init__.py`. The presence of an `__init__.py` also informs Python that the directory which contains it is a *package*.

```
1 from pyramid.config import Configurator
2
3
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6         """
7     config = Configurator(settings=settings)
8     config.include('pyramid_chameleon')
9     config.add_static_view('static', 'static', cache_max_age=3600)
10    config.add_route('home', '/')
11    config.scan()
12    return config.make_wsgi_app()
```

1. Line 1 imports the *Configurator* class from `pyramid.config` that we use later.
2. Lines 4-12 define a function named `main` that returns a Pyramid WSGI application. This function is meant to be called by the *PasteDeploy* framework as a result of running `pserve`.

Within this function, application configuration is performed.

Line 7 creates an instance of a *Configurator*.

Line 8 adds support for Chameleon templating bindings, allowing us to specify renderers with the `.pt` extension.

Line 9 registers a static view, which will serve up the files from the `myproject:static` asset specification (the `static` directory of the `myproject` package).

Line 10 adds a *route* to the configuration. This route is later used by a view in the `views` module.

Line 11 calls `config.scan()`, which picks up view registrations declared elsewhere in the package (in this case, in the `views.py` module).

Line 12 returns a *WSGI* application to the caller of the function (Pyramid's `pserve`).

views.py

Much of the heavy lifting in a Pyramid application is done by *view callables*. A *view callable* is the main tool of a Pyramid web application developer; it is a bit of code which accepts a *request* and which returns a *response*.

```
1 from pyramid.view import view_config
2
3
4 @view_config(route_name='home', renderer='templates/mytemplate.pt')
5 def my_view(request):
6     return {'project': 'MyProject'}
```

Lines 4-6 define and register a *view callable* named `my_view`. The function named `my_view` is decorated with a `view_config` decorator (which is processed by the `config.scan()` line in our `__init__.py`). The `view_config` decorator asserts that this view be found when a *route* named `home` is matched. In our case, because our `__init__.py` maps the route named `home` to the URL pattern `/`, this route will match when a visitor visits the root URL. The `view_config` decorator also names a *renderer*, which in this case is a template that will be used to render the result of the view callable. This particular view declaration points at `templates/mytemplate.pt`, which is an *asset specification* that specifies the `mytemplate.pt` file within the `templates` directory of the `myproject` package. The asset specification could have also been specified as `myproject:templates/mytemplate.pt`; the leading package name and colon is optional. The template file pointed to is a *Chameleon ZPT* template file (`templates/my_template.pt`).

This view callable function is handed a single piece of information: the *request*. The *request* is an instance of the `WebOb Request` class representing the browser's request to our server.

This view is configured to invoke a *renderer* on a template. The dictionary the view returns (on line 6) provides the value the renderer substitutes into the template when generating HTML. The renderer then returns the HTML in a *response*.



Dictionaries provide values to *templates*.



When the application is run with the scaffold's *default development.ini* configuration, *logging is set up* to aid debugging. If an exception is raised, uncaught tracebacks are displayed after the startup messages on the *console running the server*. Also `print()` statements may be inserted into the application for debugging to send output to this console.



`development.ini` has a setting that controls how templates are reloaded, `pyramid.reload_templates`.

- When set to `True` (as in the scaffold `development.ini`), changed templates automatically reload without a server restart. This is convenient while developing, but slows template rendering speed.
 - When set to `False` (the default value), changing templates requires a server restart to reload them. Production applications should use `pyramid.reload_templates = False`.
-

See also:

See also *Writing View Callables Which Use a Renderer* for more information about how views, renderers, and templates relate and cooperate.

See also:

Pyramid can also dynamically reload changed Python files. See also *Reloading Code*.

See also:

See also the *The Debug Toolbar*, which provides interactive access to your application's internals and, should an exception occur, allows interactive access to traceback execution stack frames from the Python interpreter.

static

This directory contains static assets which support the `mytemplate.pt` template. It includes CSS and images.

templates/mytemplate.pt

This is the single *Chameleon* template that exists in the project. Its contents are too long to show here, but it displays a default page when rendered. It is referenced by the call to `@view_config` as the `renderer` of the `my_view` view callable in the `views.py` file. See *Writing View Callables Which Use a Renderer* for more information about renderers.

Templates are accessed and used by view configurations and sometimes by view functions themselves. See *Using Templates Directly* and *Templates Used as Renderers via Configuration*.

tests.py

The `tests.py` module includes unit tests for your application.

```

1  import unittest
2
3  from pyramid import testing
4
5
6  class ViewTests(unittest.TestCase):
7      def setUp(self):
8          self.config = testing.setUp()
9
10     def tearDown(self):
11         testing.tearDown()
12
13     def test_my_view(self):
14         from .views import my_view
15         request = testing.DummyRequest()
16         info = my_view(request)
17         self.assertEqual(info['project'], 'MyProject')
18
19
20 class FunctionalTests(unittest.TestCase):
21     def setUp(self):
22         from myproject import main
23         app = main({})
24         from webtest import TestApp
25         self.testapp = TestApp(app)
26
27     def test_root(self):
28         res = self.testapp.get('/', status=200)
29         self.assertTrue(b'Pyramid' in res.body)

```

This sample `tests.py` file has one unit test and one functional test defined within it. These tests are executed when you run `py.test -q`. You may add more tests here as you build your application. You are not required to write tests to use Pyramid. This file is simply provided for convenience and example.

See *Unit, Integration, and Functional Testing* for more information about writing Pyramid unit tests.

Modifying Package Structure

It is best practice for your application's code layout to not stray too much from accepted Pyramid scaffold defaults. If you refrain from changing things very much, other Pyramid coders will be able to more

quickly understand your application. However, the code layout choices made for you by a scaffold are in no way magical or required. Despite the choices made for you by any scaffold, you can decide to lay your code out any way you see fit.

For example, the configuration method named `add_view()` requires you to pass a *dotted Python name* or a direct object reference as the class or function to be used as a view. By default, the `starter` scaffold would have you add view functions to the `views.py` module in your package. However, you might be more comfortable creating a `views` *directory*, and adding a single file for each view.

If your project package name was `myproject` and you wanted to arrange all your views in a Python subpackage within the `myproject` *package* named `views` instead of within a single `views.py` file, you might do the following.

- Create a `views` directory inside your `myproject` package directory (the same directory which holds `views.py`).
- Create a file within the new `views` directory named `__init__.py`. (It can be empty. This just tells Python that the `views` directory is a *package*.)
- *Move* the content from the existing `views.py` file to a file inside the new `views` directory named, say, `blog.py`. Because the `templates` directory remains in the `myproject` package, the template *asset specification* values in `blog.py` must now be fully qualified with the project's package name (`myproject:templates/blog.pt`).

You can then continue to add view callable functions to the `blog.py` module, but you can also add other `.py` files which contain view callable functions to the `views` directory. As long as you use the `@view_config` directive to register views in conjunction with `config.scan()`, they will be picked up automatically when the application is restarted.

Using the Interactive Shell

It is possible to use the `pshell` command to load a Python interpreter prompt with a similar configuration as would be loaded if you were running your Pyramid application via `pserve`. This can be a useful debugging tool. See *The Interactive Shell* for more details.

What Is This `pserve` Thing

The code generated by a Pyramid scaffold assumes that you will be using the `pserve` command to start your application while you do development. `pserve` is a command that reads a *PasteDeploy* `.ini` file (e.g., `development.ini`), and configures a server to serve a Pyramid application based on the data in the file.

`pserve` is by no means the only way to start up and serve a Pyramid application. As we saw in *Creating Your First Pyramid Application*, `pserve` needn't be invoked at all to run a Pyramid application. The use of `pserve` to run a Pyramid application is purely conventional based on the output of its scaffolding. But we strongly recommend using `pserve` while developing your application because many other convenience introspection commands (such as `pviews`, `prequest`, `proutes`, and others) are also implemented in terms of configuration availability of this `.ini` file format. It also configures Pyramid logging and provides the `--reload` switch for convenient restarting of the server when code changes.

Using an Alternate WSGI Server

Pyramid scaffolds generate projects which use the *Waitress* WSGI server. *Waitress* is a server that is suited for development and light production usage. It's not the fastest nor the most featureful WSGI server. Instead, its main feature is that it works on all platforms that Pyramid needs to run on, making it a good choice as a default server from the perspective of Pyramid's developers.

Any WSGI server is capable of running a Pyramid application. But we suggest you stick with the default server for development, and that you wait to investigate other server options until you're ready to deploy your application to production. Unless for some reason you need to develop on a non-local system, investigating alternate server options is usually a distraction until you're ready to deploy. But we recommend developing using the default configuration on a local system that you have complete control over; it will provide the best development experience.

One popular production alternative to the default *Waitress* server is *mod_wsgi*. You can use `mod_wsgi` to serve your Pyramid application using the Apache web server rather than any "pure-Python" server like *Waitress*. It is fast and featureful. See *Running a Pyramid Application under mod_wsgi* for details.

Another good production alternative is *Green Unicorn* (aka `gunicorn`). It's faster than *Waitress* and slightly easier to configure than `mod_wsgi`, although it depends, in its default configuration, on having a buffering HTTP proxy in front of it. It does not, as of this writing, work on Windows.

Startup

When you cause a Pyramid application to start up in a console window, you'll see something much like this show up on the console:


```
$ $VENV/bin/pserve development.ini
Starting server in PID 16305.
serving on http://127.0.0.1:6543
```

This chapter explains what happens between the time you press the “Return” key on your keyboard after typing `pserve development.ini` and the time the line `serving on http://127.0.0.1:6543` is output to your console.

The Startup Process

The easiest and best-documented way to start and serve a Pyramid application is to use the `pserve` command against a *PasteDeploy* `.ini` file. This uses the `.ini` file to infer settings and starts a server listening on a port. For the purposes of this discussion, we’ll assume that you are using this command to run your Pyramid application.

Here’s a high-level time-ordered overview of what happens when you press `return` after running `pserve development.ini`.

1. The `pserve` command is invoked under your shell with the argument `development.ini`. As a result, Pyramid recognizes that it is meant to begin to run and serve an application using the information contained within the `development.ini` file.
2. The framework finds a section named either `[app:main]`, `[pipeline:main]`, or `[composite:main]` in the `.ini` file. This section represents the configuration of a *WSGI* application that will be served. If you’re using a simple application (e.g., `[app:main]`), the application’s `paste.app_factory` *entry point* will be named on the `use=` line within the section’s configuration. If instead of a simple application, you’re using a *WSGI pipeline* (e.g., a `[pipeline:main]` section), the application named on the “last” element will refer to your Pyramid application. If instead of a simple application or a pipeline, you’re using a “composite” (e.g., `[composite:main]`), refer to the documentation for that particular composite to understand how to make it refer to your Pyramid application. In most cases, a Pyramid application built from a scaffold will have a single `[app:main]` section in it, and this will be the application served.
3. The framework finds all logging related configuration in the `.ini` file and uses it to configure the Python standard library logging system for this application. See *Logging Configuration* for more information.

4. The application's *constructor* named by the entry point referenced on the `use=` line of the section representing your Pyramid application is passed the key/value parameters mentioned within the section in which it's defined. The constructor is meant to return a *router* instance, which is a *WSGI* application.

For Pyramid applications, the constructor will be a function named `main` in the `__init__.py` file within the *package* in which your application lives. If this function succeeds, it will return a *Pyramid router* instance. Here's the contents of an example `__init__.py` module:

```

1  from pyramid.config import Configurator
2
3
4  def main(global_config, **settings):
5      """ This function returns a Pyramid WSGI application.
6          """
7      config = Configurator(settings=settings)
8      config.include('pyramid_chameleon')
9      config.add_static_view('static', 'static', cache_max_age=3600)
10     config.add_route('home', '/')
11     config.scan()
12     return config.make_wsgi_app()
```

Note that the constructor function accepts a `global_config` argument, which is a dictionary of key/value pairs mentioned in the `[DEFAULT]` section of an `.ini` file (if `[DEFAULT]` is present). It also accepts a `**settings` argument, which collects another set of arbitrary key/value pairs. The arbitrary key/value pairs received by this function in `**settings` will be composed of all the key/value pairs that are present in the `[app:main]` section (except for the `use=` setting) when this function is called when you run `pserve`.

Our generated development `.ini` file looks like so:

```

1  ###
2  # app configuration
3  # http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
4  ↪environment.html
5  ###
6
7  [app:main]
8  use = egg:MyProject
9
10 pyramid.reload_templates = true
11 pyramid.debug_authorization = false
12 pyramid.debug_notfound = false
13 pyramid.debug_routematch = false
14 pyramid.default_locale_name = en
```

```
14 pyramid.includes =
15     pyramid_debugtoolbar
16
17 # By default, the toolbar only appears for clients from IP addresses
18 # '127.0.0.1' and '::1'.
19 # debugtoolbar.hosts = 127.0.0.1 ::1
20
21 ###
22 # wsgi server configuration
23 ###
24
25 [server:main]
26 use = egg:waitress#main
27 host = 127.0.0.1
28 port = 6543
29
30 ###
31 # logging configuration
32 # http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
33 ↪ logging.html
34 ###
35
36 [loggers]
37 keys = root, myproject
38
39 [handlers]
40 keys = console
41
42 [formatters]
43 keys = generic
44
45 [logger_root]
46 level = INFO
47 handlers = console
48
49 [logger_myproject]
50 level = DEBUG
51 handlers =
52 qualname = myproject
53
54 [handler_console]
55 class = StreamHandler
56 args = (sys.stderr,)
57 level = NOTSET
58 formatter = generic
```

```

59 [formatter_generic]
60 format = %(asctime)s %(levelname)-5.5s [% (name)s:% (lineno)s] [
    ↳ %(threadName)s] %(message)s

```

In this case, the `myproject.__init__:main` function referred to by the entry point URI `egg:MyProject` (see *development.ini* for more information about entry point URIs, and how they relate to callables) will receive the key/value pairs `{pyramid.reload_templates = true, pyramid.debug_authorization = false, pyramid.debug_notfound = false, pyramid.debug_routematch = false, pyramid.default_locale_name = en, and pyramid.includes = pyramid_debugtoolbar}`. See *Environment Variables and .ini File Settings* for the meanings of these keys.

5. The main function first constructs a *Configurator* instance, passing the settings dictionary captured via the `**settings` kwarg as its settings argument.

The settings dictionary contains all the options in the `[app:main]` section of our `.ini` file except the `use` option (which is internal to PasteDeploy) such as `pyramid.reload_templates`, `pyramid.debug_authorization`, etc.

6. The main function then calls various methods on the instance of the class *Configurator* created in the previous step. The intent of calling these methods is to populate an *application registry*, which represents the Pyramid configuration related to the application.
7. The `make_wsgi_app()` method is called. The result is a *router* instance. The router is associated with the *application registry* implied by the configurator previously populated by other methods run against the Configurator. The router is a WSGI application.
8. An *ApplicationCreated* event is emitted (see *Using Events* for more information about events).
9. Assuming there were no errors, the main function in `myproject` returns the router instance created by `pyramid.config.Configurator.make_wsgi_app()` back to `pserve`. As far as `pserve` is concerned, it is “just another WSGI application”.
10. `pserve` starts the WSGI *server* defined within the `[server:main]` section. In our case, this is the Waitress server (`use = egg:waitress#main`), and it will listen on all interfaces (`host = 127.0.0.1`), on port number 6543 (`port = 6543`). The server code itself is what prints serving on `http://127.0.0.1:6543`. The server serves the application, and the application is running, waiting to receive requests.

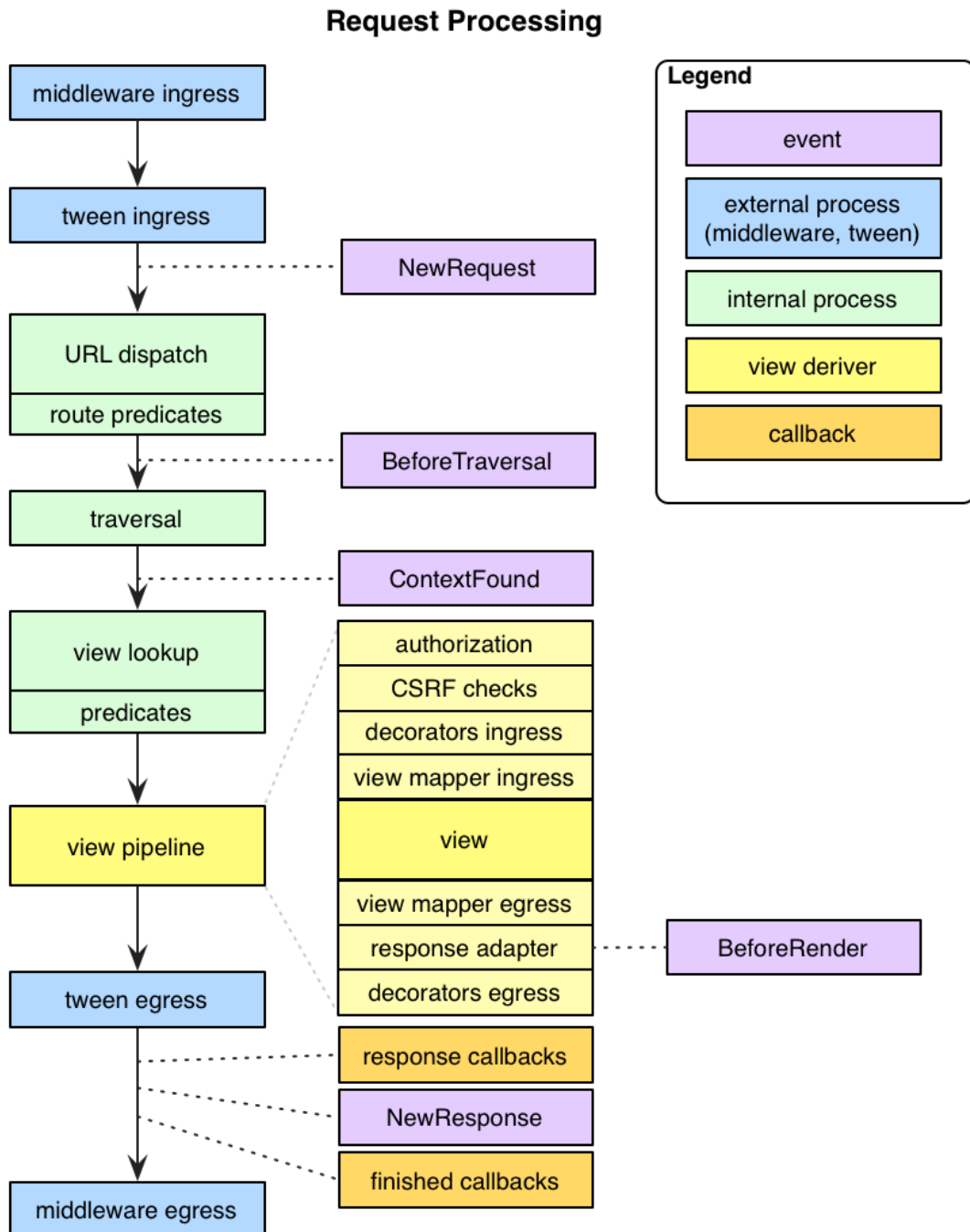
See also:

Logging configuration is described in the *Logging* chapter. There, in *Request Logging with Paste's TransLogger*, you will also find an example of how to configure *middleware* to add pre-packaged functionality to your application.

Deployment Settings

Note that an augmented version of the values passed as `**settings` to the *Configurator* constructor will be available in Pyramid *view callable* code as `request.registry.settings`. You can create objects you wish to access later from view code, and put them into the dictionary you pass to the configurator as `settings`. They will then be present in the `request.registry.settings` dictionary at application runtime.

Request Processing



Once a Pyramid application is up and running, it is ready to accept requests and return responses. What happens from the time a *WSGI* request enters a Pyramid application through to the point that Pyramid hands off a response back to *WSGI* for upstream processing?

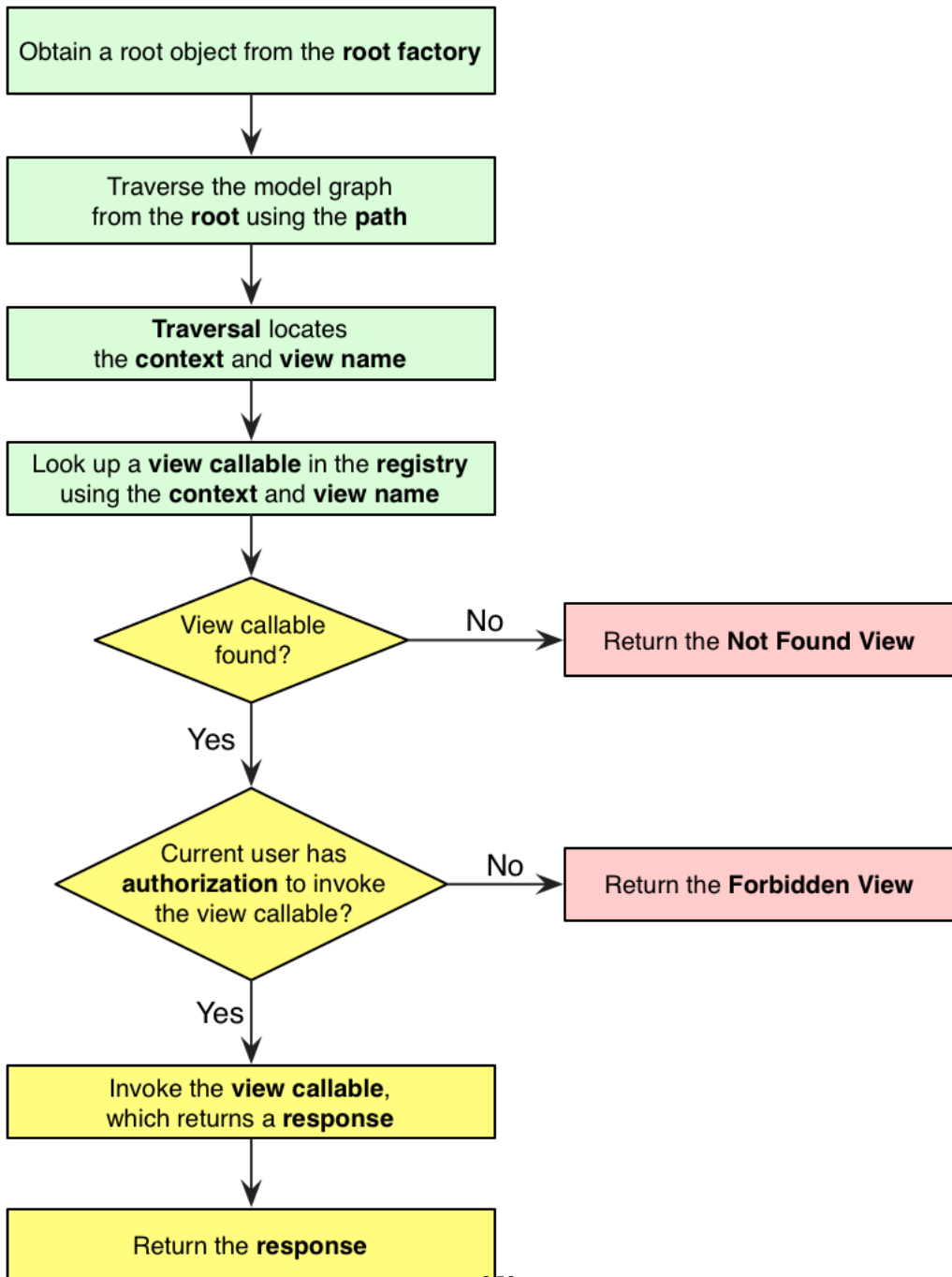
1. A user initiates a request from their browser to the hostname and port number of the *WSGI* server used by the Pyramid application.
2. The *WSGI* server used by the Pyramid application passes the *WSGI* environment to the `__call__` method of the Pyramid *router* object.
3. A *request* object is created based on the *WSGI* environment.
4. The *application registry* and the *request* object created in the last step are pushed on to the *thread local* stack that Pyramid uses to allow the functions named `get_current_request()` and `get_current_registry()` to work.
5. A *NewRequest event* is sent to any subscribers.
6. If any *route* has been defined within application configuration, the Pyramid *router* calls a *URL dispatch* “route mapper.” The job of the mapper is to examine the request to determine whether any user-defined *route* matches the current *WSGI* environment. The *router* passes the request as an argument to the mapper.
7. If any route matches, the route mapper adds the attributes `matchdict` and `matched_route` to the request object. The former contains a dictionary representing the matched dynamic elements of the request’s `PATH_INFO` value, and the latter contains the *IRoute* object representing the route which matched.
8. A *BeforeTraversal event* is sent to any subscribers.
9. Continuing, if any route matches, the root object associated with the found route is generated. If the *route configuration* which matched has an associated `factory` argument, then this factory is used to generate the root object; otherwise a default *root factory* is used.

However, if no route matches, and if a `root_factory` argument was passed to the *Configurator* constructor, that callable is used to generate the root object. If the `root_factory` argument passed to the *Configurator* constructor was `None`, a default root factory is used to generate a root object.

10. The Pyramid router calls a “traverser” function with the root object and the request. The traverser function attempts to traverse the root object (using any existing `__getitem__` on the root object and subobjects) to find a *context*. If the root object has no `__getitem__` method, the root itself is assumed to be the context. The exact traversal algorithm is described in *Traversal*. The traverser function returns a dictionary, which contains a *context* and a *view name* as well as other ancillary information.

11. The request is decorated with various names returned from the traverser (such as `context`, `view_name`, and so forth), so they can be accessed via, for example, `request.context` within *view* code.
12. A *ContextFound* event is sent to any subscribers.
13. Pyramid looks up a *view* callable using the context, the request, and the view name. If a view callable doesn't exist for this combination of objects (based on the type of the context, the type of the request, and the value of the view name, and any *predicate* attributes applied to the view configuration), Pyramid raises a *HTTPNotFound* exception, which is meant to be caught by a surrounding *exception view*.
14. If a view callable was found, Pyramid attempts to call it. If an *authorization policy* is in use, and the view configuration is protected by a *permission*, Pyramid determines whether the view callable being asked for can be executed by the requesting user based on credential information in the request and security information attached to the context. If the view execution is allowed, Pyramid calls the view callable to obtain a response. If view execution is forbidden, Pyramid raises a *HTTPForbidden* exception.
15. If any exception is raised within a *root factory*, by *traversal*, by a *view callable*, or by Pyramid itself (such as when it raises *HTTPNotFound* or *HTTPForbidden*), the router catches the exception, and attaches it to the request as the *exception* attribute. It then attempts to find a *exception view* for the exception that was caught. If it finds an exception view callable, that callable is called, and is presumed to generate a response. If an *exception view* that matches the exception cannot be found, the exception is reraised.
16. The following steps occur only when a *response* could be successfully generated by a normal *view callable* or an *exception view* callable. Pyramid will attempt to execute any *response callback* functions attached via `add_response_callback()`. A *NewResponse* event is then sent to any subscribers. The response object's `__call__` method is then used to generate a WSGI response. The response is sent back to the upstream WSGI server.
17. Pyramid will attempt to execute any *finished callback* functions attached via `add_finished_callback()`.
18. The *thread local* stack is popped.

Pyramid Router



This is a very high-level overview that leaves out various details. For more detail about subsystems invoked by the Pyramid router, such as traversal, URL dispatch, views, and event processing, see *URL Dispatch*, *Views*, and *Using Events*.

URL Dispatch

URL dispatch provides a simple way to map URLs to *view* code using a simple pattern matching language. An ordered set of patterns is checked one by one. If one of the patterns matches the path information associated with a request, a particular *view callable* is invoked. A view callable is a specific bit of code, defined in your application, that receives the *request* and returns a *response* object.

High-Level Operational Overview

If any route configuration is present in an application, the Pyramid *Router* checks every incoming request against an ordered set of URL matching patterns present in a *route map*.

If any route pattern matches the information in the *request*, Pyramid will invoke the *view lookup* process to find a matching view.

If no route pattern in the route map matches the information in the *request* provided in your application, Pyramid will fail over to using *traversal* to perform resource location and view lookup.

Route Configuration

Route configuration is the act of adding a new *route* to an application. A route has a *name*, which acts as an identifier to be used for URL generation. The name also allows developers to associate a view configuration with the route. A route also has a *pattern*, meant to match against the `PATH_INFO` portion of a URL (the portion following the scheme and port, e.g., `/foo/bar` in the URL `http://localhost:8080/foo/bar`). It also optionally has a *factory* and a set of *route predicate* attributes.

Configuring a Route to Match a View

The `pyramid.config.Configurator.add_route()` method adds a single *route configuration* to the *application registry*. Here's an example:

```
# "config" below is presumed to be an instance of the
# pyramid.config.Configurator class; "myview" is assumed
# to be a "view callable" function
from views import myview
config.add_route('myroute', '/prefix/{one}/{two}')
config.add_view(myview, route_name='myroute')
```

When a *view callable* added to the configuration by way of `add_view()` becomes associated with a route via its `route_name` predicate, that view callable will always be found and invoked when the associated route pattern matches during a request.

More commonly, you will not use any `add_view` statements in your project's "setup" code. You will instead use `add_route` statements, and use a *scan* to associate view callables with routes. For example, if this is a portion of your project's `__init__.py`:

```
config.add_route('myroute', '/prefix/{one}/{two}')
config.scan('mypackage')
```

Note that we don't call `add_view()` in this setup code. However, the above *scan* execution `config.scan('mypackage')` will pick up each *configuration decoration*, including any objects decorated with the `pyramid.view.view_config` decorator in the `mypackage` Python package. For example, if you have a `views.py` in your package, a scan will pick up any of its configuration decorators, so we can add one there that references `myroute` as a `route_name` parameter:

```
from pyramid.view import view_config
from pyramid.response import Response

@view_config(route_name='myroute')
def myview(request):
    return Response('OK')
```

The above combination of `add_route` and `scan` is completely equivalent to using the previous combination of `add_route` and `add_view`.

Route Pattern Syntax

The syntax of the pattern matching language used by Pyramid URL dispatch in the *pattern* argument is straightforward. It is close to that of the *Routes* system used by *Pylons*.

The *pattern* used in route configuration may start with a slash character. If the pattern does not start with a slash character, an implicit slash will be prepended to it at matching time. For example, the following patterns are equivalent:

```
{foo}/bar/baz
```

and:

```
/ {foo}/bar/baz
```

If a pattern is a valid URL it won't be matched against an incoming request. Instead it can be useful for generating external URLs. See *External routes* for details.

A pattern segment (an individual item between / characters in the pattern) may either be a literal string (e.g., `foo`) or it may be a replacement marker (e.g., `{foo}`), or a certain combination of both. A replacement marker does not need to be preceded by a / character.

A replacement marker is in the format `{name}`, where this means “accept any characters up to the next slash character and use this as the `name` *matchdict* value.”

A replacement marker in a pattern must begin with an uppercase or lowercase ASCII letter or an underscore, and can be composed only of uppercase or lowercase ASCII letters, underscores, and numbers. For example: `a`, `a_b`, `_b`, and `b9` are all valid replacement marker names, but `0a` is not.

Changed in version 1.2: A replacement marker could not start with an underscore until Pyramid 1.2. Previous versions required that the replacement marker start with an uppercase or lowercase letter.

A *matchdict* is the dictionary representing the dynamic parts extracted from a URL based on the routing pattern. It is available as `request.matchdict`. For example, the following pattern defines one literal segment (`foo`) and two replacement markers (`baz`, and `bar`):

```
foo/{baz}/{bar}
```

The above pattern will match these URLs, generating the following *matchdict*s:

```
foo/1/2      -> {'baz':u'1', 'bar':u'2'}
foo/abc/def   -> {'baz':u'abc', 'bar':u'def'}
```

It will not match the following patterns however:

```
foo/1/2/     -> No match (trailing slash)
bar/abc/def   -> First segment literal mismatch
```

The match for a segment replacement marker in a segment will be done only up to the first non-alphanumeric character in the segment in the pattern. So, for instance, if this route pattern was used:

```
foo/{name}.html
```

The literal path `/foo/biz.html` will match the above route pattern, and the match result will be `{'name': 'biz'}`. However, the literal path `/foo/biz` will not match, because it does not contain a literal `.html` at the end of the segment represented by `{name}.html` (it only contains `biz`, not `biz.html`).

To capture both segments, two replacement markers can be used:

```
foo/{name}.{ext}
```

The literal path `/foo/biz.html` will match the above route pattern, and the match result will be `{'name': 'biz', 'ext': 'html'}`. This occurs because there is a literal part of `.` (period) between the two replacement markers `{name}` and `{ext}`.

Replacement markers can optionally specify a regular expression which will be used to decide whether a path segment should match the marker. To specify that a replacement marker should match only a specific set of characters as defined by a regular expression, you must use a slightly extended form of replacement marker syntax. Within braces, the replacement marker name must be followed by a colon, then directly thereafter, the regular expression. The *default* regular expression associated with a replacement marker `[^/]+` matches one or more characters which are not a slash. For example, under the hood, the replacement marker `{foo}` can more verbosely be spelled as `{foo: [^/]+}`. You can change this to be an arbitrary regular expression to match an arbitrary sequence of characters, such as `{foo: \d+}` to match only digits.

It is possible to use two replacement markers without any literal characters between them, for instance `/ {foo} {bar}`. However, this would be a nonsensical pattern without specifying a custom regular expression to restrict what each marker captures.

Segments must contain at least one character in order to match a segment replacement marker. For example, for the URL `/abc/`:

- `/abc/{foo}` will not match.
- `/ {foo} /` will match.

Note that values representing matched path segments will be URL-unquoted and decoded from UTF-8 into Unicode within the `matchdict`. So for instance, the following pattern:

```
foo/{bar}
```

When matching the following URL:

```
http://example.com/foo/La%20Pe%C3%B1a
```

The matchdict will look like so (the value is URL-decoded / UTF-8 decoded):

```
{'bar':u'La Pe\xfla'}
```

Literal strings in the path segment should represent the *decoded* value of the `PATH_INFO` provided to Pyramid. You don’t want to use a URL-encoded value or a bytestring representing the literal encoded as UTF-8 in the pattern. For example, rather than this:

```
/Foo%20Bar/{baz}
```

You’ll want to use something like this:

```
/Foo Bar/{baz}
```

For patterns that contain “high-order” characters in its literals, you’ll want to use a Unicode value as the pattern as opposed to any URL-encoded or UTF-8-encoded value. For example, you might be tempted to use a bytestring pattern like this:

```
/La Pe\xc3\xbla/{x}
```

But this will either cause an error at startup time or it won’t match properly. You’ll want to use a Unicode value as the pattern instead rather than raw bytestring escapes. You can use a high-order Unicode value as the pattern by using Python source file encoding plus the “real” character in the Unicode pattern in the source, like so:

```
/La Peña/{x}
```

Or you can ignore source file encoding and use equivalent Unicode escape characters in the pattern.

```
/La Pe\xfla/{x}
```

Dynamic segment names cannot contain high-order characters, so this applies only to literals in the pattern.

If the pattern has a `*` in it, the name which follows it is considered a “remainder match”. A remainder match *must* come at the end of the pattern. Unlike segment replacement markers, it does not need to be preceded by a slash. For example:

```
foo/{baz}/{bar}*fizzle
```

The above pattern will match these URLs, generating the following matchdicts:

```
foo/1/2/          ->
    {'baz':u'1', 'bar':u'2', 'fizzle':()}

foo/abc/def/a/b/c ->
    {'baz':u'abc', 'bar':u'def', 'fizzle':(u'a', u'b', u'c')}
```

Note that when a `*stararg` remainder match is matched, the value put into the matchdict is turned into a tuple of path segments representing the remainder of the path. These path segments are URL-unquoted and decoded from UTF-8 into Unicode. For example, for the following pattern:

```
foo/*fizzle
```

When matching the following path:

```
/foo/La%20Pe%C3%B1a/a/b/c
```

Will generate the following matchdict:

```
{'fizzle':(u'La Pe\xfla', u'a', u'b', u'c')}
```

By default, the `*stararg` will parse the remainder sections into a tuple split by segment. Changing the regular expression used to match a marker can also capture the remainder of the URL, for example:

```
foo/{baz}/{bar}{fizzle:.*}
```

The above pattern will match these URLs, generating the following matchdicts:

```
foo/1/2/          -> {'baz':u'1', 'bar':u'2', 'fizzle':u''}
foo/abc/def/a/b/c -> {'baz':u'abc', 'bar':u'def', 'fizzle': u'a/b/c'}
```

This occurs because the default regular expression for a marker is `[^/]+` which will match everything up to the first `/`, while `{fizzle:.*}` will result in a regular expression match of `.*` capturing the remainder into a single value.

Route Declaration Ordering

Route configuration declarations are evaluated in a specific order when a request enters the system. As a result, the order of route configuration declarations is very important. The order in which route declarations are evaluated is the order in which they are added to the application at startup time. (This is unlike a different way of mapping URLs to code that Pyramid provides, named *traversal*, which does not depend on pattern ordering).

For routes added via the `add_route` method, the order that routes are evaluated is the order in which they are added to the configuration imperatively.

For example, route configuration statements with the following patterns might be added in the following order:

```
members/{def}  
members/abc
```

In such a configuration, the `members/abc` pattern would *never* be matched. This is because the match ordering will always match `members/{def}` first; the route configuration with `members/abc` will never be evaluated.

Route Configuration Arguments

Route configuration `add_route` statements may specify a large number of arguments. They are documented as part of the API documentation at `pyramid.config.Configurator.add_route()`.

Many of these arguments are *route predicate* arguments. A route predicate argument specifies that some aspect of the request must be true for the associated route to be considered a match during the route matching process. Examples of route predicate arguments are `pattern`, `xhr`, and `request_method`.

Other arguments are `name` and `factory`. These arguments represent neither predicates nor view configuration information.

Route Matching

The main purpose of route configuration is to match (or not match) the `PATH_INFO` present in the WSGI environment provided during a request against a URL path pattern. `PATH_INFO` represents the path portion of the URL that was requested.

The way that Pyramid does this is very simple. When a request enters the system, for each route configuration declaration present in the system, Pyramid checks the request's `PATH_INFO` against the pattern declared. This checking happens in the order that the routes were declared via `pyramid.config.Configurator.add_route()`.

When a route configuration is declared, it may contain *route predicate* arguments. All route predicates associated with a route declaration must be `True` for the route configuration to be used for a given request during a check. If any predicate in the set of *route predicate* arguments provided to a route configuration returns `False` during a check, that route is skipped and route matching continues through the ordered set of routes.

If any route matches, the route matching process stops and the *view lookup* subsystem takes over to find the most reasonable view callable for the matched route. Most often, there's only one view that will match (a view configured with a `route_name` argument matching the matched route). To gain a better understanding of how routes and views are associated in a real application, you can use the `pviews` command, as documented in *Displaying Matching Views for a Given URL*.

If no route matches after all route patterns are exhausted, Pyramid falls back to *traversal* to do *resource location* and *view lookup*.

The Matchdict


When the URL pattern associated with a particular route configuration is matched by a request, a dictionary named `matchdict` is added as an attribute of the *request* object. Thus, `request.matchdict` will contain the values that match replacement patterns in the `pattern` element. The keys in a `matchdict` will be strings. The values will be Unicode objects.



If no route URL pattern matches, the `matchdict` object attached to the request will be `None`.

The Matched Route

When the URL pattern associated with a particular route configuration is matched by a request, an object named `matched_route` is added as an attribute of the `request` object. Thus, `request.matched_route` will be an object implementing the `IRoute` interface which matched the request. The most useful attribute of the route object is `name`, which is the name of the route that matched.

 If no route URL pattern matches, the `matched_route` object attached to the request will be `None`.

Routing Examples

Let's check out some examples of how route configuration statements might be commonly declared, and what will happen if they are matched by the information present in a request.

Example 1

The simplest route declaration which configures a route match to *directly* result in a particular view callable being invoked:

```
1 config.add_route('idea', 'site/{id}')
2 config.scan()
```

When a route configuration with a `view` attribute is added to the system, and an incoming request matches the *pattern* of the route configuration, the *view callable* named as the `view` attribute of the route configuration will be invoked.

Recall that the `@view_config` is equivalent to calling `config.add_view`, because the `config.scan()` call will import `mypackage.views`, shown below, and execute `config.add_view` under the hood. Each view then maps the route name to the matching view callable. In the case of the above example, when the URL of a request matches `/site/{id}`, the view callable at the Python dotted path name `mypackage.views.site_view` will be called with the request. In other words, we've associated a view callable directly with a route pattern.

When the `/site/{id}` route pattern matches during a request, the `site_view` view callable is invoked with that request as its sole argument. When this route matches, a `matchdict` will be generated and attached to the request as `request.matchdict`. If the specific URL matched is `/site/1`, the `matchdict` will be a dictionary with a single key, `id`; the value will be the string `'1'`, ex.: `{'id': '1'}`.

The `mypackage.views` module referred to above might look like so:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='idea')
5 def site_view(request):
6     return Response(request.matchdict['id'])
```

The view has access to the matchdict directly via the request, and can access variables within it that match keys present as a result of the route pattern.

See *Views*, and *View Configuration* for more information about views.

Example 2

Below is an example of a more complicated set of route statements you might add to your application:

```
1 config.add_route('idea', 'ideas/{idea}')
2 config.add_route('user', 'users/{user}')
3 config.add_route('tag', 'tags/{tag}')
4 config.scan()
```

Here is an example of a corresponding `mypackage.views` module:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='idea')
5 def idea_view(request):
6     return Response(request.matchdict['idea'])
7
8 @view_config(route_name='user')
9 def user_view(request):
10     user = request.matchdict['user']
11     return Response(u'The user is {}'.format(user))
12
13 @view_config(route_name='tag')
14 def tag_view(request):
15     tag = request.matchdict['tag']
16     return Response(u'The tag is {}'.format(tag))
```

The above configuration will allow Pyramid to service URLs in these forms:

```
/ideas/{idea}  
/users/{user}  
/tags/{tag}
```

- When a URL matches the pattern `/ideas/{idea}`, the view callable available at the dotted Python pathname `mypackage.views.idea_view` will be called. For the specific URL `/ideas/1`, the `matchdict` generated and attached to the *request* will consist of `{'idea': '1'}`.
- When a URL matches the pattern `/users/{user}`, the view callable available at the dotted Python pathname `mypackage.views.user_view` will be called. For the specific URL `/users/1`, the `matchdict` generated and attached to the *request* will consist of `{'user': '1'}`.
- When a URL matches the pattern `/tags/{tag}`, the view callable available at the dotted Python pathname `mypackage.views.tag_view` will be called. For the specific URL `/tags/1`, the `matchdict` generated and attached to the *request* will consist of `{'tag': '1'}`.

In this example we've again associated each of our routes with a *view callable* directly. In all cases, the request, which will have a `matchdict` attribute detailing the information found in the URL by the process will be passed to the view callable.

Example 3

The *context* resource object passed in to a view found as the result of URL dispatch will, by default, be an instance of the object returned by the *root factory* configured at startup time (the `root_factory` argument to the *Configurator* used to configure the application).

You can override this behavior by passing in a *factory* argument to the `add_route()` method for a particular route. The *factory* should be a callable that accepts a *request* and returns an instance of a class that will be the context resource used by the view.

An example of using a route with a factory:

```
1 config.add_route('idea', 'ideas/{idea}', factory='myproject.resources.Idea  
  ↳')  
2 config.scan()
```

The above route will manufacture an *Idea* resource as a *context*, assuming that `mypackage.resources.Idea` resolves to a class that accepts a request in its `__init__`. For example:

```
1 class Idea(object):
2     def __init__(self, request):
3         pass
```

In a more complicated application, this root factory might be a class representing a *SQLAlchemy* model. The view `mypackage.views.idea_view` might look like this:

```
1 @view_config(route_name='idea')
2 def idea_view(request):
3     idea = request.context
4     return Response(idea)
```

Here, `request.context` is an instance of `Idea`. If indeed the resource object is a *SQLAlchemy* model, you do not even have to perform a query in the view callable, since you have access to the resource via `request.context`.

See *Route Factories* for more details about how to use route factories.

Matching the Root URL

It's not entirely obvious how to use a route pattern to match the root URL (“/”). To do so, give the empty string as a pattern in a call to `add_route()`:

```
1 config.add_route('root', '')
```

Or provide the literal string `/` as the pattern:

```
1 config.add_route('root', '/')
```

Generating Route URLs

Use the `pyramid.request.Request.route_url()` method to generate URLs based on route patterns. For example, if you've configured a route with the name “foo” and the pattern “{a}/{b}/{c}”, you might do this.

```
1 url = request.route_url('foo', a='1', b='2', c='3')
```

This would return something like the string `http://example.com/1/2/3` (at least if the current protocol and hostname implied `http://example.com`).

To generate only the *path* portion of a URL from a route, use the `pyramid.request.Request.route_path()` API instead of `route_url()`.

```
url = request.route_path('foo', a='1', b='2', c='3')
```

This will return the string `/1/2/3` rather than a full URL.

Replacement values passed to `route_url` or `route_path` must be Unicode or bytestrings encoded in UTF-8. One exception to this rule exists: if you're trying to replace a “remainder” match value (a `*stararg` replacement value), the value may be a tuple containing Unicode strings or UTF-8 strings.

Note that URLs and paths generated by `route_url` and `route_path` are always URL-quoted string types (they contain no non-ASCII characters). Therefore, if you've added a route like so:

```
config.add_route('la', u'/La Peña/{city}')
```

And you later generate a URL using `route_path` or `route_url` like so:

```
url = request.route_path('la', city=u'Québec')
```

You will wind up with the path encoded to UTF-8 and URL-quoted like so:

```
/La%20Pe%C3%B1a/Qu%C3%A9bec
```

If you have a `*stararg` remainder dynamic part of your route pattern:

```
config.add_route('abc', 'a/b/c/*foo')
```

And you later generate a URL using `route_path` or `route_url` using a *string* as the replacement value:

```
url = request.route_path('abc', foo=u'Québec/biz')
```

The value you pass will be URL-quoted except for embedded slashes in the result:

```
/a/b/c/Qu%C3%A9bec/biz
```

You can get a similar result by passing a tuple composed of path elements:

```
url = request.route_path('abc', foo=(u'Québec', u'biz'))
```

Each value in the tuple will be URL-quoted and joined by slashes in this case:

```
/a/b/c/Qu%C3%A9bec/biz
```

Static Routes

Routes may be added with a `static` keyword argument. For example:

```
1 config = Configurator()
2 config.add_route('page', '/page/{action}', static=True)
```

Routes added with a `True` `static` keyword argument will never be considered for matching at request time. Static routes are useful for URL generation purposes only. As a result, it is usually nonsensical to provide other non-name and non-pattern arguments to `add_route()` when `static` is passed as `True`, as none of the other arguments will ever be employed. A single exception to this rule is use of the `pregenerator` argument, which is not ignored when `static` is `True`.

External routes are implicitly static.

New in version 1.1: the `static` argument to `add_route()`.

External Routes

New in version 1.5.

Route patterns that are valid URLs, are treated as external routes. Like *static routes* they are useful for URL generation purposes only and are never considered for matching at request time.


```
1 >>> config = Configurator()
2 >>> config.add_route('youtube', 'https://youtube.com/watch/{video_id}')
3 ...
4 >>> request.route_url('youtube', video_id='oHg5SJYRHA0')
5 >>> "https://youtube.com/watch/oHg5SJYRHA0"
```

Most pattern replacements and calls to `pyramid.request.Request.route_url()` will work as expected. However, calls to `pyramid.request.Request.route_path()` against external patterns will raise an exception, and passing `_app_url` to `route_url()` to generate a URL against a route that has an external pattern will also raise an exception.

Redirecting to Slash-Appended Routes

For behavior like Django's `APPEND_SLASH=True`, use the `append_slash` argument to `pyramid.config.Configurator.add_notfound_view()` or the equivalent `append_slash` argument to the `pyramid.view.notfound_view_config` decorator.

Adding `append_slash=True` is a way to automatically redirect requests where the URL lacks a trailing slash, but requires one to match the proper route. When configured, along with at least one other route in your application, this view will be invoked if the value of `PATH_INFO` does not already end in a slash, and if the value of `PATH_INFO` *plus* a slash matches any route's pattern. In this case it does an HTTP redirect to the slash-appended `PATH_INFO`. In addition you may pass anything that implements `pyramid.interfaces.IResponse` which will then be used in place of the default class (`pyramid.httpexceptions.HTTPFound`).

Let's use an example. If the following routes are configured in your application:

```
1 from pyramid.httpexceptions import HTTPNotFound
2
3 def notfound(request):
4     return HTTPNotFound()
5
6 def no_slash(request):
7     return Response('No slash')
8
9 def has_slash(request):
10    return Response('Has slash')
11
12 def main(g, **settings):
13     config = Configurator()
14     config.add_route('noslash', 'no_slash')
```

```

15 config.add_route('haslash', 'has_slash/')
16 config.add_view(no_slash, route_name='noslash')
17 config.add_view(has_slash, route_name='haslash')
18 config.add_notfound_view(notfound, append_slash=True)

```

If a request enters the application with the `PATH_INFO` value of `/no_slash`, the first route will match and the browser will show “No slash”. However, if a request enters the application with the `PATH_INFO` value of `/no_slash/`, *no* route will match, and the slash-appending not found view will not find a matching route with an appended slash. As a result, the `notfound` view will be called and it will return a “Not found” body.

If a request enters the application with the `PATH_INFO` value of `/has_slash/`, the second route will match. If a request enters the application with the `PATH_INFO` value of `/has_slash`, a route *will* be found by the slash-appending *Not Found View*. An HTTP redirect to `/has_slash/` will be returned to the user’s browser. As a result, the `notfound` view will never actually be called.

The following application uses the `pyramid.view.notfound_view_config` and `pyramid.view.view_config` decorators and a `scan` to do exactly the same job:

```

1 from pyramid.httpexceptions import HTTPNotFound
2 from pyramid.view import notfound_view_config, view_config
3
4 @notfound_view_config(append_slash=True)
5 def notfound(request):
6     return HTTPNotFound()
7
8 @view_config(route_name='noslash')
9 def no_slash(request):
10     return Response('No slash')
11
12 @view_config(route_name='haslash')
13 def has_slash(request):
14     return Response('Has slash')
15
16 def main(g, **settings):
17     config = Configurator()
18     config.add_route('noslash', 'no_slash')
19     config.add_route('haslash', 'has_slash/')
20     config.scan()

```



You **should not** rely on this mechanism to redirect POST requests. The redirect of the slash-appending *Not Found View* will turn a POST request into a GET, losing any POST data in the original request.

See *pyramid.view* and *Changing the Not Found View* for a more general description of how to configure a view and/or a *Not Found View*.

Debugging Route Matching

It's useful to be able to take a peek under the hood when requests that enter your application aren't matching your routes as you expect them to. To debug route matching, use the `PYRAMID_DEBUG_ROUTE MATCH` environment variable or the `pyramid.debug_routematch` configuration file setting (set either to `true`). Details of the route matching decision for a particular request to the Pyramid application will be printed to the `stderr` of the console which you started the application from. For example:

```
1 $ PYRAMID_DEBUG_ROUTE MATCH=true $VENV/bin/pserve development.ini
2 Starting server in PID 13586.
3 serving on 0.0.0.0:6543 view at http://127.0.0.1:6543
4 2010-12-16 14:45:19,956 no route matched for url \
5                               http://localhost:6543/wontmatch
6 2010-12-16 14:45:20,010 no route matched for url \
7                               http://localhost:6543/favicon.ico
8 2010-12-16 14:41:52,084 route matched for url \
9                               http://localhost:6543/static/logo.png; \
10                              route_name: 'static/', ....
```

See *Environment Variables and .ini File Settings* for more information about how and where to set these values.

You can also use the `proutes` command to see a display of all the routes configured in your application. For more information, see *Displaying All Application Routes*.

Using a Route Prefix to Compose Applications

New in version 1.2.

The `pyramid.config.Configurator.include()` method allows configuration statements to be included from separate files. See *Rules for Building an Extensible Application* for information about this method. Using `pyramid.config.Configurator.include()` allows you to build your application from small and potentially reusable components.

The `pyramid.config.Configurator.include()` method accepts an argument named `route_prefix` which can be useful to authors of URL-dispatch-based applications. If

`route_prefix` is supplied to the `include` method, it must be a string. This string represents a route prefix that will be prepended to all route patterns added by the *included* configuration. Any calls to `pyramid.config.Configurator.add_route()` within the included callable will have their pattern prefixed with the value of `route_prefix`. This can be used to help mount a set of routes at a different location than the included callable's author intended while still maintaining the same route names. For example:

```
1 from pyramid.config import Configurator
2
3 def users_include(config):
4     config.add_route('show_users', '/show')
5
6 def main(global_config, **settings):
7     config = Configurator()
8     config.include(users_include, route_prefix='/users')
```

In the above configuration, the `show_users` route will have an effective route pattern of `/users/show` instead of `/show` because the `route_prefix` argument will be prepended to the pattern. The route will then only match if the URL path is `/users/show`, and when the `pyramid.request.Request.route_url()` function is called with the route name `show_users`, it will generate a URL with that same path.

Route prefixes are recursive, so if a callable executed via an `include` itself turns around and includes another callable, the second-level route prefix will be prepended with the first:

```
1 from pyramid.config import Configurator
2
3 def timing_include(config):
4     config.add_route('show_times', '/times')
5
6 def users_include(config):
7     config.add_route('show_users', '/show')
8     config.include(timing_include, route_prefix='/timing')
9
10 def main(global_config, **settings):
11     config = Configurator()
12     config.include(users_include, route_prefix='/users')
```

In the above configuration, the `show_users` route will still have an effective route pattern of `/users/show`. The `show_times` route, however, will have an effective pattern of `/users/timing/times`.

Route prefixes have no impact on the requirement that the set of route *names* in any given Pyramid configuration must be entirely unique. If you compose your URL dispatch application out of many small

subapplications using `pyramid.config.Configurator.include()`, it's wise to use a dotted name for your route names so they'll be unlikely to conflict with other packages that may be added in the future. For example:

```
1 from pyramid.config import Configurator
2
3 def timing_include(config):
4     config.add_route('timing.show_times', '/times')
5
6 def users_include(config):
7     config.add_route('users.show_users', '/show')
8     config.include(timing_include, route_prefix='/timing')
9
10 def main(global_config, **settings):
11     config = Configurator()
12     config.include(users_include, route_prefix='/users')
```

Custom Route Predicates

Each of the predicate callables fed to the `custom_predicates` argument of `add_route()` must be a callable accepting two arguments. The first argument passed to a custom predicate is a dictionary conventionally named `info`. The second argument is the current *request* object.

The `info` dictionary has a number of contained values, including `match` and `route`. `match` is a dictionary which represents the arguments matched in the URL by the route. `route` is an object representing the route which was matched (see `pyramid.interfaces.IRoute` for the API of such a route object).

`info['match']` is useful when predicates need access to the route match. For example:

```
1 def any_of(segment_name, *allowed):
2     def predicate(info, request):
3         if info['match'][segment_name] in allowed:
4             return True
5     return predicate
6
7 num_one_two_or_three = any_of('num', 'one', 'two', 'three')
8
9 config.add_route('route_to_num', '/{num}',
10                  custom_predicates=(num_one_two_or_three,))
```

The above `any_of` function generates a predicate which ensures that the match value named `segment_name` is in the set of allowable values represented by `allowed`. We use this `any_of` function to generate a predicate function named `num_one_two_or_three`, which ensures that the `num` segment is one of the values `one`, `two`, or `three`, and use the result as a custom predicate by feeding it inside a tuple to the `custom_predicates` argument to `add_route()`.

A custom route predicate may also *modify* the match dictionary. For instance, a predicate might do some type conversion of values:

```

1 def integers(*segment_names):
2     def predicate(info, request):
3         match = info['match']
4         for segment_name in segment_names:
5             try:
6                 match[segment_name] = int(match[segment_name])
7             except (TypeError, ValueError):
8                 pass
9         return True
10    return predicate
11
12 ymd_to_int = integers('year', 'month', 'day')
13
14 config.add_route('ymd', '{year}/{month}/{day}',
15                  custom_predicates=(ymd_to_int,))

```

Note that a conversion predicate is still a predicate, so it must return `True` or `False`. A predicate that does *only* conversion, such as the one we demonstrate above, should unconditionally return `True`.

To avoid the try/except uncertainty, the route pattern can contain regular expressions specifying requirements for that marker. For instance:

```

1 def integers(*segment_names):
2     def predicate(info, request):
3         match = info['match']
4         for segment_name in segment_names:
5             match[segment_name] = int(match[segment_name])
6         return True
7     return predicate
8
9 ymd_to_int = integers('year', 'month', 'day')
10
11 config.add_route('ymd', '{year:\d+}/{month:\d+}/{day:\d+}',
12                  custom_predicates=(ymd_to_int,))

```

Now the try/except is no longer needed because the route will not match at all unless these markers match `\d+` which requires them to be valid digits for an `int` type conversion.

The `match` dictionary passed within `info` to each predicate attached to a route will be the same dictionary. Therefore, when registering a custom predicate which modifies the `match` dict, the code registering the predicate should usually arrange for the predicate to be the *last* custom predicate in the custom predicate list. Otherwise, custom predicates which fire subsequent to the predicate which performs the `match` modification will receive the *modified* match dictionary.



It is a poor idea to rely on ordering of custom predicates to build a conversion pipeline, where one predicate depends on the side effect of another. For instance, it's a poor idea to register two custom predicates, one which handles conversion of a value to an `int`, the next which handles conversion of that integer to some custom object. Just do all that in a single custom predicate.

The `route` object in the `info` dict is an object that has two useful attributes: `name` and `pattern`. The `name` attribute is the route name. The `pattern` attribute is the route pattern. Here's an example of using the route in a set of route predicates:

```
1 def twenty_ten(info, request):
2     if info['route'].name in ('ymd', 'ym', 'y'):
3         return info['match']['year'] == '2010'
4
5 config.add_route('y', '/{year}', custom_predicates=(twenty_ten,))
6 config.add_route('ym', '/{year}/{month}', custom_predicates=(twenty_ten,))
7 config.add_route('ymd', '/{year}/{month}/{day}',
8                 custom_predicates=(twenty_ten,))
```

The above predicate, when added to a number of route configurations ensures that the year match argument is '2010' if and only if the route name is 'ymd', 'ym', or 'y'.

You can also caption the predicates by setting the `__text__` attribute. This will help you with the `pviews` command (see *Displaying All Application Routes*) and the `pyramid_debugtoolbar`.

If a predicate is a class, just add `__text__` property in a standard manner.

```
1 class DummyCustomPredicate1(object):
2     def __init__(self):
3         self.__text__ = 'my custom class predicate'
4
5 class DummyCustomPredicate2(object):
6     __text__ = 'my custom class predicate'
```

If a predicate is a method, you’ll need to assign it after method declaration (see PEP 232).

```
1 def custom_predicate():
2     pass
3 custom_predicate.__text__ = 'my custom method predicate'
```

If a predicate is a classmethod, using `@classmethod` will not work, but you can still easily do it by wrapping it in a classmethod call.

```
1 def classmethod_predicate():
2     pass
3 classmethod_predicate.__text__ = 'my classmethod predicate'
4 classmethod_predicate = classmethod(classmethod_predicate)
```

The same will work with `staticmethod`, using `staticmethod` instead of `classmethod`.

See also:

See also `pyramid.interfaces.IRoute` for more API documentation about route objects.

Route Factories

Although it is not a particularly common need in basic applications, a “route” configuration declaration can mention a “factory”. When a route matches a request, and a factory is attached to the route, the *root factory* passed at startup time to the *Configurator* is ignored. Instead the factory associated with the route is used to generate a *root* object. This object will usually be used as the *context* resource of the view callable ultimately found via *view lookup*.

```
1 config.add_route('abc', '/abc',
2                 factory='myproject.resources.root_factory')
3 config.add_view('myproject.views.theview', route_name='abc')
```

The factory can either be a Python object or a *dotted Python name* (a string) which points to such a Python object, as it is above.

In this way, each route can use a different factory, making it possible to supply a different *context* resource object to the view related to each particular route.

A factory must be a callable which accepts a request and returns an arbitrary Python object. For example, the below class can be used as a factory:


```
1 class Mine(object):
2     def __init__(self, request):
3         pass
```

A route factory is actually conceptually identical to the *root factory* described at *The Resource Tree*.

Supplying a different resource factory for each route is useful when you're trying to use a Pyramid *authorization policy* to provide declarative, “context sensitive” security checks. Each resource can maintain a separate *ACL*, as documented in *Using Pyramid Security with URL Dispatch*. It is also useful when you wish to combine URL dispatch with *traversal* as documented within *Combining Traversal and URL Dispatch*.

Using Pyramid Security with URL Dispatch

Pyramid provides its own security framework which consults an *authorization policy* before allowing any application code to be called. This framework operates in terms of an access control list, which is stored as an `__acl__` attribute of a resource object. A common thing to want to do is to attach an `__acl__` to the resource object dynamically for declarative security purposes. You can use the `factory` argument that points at a factory which attaches a custom `__acl__` to an object at its creation time.

Such a factory might look like so:

```
1 class Article(object):
2     def __init__(self, request):
3         matchdict = request.matchdict
4         article = matchdict.get('article', None)
5         if article == '1':
6             self.__acl__ = [ (Allow, 'editor', 'view') ]
```

If the route `archives/{article}` is matched, and the article number is 1, Pyramid will generate an *Article context* resource with an ACL on it that allows the `editor` principal the `view` permission. Obviously you can do more generic things than inspect the route's match dict to see if the `article` argument matches a particular string. Our sample *Article* factory class is not very ambitious.



See *Security* for more information about Pyramid security and ACLs.

Route View Callable Registration and Lookup Details

When a request enters the system which matches the pattern of the route, the usual result is simple: the view callable associated with the route is invoked with the request that caused the invocation.

For most usage, you needn't understand more than this. How it works is an implementation detail. In the interest of completeness, however, we'll explain how it *does* work in this section. You can skip it if you're uninterested.

When a view is associated with a route configuration, Pyramid ensures that a *view configuration* is registered that will always be found when the route pattern is matched during a request. To do so:

- A special route-specific *interface* is created at startup time for each route configuration declaration.
- When an `add_view` statement mentions a `route_name` attribute, a *view configuration* is registered at startup time. This view configuration uses a route-specific interface as a *request* type.
- At runtime, when a request causes any route to match, the *request* object is decorated with the route-specific interface.
- The fact that the request is decorated with a route-specific interface causes the *view lookup* machinery to always use the view callable registered using that interface by the route configuration to service requests that match the route pattern.

As we can see from the above description, technically, URL dispatch doesn't actually map a URL pattern directly to a view callable. Instead URL dispatch is a *resource location* mechanism. A Pyramid *resource location* subsystem (i.e., *URL dispatch* or *traversal*) finds a *resource* object that is the *context* of a *request*. Once the *context* is determined, a separate subsystem named *view lookup* is then responsible for finding and invoking a *view callable* based on information available in the context and the request. When URL dispatch is used, the resource location and view lookup subsystems provided by Pyramid are still being utilized, but in a way which does not require a developer to understand either of them in detail.

If no route is matched using *URL dispatch*, Pyramid falls back to *traversal* to handle the *request*.

References

A tutorial showing how *URL dispatch* can be used to create a Pyramid application exists in *SQLAlchemy + URL dispatch wiki tutorial*.

Views

One of the primary jobs of Pyramid is to find and invoke a *view callable* when a *request* reaches your application. View callables are bits of code which do something interesting in response to a request made to your application. They are the “meat” of any interesting web application.



A Pyramid *view callable* is often referred to in conversational shorthand as a *view*. In this documentation, however, we need to use less ambiguous terminology because there are significant differences between *view configuration*, the code that implements a *view callable*, and the process of *view lookup*.

This chapter describes how view callables should be defined. We’ll have to wait until a following chapter (entitled *View Configuration*) to find out how we actually tell Pyramid to wire up view callables to particular URL patterns and other request circumstances.

View Callables

View callables are, at the risk of sounding obvious, callable Python objects. Specifically, view callables can be functions, classes, or instances that implement a `__call__` method (making the instance callable).

View callables must, at a minimum, accept a single argument named `request`. This argument represents a Pyramid *Request* object. A request object represents a *WSGI* environment provided to Pyramid by the upstream WSGI server. As you might expect, the request object contains everything your application needs to know about the specific HTTP request being made.

A view callable’s ultimate responsibility is to create a Pyramid *Response* object. This can be done by creating a *Response* object in the view callable code and returning it directly or by raising special kinds of exceptions from within the body of a view callable.

Defining a View Callable as a Function

One of the easiest ways to define a view callable is to create a function that accepts a single argument named `request`, and which returns a *Response* object. For example, this is a “hello world” view callable implemented as a function:

```
1 from pyramid.response import Response
2
3 def hello_world(request):
4     return Response('Hello world!')
```

Defining a View Callable as a Class

A view callable may also be represented by a Python class instead of a function. When a view callable is a class, the calling semantics are slightly different than when it is a function or another non-class callable. When a view callable is a class, the class's `__init__` method is called with a `request` parameter. As a result, an instance of the class is created. Subsequently, that instance's `__call__` method is invoked with no parameters. Views defined as classes must have the following traits.

- an `__init__` method that accepts a `request` argument
- a `__call__` (or other) method that accepts no parameters and which returns a response

For example:

```
1 from pyramid.response import Response
2
3 class MyView(object):
4     def __init__(self, request):
5         self.request = request
6
7     def __call__(self):
8         return Response('hello')
```

The request object passed to `__init__` is the same type of request object described in *Defining a View Callable as a Function*.


If you'd like to use a different attribute than `__call__` to represent the method expected to return a response, you can use an `attr` value as part of the configuration for the view. See *View Configuration Parameters*. The same view callable class can be used in different view configuration statements with different `attr` values, each pointing at a different method of the class if you'd like the class to represent a collection of related view callables.

View Callable Responses

A view callable may return an object that implements the Pyramid *Response* interface. The easiest way to return something that implements the *Response* interface is to return a `pyramid.response.Response` object instance directly. For example:

```
1 from pyramid.response import Response
2
3 def view(request):
4     return Response('OK')
```

Pyramid provides a range of different “exception” classes which inherit from `pyramid.response.Response`. For example, an instance of the class `pyramid.httpexceptions.HTTPFound` is also a valid response object because it inherits from `Response`. For examples, see *HTTP Exceptions* and *Using a View Callable to do an HTTP Redirect*.

 You can also return objects from view callables that aren’t instances of `pyramid.response.Response` in various circumstances. This can be helpful when writing tests and when attempting to share code between view callables. See *Renderers* for the common way to allow for this. A much less common way to allow for view callables to return non-Response objects is documented in *Changing How Pyramid Treats View Responses*.

Using Special Exceptions in View Callables

Usually when a Python exception is raised within a view callable, Pyramid allows the exception to propagate all the way out to the *WSGI* server which invoked the application. It is usually caught and logged there.

However, for convenience, a special set of exceptions exists. When one of these exceptions is raised within a view callable, it will always cause Pyramid to generate a response. These are known as *HTTP exception* objects.

HTTP Exceptions

All `pyramid.httpexceptions` classes which are documented as inheriting from the `pyramid.httpexceptions.HTTPException` are *http exception* objects. Instances of an HTTP exception object may either be *returned* or *raised* from within view code. In either case (return or raise) the instance will be used as the view’s response.

For example, the `pyramid.httpexceptions.HTTPUnauthorized` exception can be raised. This will cause a response to be generated with a 401 `Unauthorized` status:

```
1 from pyramid.httpexceptions import HTTPUnauthorized
2
3 def aview(request):
4     raise HTTPUnauthorized()
```

An HTTP exception, instead of being raised, can alternately be *returned* (HTTP exceptions are also valid response objects):

```
1 from pyramid.httpexceptions import HTTPUnauthorized
2
3 def aview(request):
4     return HTTPUnauthorized()
```

A shortcut for creating an HTTP exception is the `pyramid.httpexceptions.exception_response()` function. This function accepts an HTTP status code and returns the corresponding HTTP exception. For example, instead of importing and constructing a `HTTPUnauthorized` response object, you can use the `exception_response()` function to construct and return the same object.

```
1 from pyramid.httpexceptions import exception_response
2
3 def aview(request):
4     raise exception_response(401)
```

This is the case because 401 is the HTTP status code for “HTTP Unauthorized”. Therefore, `raise exception_response(401)` is functionally equivalent to `raise HTTPUnauthorized()`. Documentation which maps each HTTP response code to its purpose and its associated HTTP exception object is provided within `pyramid.httpexceptions`.

New in version 1.1: The `exception_response()` function.

How Pyramid Uses HTTP Exceptions

HTTP exceptions are meant to be used directly by application developers. However, Pyramid itself will raise two HTTP exceptions at various points during normal operations.

- `HTTPNotFound` gets raised when a view to service a request is not found.
- `HTTPForbidden` gets raised when authorization was forbidden by a security policy.

If `HTTPNotFound` is raised by Pyramid itself or within view code, the result of the *Not Found View* will be returned to the user agent which performed the request.

If `HTTPForbidden` is raised by Pyramid itself within view code, the result of the *Forbidden View* will be returned to the user agent which performed the request.

Custom Exception Views

The machinery which allows HTTP exceptions to be raised and caught by specialized views as described in *Using Special Exceptions in View Callables* can also be used by application developers to convert arbitrary exceptions to responses.

To register a view that should be called whenever a particular exception is raised from within Pyramid view code, use the exception class (or one of its superclasses) as the *context* of a view configuration which points at a view callable for which you'd like to generate a response.

For example, given the following exception class in a module named `helloworld.exceptions`:

```
1 class ValidationFailure(Exception):
2     def __init__(self, msg):
3         self.msg = msg
```

You can wire a view callable to be called whenever any of your *other* code raises a `helloworld.exceptions.ValidationFailure` exception:

```
1 from pyramid.view import view_config
2 from helloworld.exceptions import ValidationFailure
3
4 @view_config(context=ValidationFailure)
5 def failed_validation(exc, request):
6     response = Response('Failed validation: %s' % exc.msg)
7     response.status_int = 500
8     return response
```

Assuming that a *scan* was run to pick up this view registration, this view callable will be invoked whenever a `helloworld.exceptions.ValidationFailure` is raised by your application's view code. The same exception raised by a custom root factory, a custom traverser, or a custom view or route predicate is also caught and hooked.

Other normal view predicates can also be used in combination with an exception view registration:

```
1 from pyramid.view import view_config
2 from helloworld.exceptions import ValidationFailure
3
4 @view_config(context=ValidationFailure, route_name='home')
5 def failed_validation(exc, request):
6     response = Response('Failed validation: %s' % exc.msg)
7     response.status_int = 500
8     return response
```

The above exception view names the `route_name` of `home`, meaning that it will only be called when the route matched has a name of `home`. You can therefore have more than one exception view for any given exception in the system: the “most specific” one will be called when the set of request circumstances match the view registration.

The only view predicate that cannot be used successfully when creating an exception view configuration is `name`. The name used to look up an exception view is always the empty string. Views registered as exception views which have a name will be ignored.

i Normal (i.e., non-exception) views registered against a context resource type which inherits from `Exception` will work normally. When an exception view configuration is processed, *two* views are registered. One as a “normal” view, the other as an “exception” view. This means that you can use an exception as `context` for a normal view.

Exception views can be configured with any view registration mechanism: `@view_config` decorator or imperative `add_view` styles.

i Pyramid’s *exception view* handling logic is implemented as a tween factory function: `pyramid.tweens.excview_tween_factory()`. If Pyramid exception view handling is desired, and tween factories are specified via the `pyramid.tweens` configuration setting, the `pyramid.tweens.excview_tween_factory()` function must be added to the `pyramid.tweens` configuration setting list explicitly. If it is not present, Pyramid will not perform exception view handling.

Using a View Callable to do an HTTP Redirect

You can issue an HTTP redirect by using the `pyramid.httpexceptions.HTTPFound` class. Raising or returning an instance of this class will cause the client to receive a “302 Found” response.

To do so, you can *return* a `pyramid.httpexceptions.HTTPFound` instance.

```
1 from pyramid.httpexceptions import HTTPFound
2
3 def myview(request):
4     return HTTPFound(location='http://example.com')
```

Alternately, you can *raise* an `HTTPFound` exception instead of returning one.


```
1 from pyramid.httpexceptions import HTTPFound
2
3 def myview(request):
4     raise HTTPFound(location='http://example.com')
```

When the instance is raised, it is caught by the default *exception response* handler and turned into a response.

Handling Form Submissions in View Callables (Unicode and Character Set Issues)

Most web applications need to accept form submissions from web browsers and various other clients. In Pyramid, form submission handling logic is always part of a *view*. For a general overview of how to handle form submission data using the *WebOb* API, see *Request and Response Objects* and “Query and POST variables” within the *WebOb* documentation. Pyramid defers to *WebOb* for its request and response implementations, and handling form submission data is a property of the request implementation. Understanding *WebOb*’s request API is the key to understanding how to process form submission data.

There are some defaults that you need to be aware of when trying to handle form submission data in a Pyramid view. Having high-order (i.e., non-ASCII) characters in data contained within form submissions is exceedingly common, and the UTF-8 encoding is the most common encoding used on the web for character data. Since Unicode values are much saner than working with and storing bytestrings, Pyramid configures the *WebOb* request machinery to attempt to decode form submission values into Unicode from UTF-8 implicitly. This implicit decoding happens when view code obtains form field values via the `request.params`, `request.GET`, or `request.POST` APIs (see *pyramid.request* for details about these APIs).



Many people find the difference between Unicode and UTF-8 confusing. Unicode is a standard for representing text that supports most of the world’s writing systems. However, there are many ways that Unicode data can be encoded into bytes for transit and storage. UTF-8 is a specific encoding for Unicode that is backwards-compatible with ASCII. This makes UTF-8 very convenient for encoding data where a large subset of that data is ASCII characters, which is largely true on the web. UTF-8 is also the standard character encoding for URLs.

As an example, let’s assume that the following form page is served up to a browser client, and its `action` points at some Pyramid view code:

```

1 <html xmlns="http://www.w3.org/1999/xhtml">
2   <head>
3     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
4   </head>
5   <form method="POST" action="myview">
6     <div>
7       <input type="text" name="firstname"/>
8     </div>
9     <div>
10      <input type="text" name="lastname"/>
11    </div>
12    <input type="submit" value="Submit"/>
13  </form>
14 </html>

```

The `myview` view code in the Pyramid application *must* expect that the values returned by `request.params` will be of type `unicode`, as opposed to type `str`. The following will work to accept a form post from the above form:

```

1 def myview(request):
2     firstname = request.params['firstname']
3     lastname = request.params['lastname']

```

But the following `myview` view code *may not* work, as it tries to decode already-decoded (`unicode`) values obtained from `request.params`:

```

1 def myview(request):
2     # the .decode('utf-8') will break below if there are any high-order
3     # characters in the firstname or lastname
4     firstname = request.params['firstname'].decode('utf-8')
5     lastname = request.params['lastname'].decode('utf-8')

```

For implicit decoding to work reliably, you should ensure that every form you render that posts to a Pyramid view explicitly defines a charset encoding of UTF-8. This can be done via a response that has a `charset=UTF-8` in its `Content-Type` header; or, as in the form above, with a `meta http-equiv` tag that implies that the charset is UTF-8 within the HTML head of the page containing the form. This must be done explicitly because all known browser clients assume that they should encode form data in the same character set implied by the `Content-Type` value of the response containing the form when subsequently submitting that form. There is no other generally accepted way to tell browser clients which charset to use to encode form data. If you do not specify an encoding explicitly, the browser client will choose to encode form data in its default character set before submitting it, which may not be UTF-8 as the server expects. If a request containing form data encoded in a non-UTF-8 charset is handled

by your view code, eventually the request code accessed within your view will throw an error when it can't decode some high-order character encoded in another character set within form data, e.g., when `request.params['somename']` is accessed.

If you are using the `Response` class to generate a response, or if you use the `render_template_*` templating APIs, the UTF-8 charset is set automatically as the default via the `Content-Type` header. If you return a `Content-Type` header without an explicit charset, a request will add a `; charset=utf-8` trailer to the `Content-Type` header value for you for response content types that are textual (e.g., `text/html` or `application/xml`) as it is rendered. If you are using your own response object, you will need to ensure you do this yourself.

i Only the *values* of request params obtained via `request.params`, `request.GET` or `request.POST` are decoded to Unicode objects implicitly in the Pyramid default configuration. The keys are still (byte) strings.

Alternate View Callable Argument/Calling Conventions

Usually view callables are defined to accept only a single argument: `request`. However, view callables may alternately be defined as classes, functions, or any callable that accept *two* positional arguments: a *context* resource as the first argument and a *request* as the second argument.

The *context* and *request* arguments passed to a view function defined in this style can be defined as follows:

context The *resource* object found via tree *traversal* or *URL dispatch*.

request A Pyramid Request object representing the current WSGI request.

The following types work as view callables in this style:

1. Functions that accept two arguments: `context` and `request`, e.g.:

```
1 from pyramid.response import Response
2
3 def view(context, request):
4     return Response('OK')
```

2. Classes that have an `__init__` method that accepts `context`, `request`, and a `__call__` method which accepts no arguments, e.g.:

```

1 from pyramid.response import Response
2
3 class view(object):
4     def __init__(self, context, request):
5         self.context = context
6         self.request = request
7
8     def __call__(self):
9         return Response('OK')

```

3. Arbitrary callables that have a `__call__` method that accepts `context`, `request`, e.g.:

```

1 from pyramid.response import Response
2
3 class View(object):
4     def __call__(self, context, request):
5         return Response('OK')
6 view = View() # this is the view callable

```

This style of calling convention is most useful for *traversal* based applications, where the context object is frequently used within the view callable code itself.

No matter which view calling convention is used, the view code always has access to the context via `request.context`.

Passing Configuration Variables to a View

For information on passing a variable from the configuration `.ini` files to a view, see *Deployment Settings*.

Pylons-1.0-Style “Controller” Dispatch

A package named *pyramid_handlers* (available from PyPI) provides an analogue of *Pylons*-style “controllers”, which are a special kind of view class which provides more automation when your application uses *URL dispatch* solely.

Renderers

A view callable needn’t *always* return a *Response* object. If a view happens to return something which does not implement the Pyramid Response interface, Pyramid will attempt to use a *renderer* to construct a response. For example:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='json')
4 def hello_world(request):
5     return {'content': 'Hello!'}
```

The above example returns a *dictionary* from the view callable. A dictionary does not implement the Pyramid response interface, so you might believe that this example would fail. However, since a `renderer` is associated with the view callable through its *view configuration* (in this case, using a `renderer` argument passed to `view_config()`), if the view does *not* return a Response object, the renderer will attempt to convert the result of the view to a response on the developer's behalf.

Of course, if no renderer is associated with a view's configuration, returning anything except an object which implements the Response interface will result in an error. And, if a renderer *is* used, whatever is returned by the view must be compatible with the particular kind of renderer used, or an error may occur during view invocation.

One exception exists: it is *always* OK to return a Response object, even when a `renderer` is configured. In such cases, the renderer is bypassed entirely.

Various types of renderers exist, including serialization renderers and renderers which use templating systems.

Writing View Callables Which Use a Renderer

As we've seen, a view callable needn't always return a Response object. Instead, it may return an arbitrary Python object, with the expectation that a *renderer* will convert that object into a response instance on your behalf. Some renderers use a templating system, while other renderers use object serialization techniques. In practice, renderers obtain application data values from Python dictionaries so, in practice, view callables which use renderers return Python dictionaries.

View callables can *explicitly* call renderers, but typically don't. Instead view configuration declares the renderer used to render a view callable's results. This is done with the `renderer` attribute. For example, this call to `add_view()` associates the `json` renderer with a view callable:

```
config.add_view('myproject.views.my_view', renderer='json')
```

When this configuration is added to an application, the `myproject.views.my_view` view callable will now use a `json` renderer, which renders view return values to a *JSON* response serialization.

Pyramid defines several *Built-in Renderers*, and additional renderers can be added by developers to the system as necessary. See *Adding and Changing Renderers*.

Views which use a renderer and return a non-Response value can vary non-body response attributes (such as headers and the HTTP status code) by attaching a property to the `request.response` attribute. See *Varying Attributes of Rendered Responses*.

As already mentioned, if the *view callable* associated with a *view configuration* returns a Response object (or its instance), any renderer associated with the view configuration is ignored, and the response is passed back to Pyramid unchanged. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(renderer='json')
5 def view(request):
6     return Response('OK') # json renderer avoided
```

Likewise for an *HTTP exception* response:


```
1 from pyramid.httpexceptions import HTTPFound
2 from pyramid.view import view_config
3
4 @view_config(renderer='json')
5 def view(request):
6     return HTTPFound(location='http://example.com') # json renderer avoided
```

You can of course also return the `request.response` attribute instead to avoid rendering:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='json')
4 def view(request):
5     request.response.body = 'OK'
6     return request.response # json renderer avoided
```

Built-in Renderers

Several built-in renderers exist in Pyramid. These renderers can be used in the `renderer` attribute of view configurations.

 Bindings for officially supported templating languages can be found at *Available Add-On Template System Bindings*.

string: String Renderer

The `string` renderer renders a view callable result to a string. If a view callable returns a non-Response object, and the `string` renderer is associated in that view's configuration, the result will be to run the object through the Python `str` function to generate a string. Note that if a Unicode object is returned by the view callable, it is not `str()`-ified.

Here's an example of a view that returns a dictionary. If the `string` renderer is specified in the configuration for this view, the view will render the returned dictionary to the `str()` representation of the dictionary:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='string')
4 def hello_world(request):
5     return {'content': 'Hello!'}
```

The body of the response returned by such a view will be a string representing the `str()` serialization of the return value:

```
{'content': 'Hello!'}
```

Views which use the `string` renderer can vary non-body response attributes by using the API of the `request.response` attribute. See *Varying Attributes of Rendered Responses*.

JSON Renderer

The `json` renderer renders view callable results to *JSON*. By default, it passes the return value through the `json.dumps` standard library function, and wraps the result in a response object. It also sets the response content-type to `application/json`.

Here's an example of a view that returns a dictionary. Since the `json` renderer is specified in the configuration for this view, the view will render the returned dictionary to a JSON serialization:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='json')
4 def hello_world(request):
5     return {'content': 'Hello!'}
```

The body of the response returned by such a view will be a string representing the JSON serialization of the return value:

```
{"content": "Hello!"}
```

The return value needn't be a dictionary, but the return value must contain values serializable by the configured serializer (by default `json.dumps`).

You can configure a view to use the JSON renderer by naming `json` as the `renderer` argument of a view configuration, e.g., by using `add_view()`:

```
1 config.add_view('myproject.views.hello_world',
2                 name='hello',
3                 context='myproject.resources.Hello',
4                 renderer='json')
```

Views which use the JSON renderer can vary non-body response attributes by using the API of the `request.response` attribute. See *Varying Attributes of Rendered Responses*.

Serializing Custom Objects

Some objects are not, by default, JSON-serializable (such as datetimes and other arbitrary Python objects). You can, however, register code that makes non-serializable objects serializable in two ways:

- Define a `__json__` method on objects in your application.
- For objects you don't "own", you can register a JSON renderer that knows about an *adapter* for that kind of object.

Using a Custom `__json__` Method

Custom objects can be made easily JSON-serializable in Pyramid by defining a `__json__` method on the object's class. This method should return values natively JSON-serializable (such as ints, lists, dictionaries, strings, and so forth). It should accept a single additional argument, `request`, which will be the active request object at render time.

```
1 from pyramid.view import view_config
2
3 class MyObject(object):
4     def __init__(self, x):
5         self.x = x
6
7     def __json__(self, request):
8         return {'x':self.x}
9
10 @view_config(renderer='json')
11 def objects(request):
12     return [MyObject(1), MyObject(2)]
13
14 # the JSON value returned by ``objects`` will be:
15 # [{"x": 1}, {"x": 2}]
```

Using the `add_adapter` Method of a Custom JSON Renderer

If you aren't the author of the objects being serialized, it won't be possible (or at least not reasonable) to add a custom `__json__` method to their classes in order to influence serialization. If the object passed to the renderer is not a serializable type and has no `__json__` method, usually a `TypeError` will be raised during serialization. You can change this behavior by creating a custom JSON renderer and adding adapters to handle custom types. The renderer will attempt to adapt non-serializable objects using the registered adapters. A short example follows:

```
1 from pyramid.renderers import JSON
2
3 if __name__ == '__main__':
4     config = Configurator()
5     json_renderer = JSON()
6     def datetime_adapter(obj, request):
7         return obj.isoformat()
8     json_renderer.add_adapter(datetime.datetime, datetime_adapter)
9     config.add_renderer('json', json_renderer)
```

The `add_adapter` method should accept two arguments: the *class* of the object that you want this adapter to run for (in the example above, `datetime.datetime`), and the adapter itself.

The adapter should be a callable. It should accept two arguments: the object needing to be serialized and `request`, which will be the current request object at render time. The adapter should raise a `TypeError` if it can't determine what to do with the object.

See `pyramid.renderers.JSON` and *Adding and Changing Renderers* for more information.

New in version 1.4: Serializing custom objects.

JSONP Renderer

New in version 1.1.

`pyramid.renderers.JSONP` is a JSONP renderer factory helper which implements a hybrid JSON/JSONP renderer. JSONP is useful for making cross-domain AJAX requests.

Unlike other renderers, a JSONP renderer needs to be configured at startup time “by hand”. Configure a JSONP renderer using the `pyramid.config.Configurator.add_renderer()` method:

```
from pyramid.config import Configurator
from pyramid.renderers import JSONP

config = Configurator()
config.add_renderer('jsonp', JSONP(param_name='callback'))
```

Once this renderer is registered via `add_renderer()` as above, you can use `jsonp` as the `renderer=` parameter to `@view_config` or `pyramid.config.Configurator.add_view()`:

```
from pyramid.view import view_config

@view_config(renderer='jsonp')
def myview(request):
    return {'greeting': 'Hello world'}
```

When a view is called that uses a JSONP renderer:

- If there is a parameter in the request's HTTP query string (aka `request.GET`) that matches the `param_name` of the registered JSONP renderer (by default, `callback`), the renderer will return a JSONP response.

- If there is no callback parameter in the request's query string, the renderer will return a “plain” JSON response.

Javascript library AJAX functionality will help you make JSONP requests. For example, JQuery has a `getJSON` function, and has equivalent (but more complicated) functionality in its `ajax` function.

For example (JavaScript):

```
var api_url = 'http://api.geonames.org/timezoneJSON' +
              '?lat=38.301733840000004' +
              '&lng=-77.45869621' +
              '&username=fred' +
              '&callback=?';
jqxhr = $.getJSON(api_url);
```

The string `callback=?` above in the url param to the JQuery `getJSON` function indicates to jQuery that the query should be made as a JSONP request; the `callback` parameter will be automatically filled in for you and used.

The same custom-object serialization scheme defined used for a “normal” JSON renderer in *Serializing Custom Objects* can be used when passing values to a JSONP renderer too.

Varying Attributes of Rendered Responses

Before a response constructed by a *renderer* is returned to Pyramid, several attributes of the request are examined which have the potential to influence response behavior.

View callables that don't directly return a response should use the API of the *pyramid.response.Response* attribute, available as `request.response` during their execution, to influence associated response behavior.

For example, if you need to change the response status from within a view callable that uses a renderer, assign the `status` attribute to the `response` attribute of the request before returning a result:

```
1 from pyramid.view import view_config
2
3 @view_config(name='gone', renderer='templates/gone.pt')
4 def myview(request):
5     request.response.status = '404 Not Found'
6     return {'URL': request.URL}
```

Note that mutations of `request.response` in views which return a `Response` object directly will have no effect unless the response object returned *is* `request.response`. For example, the following example calls `request.response.set_cookie`, but this call will have no effect because a different `Response` object is returned.

```
1 from pyramid.response import Response
2
3 def view(request):
4     request.response.set_cookie('abc', '123') # this has no effect
5     return Response('OK') # because we're returning a different response
```

If you mutate `request.response` and you'd like the mutations to have an effect, you must return `request.response`:

```
1 def view(request):
2     request.response.set_cookie('abc', '123')
3     return request.response
```

For more information on attributes of the request, see the API documentation in `pyramid.request`. For more information on the API of `request.response`, see `pyramid.request.Request.response`.

Adding and Changing Renderers

New templating systems and serializers can be associated with Pyramid renderer names. To this end, configuration declarations can be made which change an existing *renderer factory*, and which add a new renderer factory.

Renderers can be registered imperatively using the `pyramid.config.Configurator.add_renderer()` API.

For example, to add a renderer which renders views which have a `renderer` attribute that is a path that ends in `.jinja2`:

```
config.add_renderer('.jinja2', 'mypackage.MyJinja2Renderer')
```

The first argument is the renderer name. The second argument is a reference to an implementation of a *renderer factory* or a *dotted Python name* referring to such an object.

Adding a New Renderer

You may add a new renderer by creating and registering a *renderer factory*.

A renderer factory implementation should conform to the `pyramid.interfaces.IRendererFactory` interface. It should be capable of creating an object that conforms to the `pyramid.interfaces.IRenderer` interface. A typical class that follows this setup is as follows:

```
1 class RendererFactory:
2     def __init__(self, info):
3         """ Constructor: info will be an object having the
4           following attributes: name (the renderer name), package
5           (the package that was 'current' at the time the
6           renderer was registered), type (the renderer type
7           name), registry (the current application registry) and
8           settings (the deployment settings dictionary). """
9
10    def __call__(self, value, system):
11        """ Call the renderer implementation with the value
12          and the system value passed in as arguments and return
13          the result (a string or unicode object). The value is
14          the return value of a view. The system value is a
15          dictionary containing available system values
16          (e.g., view, context, and request). """
```

The formal interface definition of the `info` object passed to a renderer factory constructor is available as `pyramid.interfaces.IRendererInfo`.

There are essentially two different kinds of renderer factories:

- A renderer factory which expects to accept an *asset specification*, or an absolute path, as the `name` attribute of the `info` object fed to its constructor. These renderer factories are registered with a `name` value that begins with a dot (`.`). These types of renderer factories usually relate to a file on the filesystem, such as a template.
- A renderer factory which expects to accept a token that does not represent a filesystem path or an asset specification in the `name` attribute of the `info` object fed to its constructor. These renderer factories are registered with a `name` value that does not begin with a dot. These renderer factories are typically object serializers.

Asset Specifications

An asset specification is a colon-delimited identifier for an *asset*. The colon separates a Python *package* name from a package subpath. For example, the asset specification `my.package:static/baz.css` identifies the file named `baz.css` in the `static` subdirectory of the `my.package` Python *package*.

Here's an example of the registration of a simple renderer factory via `add_renderer()`, where `config` is an instance of `pyramid.config.Configurator()`:

```
config.add_renderer(name='amf', factory='my.package.MyAMFRenderer')
```

Adding the above code to your application startup configuration will allow you to use the `my.package.MyAMFRenderer` renderer factory implementation in view configurations. Your application can use this renderer by specifying `amf` in the `renderer` attribute of a *view configuration*:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='amf')
4 def myview(request):
5     return {'Hello': 'world'}
```

At startup time, when a *view configuration* is encountered which has a `name` attribute that does not contain a dot, the full name value is used to construct a renderer from the associated renderer factory. In this case, the view configuration will create an instance of an `MyAMFRenderer` for each view configuration which includes `amf` as its `renderer` value. The name passed to the `MyAMFRenderer` constructor will always be `amf`.

Here's an example of the registration of a more complicated renderer factory, which expects to be passed a filesystem path:

```
config.add_renderer(name='.jinja2', factory='my.package.MyJinja2Renderer')
```

Adding the above code to your application startup will allow you to use the `my.package.MyJinja2Renderer` renderer factory implementation in view configurations by referring to any renderer which *ends in* `.jinja2` in the `renderer` attribute of a *view configuration*:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='templates/mytemplate.jinja2')
4 def myview(request):
5     return {'Hello': 'world'}
```

When a *view configuration* is encountered at startup time which has a `name` attribute that does contain a dot, the value of the `name` attribute is split on its final dot. The second element of the split is typically the filename extension. This extension is used to look up a renderer factory for the configured view. Then the value of `renderer` is passed to the factory to create a renderer for the view. In this case, the view configuration will create an instance of a `MyJinja2Renderer` for each view configuration which includes anything ending with `.jinja2` in its `renderer` value. The name passed to the `MyJinja2Renderer` constructor will be the full value that was set as `renderer=` in the view configuration.

Adding a Default Renderer

To associate a *default* renderer with *all* view configurations (even ones which do not possess a `renderer` attribute), pass `None` as the `name` attribute to the `renderer` tag:

```
config.add_renderer(None, 'mypackage.json_renderer_factory')
```

Changing an Existing Renderer

Pyramid supports overriding almost every aspect of its setup through its *Conflict Resolution* mechanism. This means that, in most cases, overriding a renderer is as simple as using the `pyramid.config.Configurator.add_renderer()` method to redefine the template extension. For example, if you would like to override the `json` renderer to specify a new renderer, you could do the following:

```
json_renderer = pyramid.renderers.JSON()
config.add_renderer('json', json_renderer)
```

After doing this, any views registered with the `json` renderer will use the new renderer.

Overriding a Renderer at Runtime



This is an advanced feature, not typically used by “civilians”.

In some circumstances, it is necessary to instruct the system to ignore the static renderer declaration provided by the developer in view configuration, replacing the renderer with another *after a request starts*. For example, an “omnipresent” XML-RPC implementation that detects that the request is from an XML-RPC client might override a view configuration statement made by the user instructing the view to use a template renderer with one that uses an XML-RPC renderer. This renderer would produce an XML-RPC representation of the data returned by an arbitrary view callable.

To use this feature, create a *NewRequest subscriber* which sniffs at the request data and which conditionally sets an `override_renderer` attribute on the request itself, which in turn is the *name* of a registered renderer. For example:

```
1 from pyramid.events import subscriber
2 from pyramid.events import NewRequest
3
4 @subscriber(NewRequest)
5 def set_xmlrpc_params(event):
6     request = event.request
7     if (request.content_type == 'text/xml'
8         and request.method == 'POST'
9         and not 'soapaction' in request.headers
10        and not 'x-pyramid-avoid-xmlrpc' in request.headers):
11         params, method = parse_xmlrpc_request(request)
12         request.xmlrpc_params, request.xmlrpc_method = params, method
13         request.is_xmlrpc = True
14         request.override_renderer = 'xmlrpc'
15     return True
```

The result of such a subscriber will be to replace any existing static renderer configured by the developer with a (notional, nonexistent) XML-RPC renderer, if the request appears to come from an XML-RPC client.

Templates

A *template* is a file on disk which can be used to render dynamic data provided by a *view*. Pyramid offers a number of ways to perform templating tasks out of the box, and provides add-on templating support through a set of bindings packages.

Before discussing how built-in templates are used in detail, we’ll discuss two ways to render templates within Pyramid in general: directly and via renderer configuration.

Using Templates Directly

The most straightforward way to use a template within Pyramid is to cause it to be rendered directly within a *view callable*. You may use whatever API is supplied by a given templating engine to do so.

Pyramid provides various APIs that allow you to render templates directly from within a view callable. For example, if there is a *Chameleon* ZPT template named `foo.pt` in a directory named `templates` in your application, you can render the template from within the body of a view callable like so:

```
1 from pyramid.renderers import render_to_response
2
3 def sample_view(request):
4     return render_to_response('templates/foo.pt',
5                               {'foo':1, 'bar':2},
6                               request=request)
```

The `sample_view` view callable function above returns a *response* object which contains the body of the `templates/foo.pt` template. In this case, the `templates` directory should live in the same directory as the module containing the `sample_view` function. The template author will have the names `foo` and `bar` available as top-level names for replacement or comparison purposes.

In the example above, the path `templates/foo.pt` is relative to the directory containing the file which defines the view configuration. In this case, this is the directory containing the file that defines the `sample_view` function. Although a renderer path is usually just a simple relative path-name, a path named as a renderer can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternatively be an *asset specification* in the form `some.dotted.package_name:relative/path`. This makes it possible to address template assets which live in another package. For example:

```
1 from pyramid.renderers import render_to_response
2
3 def sample_view(request):
4     return render_to_response('mypackage:templates/foo.pt',
5                               {'foo':1, 'bar':2},
6                               request=request)
```

An asset specification points at a file within a Python *package*. In this case, it points at a file named `foo.pt` within the `templates` directory of the `mypackage` package. Using an asset specification instead of a relative template name is usually a good idea, because calls to `render_to_response()` using asset specifications will continue to work properly if you move the code containing them to another location.

In the examples above we pass in a keyword argument named `request` representing the current Pyramid request. Passing a request keyword argument will cause the `render_to_response` function to supply the renderer with more correct system values (see *System Values Used During Rendering*), because most of the information required to compose proper system values is present in the request. If your template relies on the name `request` or `context`, or if you’ve configured special *renderer globals*, make sure to pass `request` as a keyword argument in every call to a `pyramid.renderers.render_*` function.

Every view must return a *response* object, except for views which use a *renderer* named via view configuration (which we’ll see shortly). The `pyramid.renderers.render_to_response()` function is a shortcut function that actually returns a response object. This allows the example view above to simply return the result of its call to `render_to_response()` directly.

Obviously not all APIs you might call to get response data will return a response object. For example, you might render one or more templates to a string that you want to use as response data. The `pyramid.renderers.render()` API renders a template to a string. We can manufacture a *response* object directly, and use that string as the body of the response:

```
1 from pyramid.renderers import render
2 from pyramid.response import Response
3
4 def sample_view(request):
5     result = render('mypackage:templates/foo.pt',
6                   {'foo':1, 'bar':2},
7                   request=request)
8     response = Response(result)
9     return response
```

Because *view callable* functions are typically the only code in Pyramid that need to know anything about templates, and because view functions are very simple Python, you can use whatever templating system with which you’re most comfortable within Pyramid. Install the templating system, import its API functions into your views module, use those APIs to generate a string, then return that string as the body of a Pyramid *Response* object.

For example, here’s an example of using “raw” Mako from within a Pyramid *view*:

```
1 from mako.template import Template
2 from pyramid.response import Response
3
4 def make_view(request):
5     template = Template(filename='/templates/template.mak')
6     result = template.render(name=request.params['name'])
7     response = Response(result)
8     return response
```

You probably wouldn't use this particular snippet in a project, because it's easier to use the supported *Mako bindings*. But if your favorite templating system is not supported as a renderer extension for Pyramid, you can create your own simple combination as shown above.



If you use third-party templating languages without cooperating Pyramid bindings directly within view callables, the auto-template-reload strategy explained in *Automatically Reloading Templates* will not be available, nor will the template asset overriding capability explained in *Overriding Assets* be available, nor will it be possible to use any template using that language as a *renderer*. However, it's reasonably easy to write custom templating system binding packages for use under Pyramid so that templates written in the language can be used as renderers. See *Adding and Changing Renderers* for instructions on how to create your own template renderer and *Available Add-On Template System Bindings* for example packages.

If you need more control over the status code and content-type, or other response attributes from views that use direct templating, you may set attributes on the response that influence these values.

Here's an example of changing the content-type and status of the response object returned by `render_to_response()`:

```
1 from pyramid.renderers import render_to_response
2
3 def sample_view(request):
4     response = render_to_response('templates/foo.pt',
5                                 {'foo':1, 'bar':2},
6                                 request=request)
7     response.content_type = 'text/plain'
8     response.status_int = 204
9     return response
```

Here's an example of manufacturing a response object using the result of `render()` (a string):

```
1 from pyramid.renderers import render
2 from pyramid.response import Response
3
4 def sample_view(request):
5     result = render('mypackage:templates/foo.pt',
6                   {'foo':1, 'bar':2},
7                   request=request)
8     response = Response(result)
9     response.content_type = 'text/plain'
10    return response
```

System Values Used During Rendering

When a template is rendered using `render_to_response()` or `render()`, or a `renderer=` argument to view configuration (see *Templates Used as Renderers via Configuration*), the renderer representing the template will be provided with a number of *system* values. These values are provided to the template:

request The value provided as the `request` keyword argument to `render_to_response` or `render` or the request object passed to the view when the `renderer=` argument to view configuration is being used to render the template.

req An alias for `request`.

context The current Pyramid *context* if `request` was provided as a keyword argument to `render_to_response` or `render`, or `None` if the `request` keyword argument was not provided. This value will always be provided if the template is rendered as the result of a `renderer=` argument to the view configuration being used.

renderer_name The renderer name used to perform the rendering, e.g., `mypackage:templates/foo.pt`.

renderer_info An object implementing the `pyramid.interfaces.IRendererInfo` interface. Basically, an object with the following attributes: `name`, `package`, and `type`.

view The view callable object that was used to render this template. If the view callable is a method of a class-based view, this will be an instance of the class that the method was defined on. If the view callable is a function or instance, it will be that function or instance. Note that this value will only be automatically present when a template is rendered as a result of a `renderer=` argument; it will be `None` when the `render_to_response` or `render` APIs are used.

You can define more values which will be passed to every template executed as a result of rendering by defining *renderer globals*.

What any particular renderer does with these system values is up to the renderer itself, but most template renderers make these names available as top-level template variables.

Templates Used as Renderers via Configuration

An alternative to using `render_to_response()` to render templates manually in your view callable code is to specify the template as a *renderer* in your *view configuration*. This can be done with any of the templating languages supported by Pyramid.

To use a renderer via view configuration, specify a template *asset specification* as the `renderer` argument, or attribute to the *view configuration* of a *view callable*. Then return a *dictionary* from that view callable. The dictionary items returned by the view callable will be made available to the renderer template as top-level names.

The association of a template as a renderer for a *view configuration* makes it possible to replace code within a *view callable* that handles the rendering of a template.

Here's an example of using a `view_config` decorator to specify a *view configuration* that names a template renderer:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='templates/foo.pt')
4 def my_view(request):
5     return {'foo':1, 'bar':2}
```



You do not need to supply the `request` value as a key in the dictionary result returned from a renderer-configured view callable. Pyramid automatically supplies this value for you, so that the “most correct” system values are provided to the renderer.



The `renderer` argument to the `@view_config` configuration decorator shown above is the template *path*. In the example above, the path `templates/foo.pt` is *relative*. Relative to what, you ask? Because we're using a Chameleon renderer, it means “relative to the directory in which the file that defines the view configuration lives”. In this case, this is the directory containing the file that defines the `my_view` function.

Similar renderer configuration can be done imperatively. See *Writing View Callables Which Use a Renderer*.

See also:

See also *Built-in Renderers*.

Although a renderer path is usually just a simple relative pathname, a path named as a renderer can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternatively be an *asset specification* in the form `some.dotted.package_name:relative/path`, making it possible to address template assets which live in another package.

Not just any template from any arbitrary templating system may be used as a renderer. Bindings must exist specifically for Pyramid to use a templating language template as a renderer.

Why Use a Renderer via View Configuration

Using a renderer in view configuration is usually a better way to render templates than using any rendering API directly from within a *view callable* because it makes the view callable more unit-testable. Views which use templating or rendering APIs directly must return a *Response* object. Making testing assertions about response objects is typically an indirect process, because it means that your test code often needs to somehow parse information out of the response body (often HTML). View callables configured with renderers externally via view configuration typically return a dictionary, as above. Making assertions about results returned in a dictionary is almost always more direct and straightforward than needing to parse HTML.

By default, views rendered via a template renderer return a *Response* object which has a *status code* of 200 OK, and a *content-type* of `text/html`. To vary attributes of the response of a view that uses a renderer, such as the content-type, headers, or status attributes, you must use the API of the *pyramid.response.Response* object exposed as `request.response` within the view before returning the dictionary. See *Varying Attributes of Rendered Responses* for more information.

The same set of system values are provided to templates rendered via a renderer view configuration as those provided to templates rendered imperatively. See *System Values Used During Rendering*.

Debugging Templates

A `NameError` exception resulting from rendering a template with an undefined variable (e.g. `${wrong}`) might end up looking like this:

```
RuntimeError: Caught exception rendering template.
- Expression: ``wrong``
- Filename:   /home/fred/env/proj/proj/templates/mytemplate.pt
- Arguments:  renderer_name: proj:templates/mytemplate.pt
```

```
template: <PageTemplateFile - at 0x1d2ecf0>
xincludes: <XIncludes - at 0x1d3a130>
request: <Request - at 0x1d2ecd0>
project: proj
macros: <Macros - at 0x1d3aed0>
context: <MyResource None at 0x1d39130>
view: <function my_view at 0x1d23570>
```

```
NameError: wrong
```

The output tells you which template the error occurred in, as well as displaying the arguments passed to the template itself.

Automatically Reloading Templates

It's often convenient to see changes you make to a template file appear immediately without needing to restart the application process. Pyramid allows you to configure your application development environment so that a change to a template will be automatically detected, and the template will be reloaded on the next rendering.



Auto-template-reload behavior is not recommended for production sites as it slows rendering slightly; it's usually only desirable during development.

In order to turn on automatic reloading of templates, you can use an environment variable or a configuration file setting.

To use an environment variable, start your application under a shell using the `PYRAMID_RELOAD_TEMPLATES` operating system environment variable set to 1. For example:

```
$ PYRAMID_RELOAD_TEMPLATES=1 $VENV/bin/pserve myproject.ini
```

To use a setting in the application `.ini` file for the same purpose, set the `pyramid.reload_templates` key to `true` within the application's configuration section, e.g.:

```
1 [app:main]
2 use = egg:MyProject
3 pyramid.reload_templates = true
```

Available Add-On Template System Bindings

The Pylons Project maintains several packages providing bindings to different templating languages including the following:

Template Language	Pyramid Bindings	Default Extensions
Chameleon	pyramid_chameleon	.pt, .txt
Jinja2	pyramid_jinja2	.jinja2
Mako	pyramid_mako	.mak, .mako

View Configuration

View lookup is the Pyramid subsystem responsible for finding and invoking a *view callable*. *View configuration* controls how *view lookup* operates in your application. During any given request, view configuration information is compared against request data by the view lookup subsystem in order to find the “best” view callable for that request.

In earlier chapters, you have been exposed to a few simple view configuration declarations without much explanation. In this chapter we will explore the subject in detail.

Mapping a Resource or URL Pattern to a View Callable

A developer makes a *view callable* available for use within a Pyramid application via *view configuration*. A view configuration associates a view callable with a set of statements that determine the set of circumstances which must be true for the view callable to be invoked.

A view configuration statement is made about information present in the *context* resource and the *request*.

View configuration is performed in one of two ways:

- By running a *scan* against application source code which has a `pyramid.view.view_config` decorator attached to a Python object as per *Adding View Configuration Using the @view_config Decorator*.
- By using the `pyramid.config.Configurator.add_view()` method as per *Adding View Configuration Using add_view()*.

View Configuration Parameters

All forms of view configuration accept the same general types of arguments.

Many arguments supplied during view configuration are *view predicate* arguments. View predicate arguments used during view configuration are used to narrow the set of circumstances in which *view lookup* will find a particular view callable.

View predicate attributes are an important part of view configuration that enables the *view lookup* subsystem to find and invoke the appropriate view. The greater the number of predicate attributes possessed by a view's configuration, the more specific the circumstances need to be before the registered view callable will be invoked. The fewer the number of predicates which are supplied to a particular view configuration, the more likely it is that the associated view callable will be invoked. A view with five predicates will always be found and evaluated before a view with two, for example.

This does not mean however, that Pyramid “stops looking” when it finds a view registration with predicates that don't match. If one set of view predicates does not match, the “next most specific” view (if any) is consulted for predicates, and so on, until a view is found, or no view can be matched up with the request. The first view with a set of predicates all of which match the request environment will be invoked.

If no view can be found with predicates which allow it to be matched up with the request, Pyramid will return an error to the user's browser, representing a “not found” (404) page. See *Changing the Not Found View* for more information about changing the default *Not Found View*.

Other view configuration arguments are non-predicate arguments. These tend to modify the response of the view callable or prevent the view callable from being invoked due to an authorization policy. The presence of non-predicate arguments in a view configuration does not narrow the circumstances in which the view callable will be invoked.

Non-Predicate Arguments

permission The name of a *permission* that the user must possess in order to invoke the *view callable*. See *Configuring View Security* for more information about view security and permissions.

If `permission` is not supplied, no permission is registered for this view (it's accessible by any caller).

attr The view machinery defaults to using the `__call__` method of the *view callable* (or the function itself, if the view callable is a function) to obtain a response. The `attr` value allows you to vary the method attribute used to obtain the response. For example, if your view was a class, and the class has a method named `index` and you wanted to use this method instead of the class's `__call__` method to return the response, you'd say `attr="index"` in the view configuration for the view. This is most useful when the view definition is a class.

If `attr` is not supplied, `None` is used (implying the function itself if the view is a function, or the `__call__` callable attribute if the view is a class).

renderer Denotes the *renderer* implementation which will be used to construct a *response* from the associated view callable's return value.

See also:

See also *Renderers*.

This is either a single string term (e.g., `json`) or a string implying a path or *asset specification* (e.g., `templates/views.pt`) naming a *renderer* implementation. If the `renderer` value does not contain a dot (`.`), the specified string will be used to look up a *renderer* implementation, and that *renderer* implementation will be used to construct a response from the view return value. If the `renderer` value contains a dot (`.`), the specified term will be treated as a path, and the filename extension of the last element in the path will be used to look up the *renderer* implementation, which will be passed the full path.

When the `renderer` is a path—although a path is usually just a simple relative pathname (e.g., `templates/foo.pt`, implying that a template named “foo.pt” is in the “templates” directory relative to the directory of the current *package*)—the path can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternatively be a *asset specification* in the form `some.dotted.package_name:relative/path`, making it possible to address template assets which live in a separate package.

The `renderer` attribute is optional. If it is not defined, the “null” *renderer* is assumed (no rendering is performed and the value is passed back to the upstream Pyramid machinery unchanged). Note that if the view callable itself returns a *response* (see *View Callable Responses*), the specified *renderer* implementation is never called.

http_cache When you supply an `http_cache` value to a view configuration, the `Expires` and `Cache-Control` headers of a response generated by the associated view callable are modified. The value for `http_cache` may be one of the following:

- A nonzero integer. If it's a nonzero integer, it's treated as a number of seconds. This number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=3600` instructs the requesting browser to ‘cache this response for an hour, please’.

- A `datetime.timedelta` instance. If it's a `datetime.timedelta` instance, it will be converted into a number of seconds, and that number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=datetime.timedelta(days=1)` instructs the requesting browser to 'cache this response for a day, please'.
- Zero (0). If the value is zero, the `Cache-Control` and `Expires` headers present in all responses from this view will be composed such that client browser cache (and any intermediate caches) are instructed to never cache the response.
- A two-tuple. If it's a two-tuple (e.g., `http_cache=(1, {'public':True})`), the first value in the tuple may be a nonzero integer or a `datetime.timedelta` instance. In either case this value will be used as the number of seconds to cache the response. The second value in the tuple must be a dictionary. The values present in the dictionary will be used as input to the `Cache-Control` response header. For example: `http_cache=(3600, {'public':True})` means 'cache for an hour, and add public to the `Cache-Control` header of the response'. All keys and values supported by the `webob.cachecontrol.CacheControl` interface may be added to the dictionary. Supplying `{'public':True}` is equivalent to calling `response.cache_control.public = True`.

Providing a non-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value)` within your view's body.

Providing a two-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value[0], **value[1])` within your view's body.

If you wish to avoid influencing the `Expires` header, and instead wish to only influence `Cache-Control` headers, pass a tuple as `http_cache` with the first element of `None`, i.e., `(None, {'public':True})`.

`require_csrf`

CSRF checks will affect any request method that is not defined as a "safe" method by RFC2616. In practice this means that `GET`, `HEAD`, `OPTIONS`, and `TRACE` methods will pass untouched and all other methods will require CSRF. This option is used in combination with the `pyramid.require_default_csrf` setting to control which request parameters are checked for CSRF tokens.

This feature requires a configured *session factory*.

If this option is set to `True` then CSRF checks will be enabled for `POST` requests to this view. The required token will be whatever was specified by the `pyramid.require_default_csrf` setting, or will fallback to `csrf_token`.

If this option is set to a string then CSRF checks will be enabled and it will be used as the required token regardless of the `pyramid.require_default_csrf` setting.

If this option is set to `False` then CSRF checks will be disabled regardless of the `pyramid.require_default_csrf` setting.

In addition, if this option is set to `True` or a string then CSRF origin checking will be enabled.

See *Checking CSRF Tokens Automatically* for more information.

New in version 1.7.

wrapper The *view name* of a different *view configuration* which will receive the response body of this view as the `request.wrapped_body` attribute of its own *request*, and the *response* returned by this view as the `request.wrapped_response` attribute of its own request. Using a wrapper makes it possible to “chain” views together to form a composite response. The response of the outermost wrapper view will be returned to the user. The wrapper view will be found as any view is found. See *View Configuration*. The “best” wrapper view will be found based on the lookup ordering. “Under the hood” this wrapper view is looked up via `pyramid.view.render_view_to_response(context, request, 'wrapper_viewname')`. The context and request of a wrapper view is the same context and request of the inner view.

If `wrapper` is not supplied, no wrapper view is used.

decorator A dotted Python name to a function (or the function itself) which will be used to decorate the registered *view callable*. The decorator function will be called with the view callable as a single argument. The view callable it is passed will accept `(context, request)`. The decorator must return a replacement view callable which also accepts `(context, request)`. The decorator may also be an iterable of decorators, in which case they will be applied one after the other to the view, in reverse order. For example:

```
@view_config(..., decorator=(decorator2, decorator1))
def myview(request):
    ...
```

Is similar to decorating the view callable directly:

```
@view_config(...)
@decorator2
@decorator1
def myview(request):
    ...
```

An important distinction is that each decorator will receive a response object implementing `pyramid.interfaces.IResponse` instead of the raw value returned from the view callable. All decorators in the chain must return a response object or raise an exception:

```
def log_timer(wrapped):
    def wrapper(context, request):
        start = time.time()
        response = wrapped(context, request)
        duration = time.time() - start
        response.headers['X-View-Time'] = '%.3f' % (duration,)
        log.info('view took %.3f seconds', duration)
    return response
return wrapper
```

mapper A Python object or *dotted Python name* which refers to a *view mapper*, or `None`. By default it is `None`, which indicates that the view should use the default view mapper. This plug-point is useful for Pyramid extension developers, but it's not very useful for “civilians” who are just developing stock Pyramid applications. Pay no attention to the man behind the curtain.

Predicate Arguments

These arguments modify view lookup behavior. In general the more predicate arguments that are supplied, the more specific and narrower the usage of the configured view.

name The *view name* required to match this view callable. A `name` argument is typically only used when your application uses *traversal*. Read *Traversal* to understand the concept of a view name.

If `name` is not supplied, the empty string is used (implying the default view).

context An object representing a Python class of which the *context* resource must be an instance *or* the *interface* that the *context* resource must provide in order for this view to be found and called. This predicate is true when the *context* resource is an instance of the represented class or if the *context* resource provides the represented interface; it is otherwise false.

If `context` is not supplied, the value `None`, which matches any resource, is used.

route_name If `route_name` is supplied, the view callable will be invoked only when the named route has matched.

This value must match the `name` of a *route configuration* declaration (see *URL Dispatch*) that must match before this view will be called. Note that the `route` configuration referred to by

`route_name` will usually have a `*traverse` token in the value of its `pattern`, representing a part of the path that will be used by *traversal* against the result of the route's *root factory*.

If `route_name` is not supplied, the view callable will only have a chance of being invoked if no other route was matched. This is when the request/context pair found via *resource location* does not indicate it matched any configured route.

request_type This value should be an *interface* that the *request* must provide in order for this view to be found and called.

If `request_type` is not supplied, the value `None` is used, implying any request type.

This is an advanced feature, not often used by “civilians”.

request_method This value can be either a string (such as "GET", "POST", "PUT", "DELETE", "HEAD", or "OPTIONS") representing an HTTP REQUEST_METHOD or a tuple containing one or more of these strings. A view declaration with this argument ensures that the view will only be called when the `method` attribute of the request (i.e., the REQUEST_METHOD of the WSGI environment) matches a supplied value.

Changed in version 1.4: The use of "GET" also implies that the view will respond to "HEAD".

If `request_method` is not supplied, the view will be invoked regardless of the REQUEST_METHOD of the WSGI environment.

request_param This value can be any string or a sequence of strings. A view declaration with this argument ensures that the view will only be called when the *request* has a key in the `request.params` dictionary (an HTTP GET or POST variable) that has a name which matches the supplied value.

If any value supplied has an `=` sign in it, e.g., `request_param="foo=123"`, then the key (`foo`) must both exist in the `request.params` dictionary, *and* the value must match the right hand side of the expression (`123`) for the view to “match” the current request.

If `request_param` is not supplied, the view will be invoked without consideration of keys and values in the `request.params` dictionary.

match_param This param may be either a single string of the format “key=value” or a tuple containing one or more of these strings.

This argument ensures that the view will only be called when the *request* has key/value pairs in its *matchdict* that equal those supplied in the predicate. For example, `match_param="action=edit"` would require the `action` parameter in the *matchdict* match the right hand side of the expression (`edit`) for the view to “match” the current request.

If the `match_param` is a tuple, every key/value pair must match for the predicate to pass.

If `match_param` is not supplied, the view will be invoked without consideration of the keys and values in `request.matchdict`.

New in version 1.2.

containment This value should be a reference to a Python class or *interface* that a parent object in the context resource’s *lineage* must provide in order for this view to be found and called. The resources in your resource tree must be “location-aware” to use this feature.

If `containment` is not supplied, the interfaces and classes in the lineage are not considered when deciding whether or not to invoke the view callable.

See *Location-Aware Resources* for more information about location-awareness.

xhr This value should be either `True` or `False`. If this value is specified and is `True`, the *WSGI* environment must possess an `HTTP_X_REQUESTED_WITH` header (i.e., `X-Requested-With`) that has the value `XMLHttpRequest` for the associated view callable to be found and called. This is useful for detecting AJAX requests issued from jQuery, Prototype, and other Javascript libraries.

If `xhr` is not specified, the `HTTP_X_REQUESTED_WITH` HTTP header is not taken into consideration when deciding whether or not to invoke the associated view callable.

accept The value of this argument represents a match query for one or more mimetypes in the `Accept` HTTP request header. If this value is specified, it must be in one of the following forms: a mimetype match token in the form `text/plain`, a wildcard mimetype match token in the form `text/*`, or a match-all wildcard mimetype match token in the form `*/*`. If any of the forms matches the `Accept` header of the request, this predicate will be true.

If `accept` is not specified, the `HTTP_ACCEPT` HTTP header is not taken into consideration when deciding whether or not to invoke the associated view callable.

header This value represents an HTTP header name or a header name/value pair.

If `header` is specified, it must be a header name or a `headername:headervalue` pair.

If `header` is specified without a value (a bare header name only, e.g., `If-Modified-Since`), the view will only be invoked if the HTTP header exists with any value in the request.

If `header` is specified, and possesses a name/value pair (e.g., `User-Agent:Mozilla/.*)`, the view will only be invoked if the HTTP header exists *and* the HTTP header matches the value requested. When the `headervalue` contains a `:` (colon), it will be considered a name/value pair (e.g., `User-Agent:Mozilla/.*)` or `Host:localhost`). The value portion should be a regular expression.

Whether or not the value represents a header name or a header name/value pair, the case of the header name is not significant.

If `header` is not specified, the composition, presence, or absence of HTTP headers is not taken into consideration when deciding whether or not to invoke the associated view callable.

path_info This value represents a regular expression pattern that will be tested against the `PATH_INFO` WSGI environment variable to decide whether or not to call the associated view callable. If the regex matches, this predicate will be `True`.

If `path_info` is not specified, the WSGI `PATH_INFO` is not taken into consideration when deciding whether or not to invoke the associated view callable.

check_csrf If specified, this value should be one of `None`, `True`, `False`, or a string representing the “check name”. If the value is `True` or a string, CSRF checking will be performed. If the value is `False` or `None`, CSRF checking will not be performed.

If the value provided is a string, that string will be used as the “check name”. If the value provided is `True`, `csrf_token` will be used as the check name.

If CSRF checking is performed, the checked value will be the value of `request.POST[check_name]`. This value will be compared against the value of `request.session.get_csrf_token()`, and the check will pass if these two values are the same. If the check passes, the associated view will be permitted to execute. If the check fails, the associated view will not be permitted to execute.

Note that using this feature requires a *session factory* to have been configured.

New in version 1.4a2.

physical_path If specified, this value should be a string or a tuple representing the *physical path* of the context found via traversal for this predicate to match as true. For example, `physical_path='/'`, `physical_path='/a/b/c'`, or `physical_path=(' ', 'a', 'b', 'c')`. This is not a path prefix match or a regex, but a whole-path match. It’s useful when you want to always potentially show a view when some object is traversed to, but you can’t be sure about what kind of object it will be, so you can’t use the `context` predicate. The individual path elements between slash characters or in tuple elements should be the Unicode representation of the name of the resource and should not be encoded in any way.

New in version 1.4a3.

effective_principals If specified, this value should be a *principal* identifier or a sequence of principal identifiers. If the `pyramid.request.Request.effective_principals()` method indicates that every principal named in the argument list is present in the current request, this predicate will return `True`; otherwise it will return `False`. For example: `effective_principals=pyramid.security.Authenticated` or `effective_principals=('fred', 'group:admins')`.

New in version 1.4a4.

custom_predicates If `custom_predicates` is specified, it must be a sequence of references to custom predicate callables. Use custom predicates when no set of predefined predicates do what you need. Custom predicates can be combined with predefined predicates as necessary. Each custom predicate callable should accept two arguments, `context` and `request`, and should return either `True` or `False` after doing arbitrary evaluation of the context resource and/or the request. If all callables return `True`, the associated view callable will be considered viable for a given request.

If `custom_predicates` is not specified, no custom predicates are used.

predicates Pass a key/value pair here to use a third-party predicate registered via *pyramid.config.Configurator.add_view_predicate()*. More than one key/value pair can be used at the same time. See *View and Route Predicates* for more information about third-party predicates.

New in version 1.4a1.

Inverting Predicate Values

You can invert the meaning of any predicate value by wrapping it in a call to *pyramid.config.not_*.

```
1 from pyramid.config import not_  
2  
3 config.add_view(  
4     'mypackage.views.my_view',  
5     route_name='ok',  
6     request_method=not_('POST')  
7 )
```

The above example will ensure that the view is called if the request method is *not* POST, at least if no other view is more specific.

This technique of wrapping a predicate value in `not_` can be used anywhere predicate values are accepted:

- `pyramid.config.Configurator.add_view()`
- `pyramid.view.view_config()`

New in version 1.5.

Adding View Configuration Using the `@view_config` Decorator



Using this feature tends to slow down application startup slightly, as more work is performed at application startup to scan for view configuration declarations. For maximum startup performance, use the view configuration method described in *Adding View Configuration Using `add_view()`* instead.

The `view_config` decorator can be used to associate *view configuration* information with a function, method, or class that acts as a Pyramid view callable.

Here's an example of the `view_config` decorator that lives within a Pyramid application module `views.py`:

```
1 from resources import MyResource
2 from pyramid.view import view_config
3 from pyramid.response import Response
4
5 @view_config(route_name='ok', request_method='POST', permission='read')
6 def my_view(request):
7     return Response('OK')
```

Using this decorator as above replaces the need to add this imperative configuration stanza:

```
1 config.add_view('mypackage.views.my_view', route_name='ok',
2               request_method='POST', permission='read')
```

All arguments to `view_config` may be omitted. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config()
5 def my_view(request):
6     """ My view """
7     return Response()
```

Such a registration as the one directly above implies that the view name will be `my_view`, registered with a context argument that matches any resource type, using no permission, registered against requests with any request method, request type, request param, route name, or containment.

The mere existence of a `@view_config` decorator doesn't suffice to perform view configuration. All that the decorator does is "annotate" the function with your configuration declarations, it doesn't process them. To make Pyramid process your `pyramid.view.view_config` declarations, you *must* use the `scan` method of a `pyramid.config.Configurator`:

```
1 # config is assumed to be an instance of the
2 # pyramid.config.Configurator class
3 config.scan()
```

Please see *Declarative Configuration* for detailed information about what happens when code is scanned for configuration declarations resulting from use of decorators like *view_config*.

See *pyramid.config* for additional API arguments to the *scan()* method. For example, the method allows you to supply a package argument to better control exactly *which* code will be scanned.

All arguments to the *view_config* decorator mean precisely the same thing as they would if they were passed as arguments to the *pyramid.config.Configurator.add_view()* method save for the view argument. Usage of the *view_config* decorator is a form of *declarative configuration*, while *pyramid.config.Configurator.add_view()* is a form of *imperative configuration*. However, they both do the same thing.

@view_config Placement

A *view_config* decorator can be placed in various points in your application.

If your view callable is a function, it may be used as a function decorator:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='edit')
5 def edit(request):
6     return Response('edited!')
```

If your view callable is a class, the decorator can also be used as a class decorator. All the arguments to the decorator are the same when applied against a class as when they are applied against a function. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(route_name='hello')
5 class MyView(object):
6     def __init__(self, request):
7         self.request = request
8
9     def __call__(self):
10        return Response('hello')
```

More than one `view_config` decorator can be stacked on top of any number of others. Each decorator creates a separate view registration. For example:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='edit')
5 @view_config(route_name='change')
6 def edit(request):
7     return Response('edited!')
```

This registers the same view under two different names.

The decorator can also be used against a method of a class:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 class MyView(object):
5     def __init__(self, request):
6         self.request = request
7
8     @view_config(route_name='hello')
9     def amethod(self):
10        return Response('hello')
```

When the decorator is used against a method of a class, a view is registered for the *class*, so the class constructor must accept an argument list in one of two forms: either a single argument, `request`, or two arguments, `context`, `request`.

The method which is decorated must return a *response*.

Using the decorator against a particular method of a class is equivalent to using the `attr` parameter in a decorator attached to the class itself. For example, the above registration implied by the decorator being used against the `amethod` method could be written equivalently as follows:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(attr='amethod', route_name='hello')
5 class MyView(object):
6     def __init__(self, request):
7         self.request = request
8
9     def amethod(self):
10        return Response('hello')
```

Adding View Configuration Using `add_view()`

The `pyramid.config.Configurator.add_view()` method within `pyramid.config` is used to configure a view “imperatively” (without a `view_config` decorator). The arguments to this method are very similar to the arguments that you provide to the `view_config` decorator. For example:

```
1 from pyramid.response import Response
2
3 def hello_world(request):
4     return Response('hello!')
5
6 # config is assumed to be an instance of the
7 # pyramid.config.Configurator class
8 config.add_view(hello_world, route_name='hello')
```

The first argument, a *view callable*, is the only required argument. It must either be a Python object which is the view itself or a *dotted Python name* to such an object. In the above example, the `view callable` is the `hello_world` function.

When you use only `add_view()` to add view configurations, you don’t need to issue a `scan` in order for the view configuration to take effect.

@view_defaults Class Decorator

New in version 1.3.

If you use a class as a view, you can use the `pyramid.view.view_defaults` class decorator on the class to provide defaults to the view configuration information used by every `@view_config` decorator that decorates a method of that class.

For instance, if you’ve got a class that has methods that represent “REST actions”, all of which are mapped to the same route but different request methods, instead of this:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 class RESTView(object):
5     def __init__(self, request):
6         self.request = request
7
8     @view_config(route_name='rest', request_method='GET')
```

```

9     def get(self):
10         return Response('get')
11
12     @view_config(route_name='rest', request_method='POST')
13     def post(self):
14         return Response('post')
15
16     @view_config(route_name='rest', request_method='DELETE')
17     def delete(self):
18         return Response('delete')

```

You can do this:

```

1  from pyramid.view import view_defaults
2  from pyramid.view import view_config
3  from pyramid.response import Response
4
5  @view_defaults(route_name='rest')
6  class RESTView(object):
7      def __init__(self, request):
8          self.request = request
9
10     @view_config(request_method='GET')
11     def get(self):
12         return Response('get')
13
14     @view_config(request_method='POST')
15     def post(self):
16         return Response('post')
17
18     @view_config(request_method='DELETE')
19     def delete(self):
20         return Response('delete')

```

In the above example, we were able to take the `route_name='rest'` argument out of the call to each individual `@view_config` statement because we used a `@view_defaults` class decorator to provide the argument as a default to each view method it possessed.

Arguments passed to `@view_config` will override any default passed to `@view_defaults`.

The `view_defaults` class decorator can also provide defaults to the `pyramid.config.Configurator.add_view()` directive when a decorated class is passed to that directive as its `view` argument. For example, instead of this:

```
1 from pyramid.response import Response
2 from pyramid.config import Configurator
3
4 class RESTView(object):
5     def __init__(self, request):
6         self.request = request
7
8     def get(self):
9         return Response('get')
10
11    def post(self):
12        return Response('post')
13
14    def delete(self):
15        return Response('delete')
16
17 def main(global_config, **settings):
18     config = Configurator()
19     config.add_route('rest', '/rest')
20     config.add_view(
21         RESTView, route_name='rest', attr='get', request_method='GET')
22     config.add_view(
23         RESTView, route_name='rest', attr='post', request_method='POST')
24     config.add_view(
25         RESTView, route_name='rest', attr='delete', request_method='DELETE
26         ↪ ')
27     return config.make_wsgi_app()
```

To reduce the amount of repetition in the `config.add_view` statements, we can move the `route_name='rest'` argument to a `@view_defaults` class decorator on the `RESTView` class:

```
1 from pyramid.view import view_defaults
2 from pyramid.response import Response
3 from pyramid.config import Configurator
4
5 @view_defaults(route_name='rest')
6 class RESTView(object):
7     def __init__(self, request):
8         self.request = request
9
10    def get(self):
11        return Response('get')
12
13    def post(self):
14        return Response('post')
```

```

15
16     def delete(self):
17         return Response('delete')
18
19 def main(global_config, **settings):
20     config = Configurator()
21     config.add_route('rest', '/rest')
22     config.add_view(RESTView, attr='get', request_method='GET')
23     config.add_view(RESTView, attr='post', request_method='POST')
24     config.add_view(RESTView, attr='delete', request_method='DELETE')
25     return config.make_wsgi_app()

```

`pyramid.view.view_defaults` accepts the same set of arguments that `pyramid.view.view_config` does, and they have the same meaning. Each argument passed to `view_defaults` provides a default for the view configurations of methods of the class it's decorating.

Normal Python inheritance rules apply to defaults added via `view_defaults`. For example:

```

1 @view_defaults(route_name='rest')
2 class Foo(object):
3     pass
4
5 class Bar(Foo):
6     pass

```

The `Bar` class above will inherit its view defaults from the arguments passed to the `view_defaults` decorator of the `Foo` class. To prevent this from happening, use a `view_defaults` decorator without any arguments on the subclass:

```

1 @view_defaults(route_name='rest')
2 class Foo(object):
3     pass
4
5 @view_defaults()
6 class Bar(Foo):
7     pass

```

The `view_defaults` decorator only works as a class decorator; using it against a function or a method will produce nonsensical results.

Configuring View Security

If an *authorization policy* is active, any *permission* attached to a *view configuration* found during view lookup will be verified. This will ensure that the currently authenticated user possesses that permission against the *context* resource before the view function is actually called. Here's an example of specifying a permission in a view configuration using `add_view()`:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_route('add', '/add.html', factory='mypackage.Blog')
4 config.add_view('myproject.views.add_entry', route_name='add',
5               permission='add')
```

When an *authorization policy* is enabled, this view will be protected with the `add` permission. The view will *not be called* if the user does not possess the `add` permission relative to the current *context*. Instead the *forbidden view* result will be returned to the client as per *Protecting Views with Permissions*.

NotFound Errors

It's useful to be able to debug *NotFound* error responses when they occur unexpectedly due to an application registry misconfiguration. To debug these errors, use the `PYRAMID_DEBUG_NOTFOUND` environment variable or the `pyramid.debug_notfound` configuration file setting. Details of why a view was not found will be printed to `stderr`, and the browser representation of the error will include the same information. See *Environment Variables and .ini File Settings* for more information about how, and where to set these values.

Influencing HTTP Caching

New in version 1.1.

When a non-None `http_cache` argument is passed to a view configuration, Pyramid will set `Expires` and `Cache-Control` response headers in the resulting response, causing browsers to cache the response data for some time. See `http_cache` in *Non-Predicate Arguments* for the allowable values and what they mean.

Sometimes it's undesirable to have these headers set as the result of returning a response from a view, even though you'd like to decorate the view with a view configuration decorator that has `http_cache`. Perhaps there's an alternative branch in your view code that returns a response that should never be cacheable, while the "normal" branch returns something that should always be cacheable. If this is the

case, set the `prevent_auto` attribute of the `response.cache_control` object to a non-False value. For example, the below view callable is configured with a `@view_config` decorator that indicates any response from the view should be cached for 3600 seconds. However, the view itself prevents caching from taking place unless there's a `should_cache` GET or POST variable:

```
from pyramid.view import view_config

@view_config(http_cache=3600)
def view(request):
    response = Response()
    if 'should_cache' not in request.params:
        response.cache_control.prevent_auto = True
    return response
```

Note that the `http_cache` machinery will overwrite or add to caching headers you set within the view itself, unless you use `prevent_auto`.

You can also turn off the effect of `http_cache` entirely for the duration of a Pyramid application lifetime. To do so, set the `PYRAMID_PREVENT_HTTP_CACHE` environment variable or the `pyramid.prevent_http_cache` configuration value setting to a true value. For more information, see *Preventing HTTP Caching*.

Note that setting `pyramid.prevent_http_cache` will have no effect on caching headers that your application code itself sets. It will only prevent caching headers that would have been set by the Pyramid HTTP caching machinery invoked as the result of the `http_cache` argument to view configuration.

Debugging View Configuration

See *Displaying Matching Views for a Given URL* for information about how to display each of the view callables that might match for a given URL. This can be an effective way to figure out why a particular view callable is being called instead of the one you'd like to be called.

Static Assets

An *asset* is any file contained within a Python *package* which is *not* a Python source code file. For example, each of the following is an asset:

- a GIF image file contained within a Python package or contained within any subdirectory of a Python package.

- a CSS file contained within a Python package or contained within any subdirectory of a Python package.
- a JavaScript source file contained within a Python package or contained within any subdirectory of a Python package.
- A directory within a package that does not have an `__init__.py` in it (if it possessed an `__init__.py` it would *be* a package).
- a *Chameleon* or *Mako* template file contained within a Python package.

The use of assets is quite common in most web development projects. For example, when you create a Pyramid application using one of the available scaffolds, as described in *Creating the Project*, the directory representing the application contains a Python *package*. Within that Python package, there are directories full of files which are static assets. For example, there's a `static` directory which contains `.css`, `.js`, and `.gif` files. These asset files are delivered when a user visits an application URL.

Understanding Asset Specifications

Let's imagine you've created a Pyramid application that uses a *Chameleon* ZPT template via the `pyramid.renderers.render_to_response()` API. For example, the application might address the asset using the *asset specification* `myapp:templates/some_template.pt` using that API within a `views.py` file inside a `myapp` package:

```
1 from pyramid.renderers import render_to_response
2 render_to_response('myapp:templates/some_template.pt', {}, request)
```

“Under the hood”, when this API is called, Pyramid attempts to make sense out of the string `myapp:templates/some_template.pt` provided by the developer. This string is an *asset specification*. It is composed of two parts:

- The *package name* (`myapp`)
- The *asset name* (`templates/some_template.pt`), relative to the package directory.

The two parts are separated by a colon `:` character.

Pyramid uses the Python *pkg_resources* API to resolve the package name and asset name to an absolute (operating system-specific) file name. It eventually passes this resolved absolute filesystem path to the Chameleon templating engine, which then uses it to load, parse, and execute the template file.

There is a second form of asset specification: a *relative* asset specification. Instead of using an “absolute” asset specification which includes the package name, in certain circumstances you can omit the package name from the specification. For example, you might be able to use `templates/mytemplate.pt` instead of `myapp:templates/some_template.pt`. Such asset specifications are usually relative to a “current package”. The “current package” is usually the package which contains the code that *uses* the asset specification. Pyramid APIs which accept relative asset specifications typically describe to what the asset is relative in their individual documentation.

Serving Static Assets

Pyramid makes it possible to serve up static asset files from a directory on a filesystem to an application user's browser. Use the `pyramid.config.Configurator.add_static_view()` to instruct Pyramid to serve static assets, such as JavaScript and CSS files. This mechanism makes a directory of static files available at a name relative to the application root URL, e.g., `/static`, or as an external URL.

i `add_static_view()` cannot serve a single file, nor can it serve a directory of static files directly relative to the root URL of a Pyramid application. For these features, see *Advanced: Serving Static Assets Using a View Callable*.

Here's an example of a use of `add_static_view()` that will serve files up from the `/var/www/static` directory of the computer which runs the Pyramid application as URLs beneath the `/static` URL prefix.

```
1 # config is an instance of pyramid.config.Configurator
2 config.add_static_view(name='static', path='/var/www/static')
```

The name represents a URL *prefix*. In order for files that live in the `path` directory to be served, a URL that requests one of them must begin with that prefix. In the example above, `name` is `static` and `path` is `/var/www/static`. In English this means that you wish to serve the files that live in `/var/www/static` as sub-URLs of the `/static` URL prefix. Therefore, the file `/var/www/static/foo.css` will be returned when the user visits your application's URL `/static/foo.css`.

A static directory named at `path` may contain subdirectories recursively, and any subdirectories may hold files; these will be resolved by the static view as you would expect. The `Content-Type` header returned by the static view for each particular type of file is dependent upon its file extension.

By default, all files made available via `add_static_view()` are accessible by completely anonymous users. Simple authorization can be required, however. To protect a set of static files using a permission, in addition to passing the required `name` and `path` arguments, also pass the `permission` keyword argument to `add_static_view()`. The value of the `permission` argument represents the *permission* that the user must have relative to the current *context* when the static view is invoked. A user will be required to possess this permission to view any of the files represented by `path` of the static view. If your static assets must be protected by a more complex authorization scheme, see *Advanced: Serving Static Assets Using a View Callable*.

Here's another example that uses an *asset specification* instead of an absolute path as the `path` argument. To convince `add_static_view()` to serve files up under the `/static` URL from the `a/b/c/static` directory of the Python package named `some_package`, we can use a fully qualified *asset specification* as the `path`:

```
1 # config is an instance of pyramid.config.Configurator
2 config.add_static_view(name='static', path='some_package:a/b/c/static')
```

The path provided to `add_static_view()` may be a fully qualified *asset specification* or an *absolute path*.

Instead of representing a URL prefix, the name argument of a call to `add_static_view()` can alternately be a *URL*. Each of the examples we've seen so far have shown usage of the name argument as a URL prefix. However, when name is a *URL*, static assets can be served from an external webserver. In this mode, the name is used as the URL prefix when generating a URL using `pyramid.request.Request.static_url()`.

For example, `add_static_view()` may be fed a name argument which is `http://example.com/images`:

```
1 # config is an instance of pyramid.config.Configurator
2 config.add_static_view(name='http://example.com/images',
3                        path='mypackage:images')
```

Because `add_static_view()` is provided with a name argument that is the URL `http://example.com/images`, subsequent calls to `static_url()` with paths that start with the path argument passed to `add_static_view()` will generate a URL something like `http://example.com/images/logo.png`. The external webserver listening on `example.com` must be itself configured to respond properly to such a request. The `static_url()` API is discussed in more detail later in this chapter.

Generating Static Asset URLs

When an `add_static_view()` method is used to register a static asset directory, a special helper API named `pyramid.request.Request.static_url()` can be used to generate the appropriate URL for an asset that lives in one of the directories named by the static registration path attribute.

For example, let's assume you create a set of static declarations like so:

```
1 config.add_static_view(name='static1', path='mypackage:assets/1')
2 config.add_static_view(name='static2', path='mypackage:assets/2')
```

These declarations create URL-accessible directories which have URLs that begin with `/static1` and `/static2`, respectively. The assets in the `assets/1` directory of the `mypackage` package are consulted when a user visits a URL which begins with `/static1`, and the assets in the `assets/2` directory of the `mypackage` package are consulted when a user visits a URL which begins with `/static2`.

You needn't generate the URLs to static assets “by hand” in such a configuration. Instead, use the `static_url()` API to generate them for you. For example:

```
1 from pyramid.renderers import render_to_response
2
3 def my_view(request):
4     css_url = request.static_url('mypackage:assets/1/foo.css')
5     js_url = request.static_url('mypackage:assets/2/foo.js')
6     return render_to_response('templates/my_template.pt',
7                               dict(css_url=css_url, js_url=js_url),
8                               request=request)
```

If the request “application URL” of the running system is `http://example.com`, the `css_url` generated above would be: `http://example.com/static1/foo.css`. The `js_url` generated above would be `http://example.com/static2/foo.js`.

One benefit of using the `static_url()` function rather than constructing static URLs “by hand” is that if you need to change the name of a static URL declaration, the generated URLs will continue to resolve properly after the rename.

URLs may also be generated by `static_url()` to static assets that live *outside* the Pyramid application. This will happen when the `add_static_view()` API associated with the path fed to `static_url()` is a *URL* instead of a view name. For example, the name argument may be `http://example.com` while the path given may be `mypackage:images`:

```
1 config.add_static_view(name='http://example.com/images',
2                        path='mypackage:images')
```

Under such a configuration, the URL generated by `static_url` for assets which begin with `mypackage:images` will be prefixed with `http://example.com/images`:

```
1 request.static_url('mypackage:images/logo.png')
2 # -> http://example.com/images/logo.png
```

Using `static_url()` in conjunction with a `add_static_view()` makes it possible to put static media on a separate webserver during production (if the name argument to `add_static_view()` is

a URL), while keeping static media package-internal and served by the development webserver during development (if the `name` argument to `add_static_view()` is a URL prefix).

For example, we may define a *custom setting* named `media_location` which we can set to an external URL in production when our assets are hosted on a CDN.

```
1 media_location = settings.get('media_location', 'static')
2
3 config = Configurator(settings=settings)
4 config.add_static_view(path='myapp:static', name=media_location)
```

Now we can optionally define the setting in our ini file:

```
1 # production.ini
2 [app:main]
3 use = egg:myapp#main
4
5 media_location = http://static.example.com/
```

It is also possible to serve assets that live outside of the source by referring to an absolute path on the filesystem. There are two ways to accomplish this.

First, `add_static_view()` supports taking an absolute path directly instead of an asset spec. This works as expected, looking in the file or folder of files and serving them up at some URL within your application or externally. Unfortunately, this technique has a drawback in that it is not possible to use the `static_url()` method to generate URLs, since it works based on an asset specification.

New in version 1.6.

The second approach, available in Pyramid 1.6+, uses the asset overriding APIs described in the *Overriding Assets* section. It is then possible to configure a “dummy” package which then serves its file or folder from an absolute path.

```
config.add_static_view(path='myapp:static_images', name='static')
config.override_asset(to_override='myapp:static_images/',
                     override_with='/abs/path/to/images/')
```

From this configuration it is now possible to use `static_url()` to generate URLs to the data in the folder by doing something like `request.static_url('myapp:static_images/foo.png')`. While it is not necessary that the `static_images` file or folder actually exist in the `myapp` package, it is important that the `myapp` portion points to a valid package. If the folder does exist, then the overridden folder is given priority, if the file’s name exists in both locations.

Cache Busting

New in version 1.6.

In order to maximize performance of a web application, you generally want to limit the number of times a particular client requests the same static asset. Ideally a client would cache a particular static asset “forever”, requiring it to be sent to the client a single time. The HTTP protocol allows you to send headers with an HTTP response that can instruct a client to cache a particular asset for an amount of time. As long as the client has a copy of the asset in its cache and that cache hasn’t expired, the client will use the cached copy rather than request a new copy from the server. The drawback to sending cache headers to the client for a static asset is that at some point the static asset may change, and then you’ll want the client to load a new copy of the asset. Under normal circumstances you’d just need to wait for the client’s cached copy to expire before they get the new version of the static resource.

A commonly used workaround to this problem is a technique known as *cache busting*. Cache busting schemes generally involve generating a URL for a static asset that changes when the static asset changes. This way headers can be sent along with the static asset instructing the client to cache the asset for a very long time. When a static asset is changed, the URL used to refer to it in a web page also changes, so the client sees it as a new resource and requests the asset, regardless of any caching policy set for the resource’s old URL.


Pyramid can be configured to produce cache busting URLs for static assets using `add_cache_buster()`:

```
1 import time
2 from pyramid.static import QueryStringConstantCacheBuster
3
4 # config is an instance of pyramid.config.Configurator
5 config.add_static_view(name='static', path='mypackage:folder/static/')
6 config.add_cache_buster(
7     'mypackage:folder/static/',
8     QueryStringConstantCacheBuster(str(int(time.time()))))
```

Adding the cachebuster instructs Pyramid to add the current time for a static asset to the query string in the asset’s URL:

```
1 js_url = request.static_url('mypackage:folder/static/js/myapp.js')
2 # Returns: 'http://www.example.com/static/js/myapp.js?x=1445318121'
```

When the web server restarts, the time constant will change and therefore so will its URL.

 Cache busting is an inherently complex topic as it integrates the asset pipeline and the web application. It is expected and desired that application authors will write their own cache buster implementations conforming to the properties of their own asset pipelines. See *Customizing the Cache Buster* for information on writing your own.

Disabling the Cache Buster

It can be useful in some situations (e.g., development) to globally disable all configured cache busters without changing calls to `add_cache_buster()`. To do this set the `PYRAMID_PREVENT_CACHEBUST` environment variable or the `pyramid.prevent_cachebust` configuration value to a true value.

Customizing the Cache Buster

Calls to `add_cache_buster()` may use any object that implements the interface *ICacheBuster*.

Pyramid ships with a very simplistic *QueryStringConstantCacheBuster*, which adds an arbitrary token you provide to the query string of the asset's URL. This is almost never what you want in production as it does not allow fine-grained busting of individual assets.

In order to implement your own cache buster, you can write your own class from scratch which implements the *ICacheBuster* interface. Alternatively you may choose to subclass one of the existing implementations. One of the most likely scenarios is you'd want to change the way the asset token is generated. To do this just subclass *QueryStringCacheBuster* and define a `tokenize(pathspec)` method. Here is an example which uses Git to get the hash of the current commit:

```
1 import os
2 import subprocess
3 from pyramid.static import QueryStringCacheBuster
4
5 class GitCacheBuster(QueryStringCacheBuster):
6     """
7     Assuming your code is installed as a Git checkout, as opposed to an egg
8     from an egg repository like PYPI, you can use this cachebuster to get
9     the current commit's SHA1 to use as the cache bust token.
10    """
11    def __init__(self, param='x', repo_path=None):
12        super(GitCacheBuster, self).__init__(param=param)
```

```

13         if repo_path is None:
14             repo_path = os.path.dirname(os.path.abspath(__file__))
15         self.shal = subprocess.check_output(
16             ['git', 'rev-parse', 'HEAD'],
17             cwd=repo_path).strip()
18
19     def tokenize(self, pathspec):
20         return self.shal

```

A simple cache buster that modifies the path segment can be constructed as well:

```

1 import posixpath
2
3 class PathConstantCacheBuster(object):
4     def __init__(self, token):
5         self.token = token
6
7     def __call__(self, request, subpath, kw):
8         base_subpath, ext = posixpath.splitext(subpath)
9         new_subpath = base_subpath + self.token + ext
10        return new_subpath, kw

```

The caveat with this approach is that modifying the path segment changes the file name, and thus must match what is actually on the filesystem in order for `add_static_view()` to find the file. It's better to use the *ManifestCacheBuster* for these situations, as described in the next section.

Path Segments and Choosing a Cache Buster

Many caching HTTP proxies will fail to cache a resource if the URL contains a query string. Therefore, in general, you should prefer a cache busting strategy which modifies the path segment rather than methods which add a token to the query string.

You will need to consider whether the Pyramid application will be serving your static assets, whether you are using an external asset pipeline to handle rewriting urls internal to the css/javascript, and how fine-grained do you want the cache busting tokens to be.

In many cases you will want to host the static assets on another web server or externally on a CDN. In these cases your Pyramid application may not even have access to a copy of the static assets. In order to cache bust these assets you will need some information about them.

If you are using an external asset pipeline to generate your static files you should consider using the *ManifestCacheBuster*. This cache buster can load a standard JSON formatted file generated by

your pipeline and use it to cache bust the assets. This has many performance advantages as Pyramid does not need to look at the files to generate any cache busting tokens, but still supports fine-grained per-file tokens.

Assuming an example `manifest.json` like:

```
{
  "css/main.css": "css/main-678b7c80.css",
  "images/background.png": "images/background-a8169106.png"
}
```

The following code would set up a cachebuster:

```
1 from pyramid.static import ManifestCacheBuster
2
3 config.add_static_view(
4     name='http://mycdn.example.com/',
5     path='mypackage:static')
6
7 config.add_cache_buster(
8     'mypackage:static/',
9     ManifestCacheBuster('myapp:static/manifest.json'))
```

It's important to note that the cache buster only handles generating cache-busted URLs for static assets. It does **NOT** provide any solutions for serving those assets. For example, if you generated a URL for `css/main-678b7c80.css` then that URL needs to be valid either by configuring `add_static_view` properly to point to the location of the files or some other mechanism such as the files existing on your CDN or rewriting the incoming URL to remove the cache bust tokens.

CSS and JavaScript source and cache busting

Often one needs to refer to images and other static assets inside CSS and JavaScript files. If cache busting is active, the final static asset URL is not available until the static assets have been assembled. These URLs cannot be handwritten. Below is an example of how to integrate the cache buster into the entire stack. Remember, it is just an example and should be modified to fit your specific tools.

- First, process the files by using a precompiler which rewrites URLs to their final cache-busted form. Then, you can use the *ManifestCacheBuster* to synchronize your asset pipeline with Pyramid, allowing the pipeline to have full control over the final URLs of your assets.

Now that you are able to generate static URLs within Pyramid, you'll need to handle URLs that are out of our control. To do this you may use some of the following options to get started:

- Configure your asset pipeline to rewrite URL references inline in CSS and JavaScript. This is the best approach because then the files may be hosted by Pyramid or an external CDN without having to change anything. They really are static.
- Templatize JS and CSS, and call `request.static_url()` inside their template code. While this approach may work in certain scenarios, it is not recommended because your static assets will not really be static and are now dependent on Pyramid to be served correctly. See *Advanced: Serving Static Assets Using a View Callable* for more information on this approach.

If your CSS and JavaScript assets use URLs to reference other assets it is recommended that you implement an external asset pipeline that can rewrite the generated static files with new URLs containing cache busting tokens. The machinery inside Pyramid will not help with this step as it has very little knowledge of the asset types your application may use. The integration into Pyramid is simply for linking those assets into your HTML and other dynamic content.

Advanced: Serving Static Assets Using a View Callable

For more flexibility, static assets can be served by a *view callable* which you register manually. For example, if you're using *URL dispatch*, you may want static assets to only be available as a fallback if no previous route matches. Alternatively, you might like to serve a particular static asset manually, because its download requires authentication.

Note that you cannot use the `static_url()` API to generate URLs against assets made accessible by registering a custom static view.

Root-Relative Custom Static View (URL Dispatch Only)


The `pyramid.static.static_view` helper class generates a Pyramid view callable. This view callable can serve static assets from a directory. An instance of this class is actually used by the `add_static_view()` configuration method, so its behavior is almost exactly the same once it's configured.



The following example *will not work* for applications that use *traversal*; it will only work if you use *URL dispatch* exclusively. The root-relative route we'll be registering will always be matched before traversal takes place, subverting any views registered via `add_view` (at least those without a `route_name`). A `static_view` static view cannot be made root-relative when you use traversal unless it's registered as a *Not Found View*.

To serve files within a directory located on your filesystem at `/path/to/static/dir` as the result of a “catchall” route hanging from the root that exists at the end of your routing table, create an instance of the `static_view` class inside a `static.py` file in your application root as below.

```
1 from pyramid.static import static_view
2 static_view = static_view('/path/to/static/dir', use_subpath=True)
```

 For better cross-system flexibility, use an *asset specification* as the argument to `static_view` instead of a physical absolute filesystem path, e.g., `mypackage:static`, instead of `/path/to/mypackage/static`.

Subsequently, you may wire the files that are served by this view up to be accessible as `/<filename>` using a configuration method in your application’s startup code.

```
1 # .. every other add_route declaration should come
2 # before this one, as it will, by default, catch all requests
3
4 config.add_route('catchall_static', '/*subpath')
5 config.add_view('myapp.static.static_view', route_name='catchall_static')
```

The special name `*subpath` above is used by the `static_view` view callable to signify the path of the file relative to the directory you’re serving.

Registering a View Callable to Serve a “Static” Asset

You can register a simple view callable to serve a single static asset. To do so, do things “by hand”. First define the view callable.

```
1 import os
2 from pyramid.response import FileResponse
3
4 def favicon_view(request):
5     here = os.path.dirname(__file__)
6     icon = os.path.join(here, 'static', 'favicon.ico')
7     return FileResponse(icon, request=request)
```

The above bit of code within `favicon_view` computes “here”, which is a path relative to the Python file in which the function is defined. It then creates a `pyramid.response.FileResponse` using the file path as the response’s `path` argument and the request as the response’s `request` argument. `pyramid.response.FileResponse` will serve the file as quickly as possible when it’s used this way. It makes sure to set the right content length and content_type, too, based on the file extension of the file you pass.

You might register such a view via configuration as a view callable that should be called as the result of a traversal:

```
1 config.add_view('myapp.views.favicon_view', name='favicon.ico')
```

Or you might register it to be the view callable for a particular route:

```
1 config.add_route('favicon', '/favicon.ico')
2 config.add_view('myapp.views.favicon_view', route_name='favicon')
```

Because this is a simple view callable, it can be protected with a *permission* or can be configured to respond under different circumstances using *view predicate* arguments.

Overriding Assets

It can often be useful to override specific assets from “outside” a given Pyramid application. For example, you may wish to reuse an existing Pyramid application more or less unchanged. However, some specific template file owned by the application might have inappropriate HTML, or some static asset (such as a logo file or some CSS file) might not be appropriate. You *could* just fork the application entirely, but it’s often more convenient to just override the assets that are inappropriate and reuse the application “as is”. This is particularly true when you reuse some “core” application over and over again for some set of customers (such as a CMS application, or some bug tracking application), and you want to make arbitrary visual modifications to a particular application deployment without forking the underlying code.

To this end, Pyramid contains a feature that makes it possible to “override” one asset with one or more other assets. In support of this feature, a *Configurator* API exists named `pyramid.config.Configurator.override_asset()`. This API allows you to *override* the following kinds of assets defined in any Python package:

- Individual template files.
- A directory containing multiple template files.
- Individual static files served up by an instance of the `pyramid.static.static_view` helper class.
- A directory of static files served up by an instance of the `pyramid.static.static_view` helper class.
- Any other asset (or set of assets) addressed by code that uses the `setuptools.pkg_resources` API.

The `override_asset` API

An individual call to `override_asset()` can override a single asset. For example:

```
1 config.override_asset(  
2     to_override='some.package:templates/mytemplate.pt',  
3     override_with='another.package:othertemplates/anothertemplate.pt')
```

The string value passed to both `to_override` and `override_with` sent to the `override_asset` API is called an *asset specification*. The colon separator in a specification separates the *package name* from the *asset name*. The colon and the following asset name are optional. If they are not specified, the override attempts to resolve every lookup into a package from the directory of another package. For example:

```
1 config.override_asset(to_override='some.package',  
2                       override_with='another.package')
```

Individual subdirectories within a package can also be overridden:

```
1 config.override_asset(to_override='some.package:templates/',  
2                       override_with='another.package:othertemplates/')
```

If you wish to override a directory with another directory, you *must* make sure to attach the slash to the end of both the `to_override` specification and the `override_with` specification. If you fail to attach a slash to the end of a specification that points to a directory, you will get unexpected results.

You cannot override a directory specification with a file specification, and vice versa; a startup error will occur if you try. You cannot override an asset with itself; a startup error will occur if you try.

Only individual *package* assets may be overridden. Overrides will not traverse through subpackages within an overridden package. This means that if you want to override assets for both `some.package:templates`, and `some.package.views:templates`, you will need to register two overrides.

The package name in a specification may start with a dot, meaning that the package is relative to the package in which the configuration construction file resides (or the package argument to the *Configurator* class construction). For example:

```
1 config.override_asset(to_override='.subpackage:templates/',  
2                       override_with='another.package:templates/')
```

Multiple calls to `override_asset` which name a shared `to_override` but a different `override_with` specification can be “stacked” to form a search path. The first asset that exists in the search path will be used; if no asset exists in the override path, the original asset is used.

Asset overrides can actually override assets other than templates and static files. Any software which uses the `pkg_resources.get_resource_filename()`, `pkg_resources.get_resource_stream()`, or `pkg_resources.get_resource_string()` APIs will obtain an overridden file when an override is used.

New in version 1.6: As of Pyramid 1.6, it is also possible to override an asset by supplying an absolute path to a file or directory. This may be useful if the assets are not distributed as part of a Python package.

Cache Busting and Asset Overrides

Overriding static assets that are being hosted using `pyramid.config.Configurator.add_static_view()` can affect your cache busting strategy when using any cache busters that are asset-aware such as `pyramid.static.ManifestCacheBuster`. What sets asset-aware cache busters apart is that they have logic tied to specific assets. For example, a manifest is only generated for a specific set of pre-defined assets. Now, imagine you have overridden an asset defined in this manifest with a new, unknown version. By default, the cache buster will be invoked for an asset it has never seen before and will likely end up returning a cache busting token for the original asset rather than the asset that will actually end up being served! In order to get around this issue, it’s possible to attach a different `pyramid.interfaces.ICacheBuster` implementation to the new assets. This would cause the original assets to be served by their manifest, and the new assets served by their own cache buster. To do this, `pyramid.config.Configurator.add_cache_buster()` supports an explicit option. For example:

```


1  from pyramid.static import ManifestCacheBuster
2
3  # define a static view for myapp:static assets
4  config.add_static_view('static', 'myapp:static')
5
6  # setup a cache buster for your app based on the myapp:static assets
7  my_cb = ManifestCacheBuster('myapp:static/manifest.json')
8  config.add_cache_buster('myapp:static', my_cb)
9
10 # override an asset
11 config.override_asset(
12     to_override='myapp:static/background.png',
13     override_with='theme:static/background.png')
14
15 # override the cache buster for theme:static assets
16 theme_cb = ManifestCacheBuster('theme:static/manifest.json')
17 config.add_cache_buster('theme:static', theme_cb, explicit=True)

```


In the above example there is a default cache buster, `my_cb`, for all assets served from the `myapp:static` folder. This would also affect `theme:static/background.png` when generating URLs via `request.static_url('myapp:static/background.png')`.

The `theme_cb` is defined explicitly for any assets loaded from the `theme:static` folder. Explicit cache busters have priority and thus `theme_cb` would be invoked for `request.static_url('myapp:static/background.png')`, but `my_cb` would be used for any other assets like `request.static_url('myapp:static/favicon.ico')`.

Request and Response Objects

 This chapter is adapted from a portion of the *WebOb* documentation, originally written by Ian Bicking.

Pyramid uses the *WebOb* package as a basis for its *request* and *response* object implementations. The *request* object that is passed to a Pyramid *view* is an instance of the `pyramid.request.Request` class, which is a subclass of `webob.Request`. The *response* returned from a Pyramid *view renderer* is an instance of the `pyramid.response.Response` class, which is a subclass of the `webob.Response` class. Users can also return an instance of `pyramid.response.Response` directly from a view as necessary.

WebOb is a project separate from Pyramid with a separate set of authors and a fully separate set of documentation. Pyramid adds some functionality to the standard WebOb request, which is documented in the *pyramid.request* API documentation.

WebOb provides objects for HTTP requests and responses. Specifically it does this by wrapping the WSGI request environment and response status, header list, and `app_iter` (body) values.

WebOb request and response objects provide many conveniences for parsing WSGI requests and forming WSGI responses. WebOb is a nice way to represent “raw” WSGI requests and responses. However, we won’t cover that use case in this document, as users of Pyramid don’t typically need to use the WSGI-related features of WebOb directly. The reference documentation shows many examples of creating requests and using response objects in this manner, however.

Request

The request object is a wrapper around the WSGI environ dictionary. This dictionary contains keys for each header, keys that describe the request (including the path and query string), a file-like object for the request body, and a variety of custom keys. You can always access the environ with `req.environ`.

Some of the most important and interesting attributes of a request object are below.

`req.method` The request method, e.g., GET, POST

`req.GET` A *multidict* with all the variables in the query string.

`req.POST` A *multidict* with all the variables in the request body. This only has variables if the request was a POST and it is a form submission.

`req.params` A *multidict* with a combination of everything in `req.GET` and `req.POST`.

`req.body` The contents of the body of the request. This contains the entire request body as a string. This is useful when the request is a POST that is *not* a form submission, or a request like a PUT. You can also get `req.body_file` for a file-like object.

`req.json_body` The JSON-decoded contents of the body of the request. See *Dealing with a JSON-Encoded Request Body*.

`req.cookies` A simple dictionary of all the cookies.

`req.headers` A dictionary of all the headers. This dictionary is case-insensitive.

`req.urlvars` and `req.urlargs` `req.urlvars` are the keyword parameters associated with the request URL. `req.urlargs` are the positional parameters. These are set by products like Routes and Selector.

Also for standard HTTP request headers, there are usually attributes such as `req.accept_language`, `req.content_length`, and `req.user_agent`. These properties expose the *parsed* form of each header, for whatever parsing makes sense. For instance, `req.if_modified_since` returns a datetime object (or None if the header is was not provided).



Full API documentation for the Pyramid request object is available in *pyramid.request*.

Special Attributes Added to the Request by Pyramid

In addition to the standard *WebOb* attributes, Pyramid adds special attributes to every request: `context`, `registry`, `root`, `subpath`, `traversed`, `view_name`, `virtual_root`, `virtual_root_path`, `session`, `matchdict`, and `matched_route`. These attributes are documented further within the *pyramid.request.Request* API documentation.

URLs

In addition to these attributes, there are several ways to get the URL of the request and its parts. We'll show various values for an example URL `http://localhost/app/blog?id=10`, where the application is mounted at `http://localhost/app`.

req.url The full request URL with query string, e.g., `http://localhost/app/blog?id=10`

req.host The host information in the URL, e.g., `localhost`

req.host_url The URL with the host, e.g., `http://localhost`

req.application_url The URL of the application (just the `SCRIPT_NAME` portion of the path, not `PATH_INFO`), e.g., `http://localhost/app`

req.path_url The URL of the application including the `PATH_INFO`, e.g., `http://localhost/app/blog`

req.path The URL including `PATH_INFO` without the host or scheme, e.g., `/app/blog`

req.path_qs The URL including `PATH_INFO` and the query string, e.g., `/app/blog?id=10`

req.query_string The query string in the URL, e.g., `id=10`

req.relative_url(url, to_application=False) Gives a URL relative to the current URL. If `to_application` is `True`, then resolves it relative to `req.application_url`.

Methods

There are methods of request objects documented in `pyramid.request.Request` but you'll find that you won't use very many of them. Here are a couple that might be useful:

`Request.blank(base_url)` Creates a new request with blank information, based at the given URL. This can be useful for subrequests and artificial requests. You can also use `req.copy()` to copy an existing request, or for subrequests `req.copy_get()` which copies the request but always turns it into a GET (which is safer to share for subrequests).

`req.get_response(wsgi_application)` This method calls the given WSGI application with this request, and returns a `pyramid.response.Response` object. You can also use this for subrequests or testing.

Text (Unicode)

Many of the properties of the request object will be text values (unicode under Python 2 or `str` under Python 3) if the request encoding/charset is provided. If it is provided, the values in `req.POST`, `req.GET`, `req.params`, and `req.cookies` will contain text. The client *can* indicate the charset with something like `Content-Type: application/x-www-form-urlencoded; charset=utf8`, but browsers seldom set this. You can reset the charset of an existing request with `newreq = req.decode('utf-8')`, or during instantiation with `Request(envIRON, charset='utf8')`.

Multidict

Several attributes of a WebOb request are multidict structures (such as `request.GET`, `request.POST`, and `request.params`). A multidict is a dictionary where a key can have multiple values. The quintessential example is a query string like `?pref=red&pref=blue`; the `pref` variable has two values: `red` and `blue`.

In a multidict, when you do `request.GET['pref']`, you'll get back only `"blue"` (the last value of `pref`). This returned result might not be expected—sometimes returning a string, and sometimes returning a list—and may be cause of frequent exceptions. If you want *all* the values back, use `request.GET.getall('pref')`. If you want to be sure there is *one and only one* value, use `request.GET.getone('pref')`, which will raise an exception if there is zero or more than one value for `pref`.

When you use operations like `request.GET.items()`, you'll get back something like `[('pref', 'red'), ('pref', 'blue')]`. All the key/value pairs will show up. Similarly `request.GET.keys()` returns `['pref', 'pref']`. Multidict is a view on a list of tuples; all the keys are ordered, and all the values are ordered.

API documentation for a multidict exists as `pyramid.interfaces.IMultiDict`.

Dealing with a JSON-Encoded Request Body

New in version 1.1.

`pyramid.request.Request.json_body` is a property that returns a *JSON*-decoded representation of the request body. If the request does not have a body, or the body is not a properly JSON-encoded value, an exception will be raised when this attribute is accessed.

This attribute is useful when you invoke a Pyramid view callable via, for example, jQuery's `$.ajax` function, which has the potential to send a request with a JSON-encoded body.

Using `request.json_body` is equivalent to:

```
from json import loads
loads(request.body, encoding=request.charset)
```

Here's how to construct an AJAX request in JavaScript using *jQuery* that allows you to use the `request.json_body` attribute when the request is sent to a Pyramid application:

```
jQuery.ajax({type: 'POST',
             url: 'http://localhost:6543/', // the pyramid server
             data: JSON.stringify({'a': 1}),
             contentType: 'application/json; charset=utf-8'});
```

When such a request reaches a view in your application, the `request.json_body` attribute will be available in the view callable body.

```
@view_config(renderer='string')
def aview(request):
    print(request.json_body)
    return 'OK'
```

For the above view, printed to the console will be:

```
{u'a': 1}
```

For bonus points, here's a bit of client-side code that will produce a request that has a body suitable for reading via `request.json_body` using Python's `urllib2` instead of a JavaScript AJAX request:

```
import urllib2
import json

json_payload = json.dumps({'a':1})
headers = {'Content-Type':'application/json; charset=utf-8'}
req = urllib2.Request('http://localhost:6543/', json_payload, headers)
resp = urllib2.urlopen(req)
```

If you are doing Cross-origin resource sharing (CORS), then the standard requires the browser to do a pre-flight HTTP OPTIONS request. The easiest way to handle this is to add an extra `view_config` for the same route, with `request_method` set to `OPTIONS`, and set the desired response header before returning. You can find examples of response headers [Access control CORS](#), [Preflighted requests](#).

Cleaning up after a Request

Sometimes it's required to perform some cleanup at the end of a request when a database connection is involved.

For example, let's say you have a `mypackage` Pyramid application package that uses SQLAlchemy, and you'd like the current SQLAlchemy database session to be removed after each request. Put the following in the `mypackage.__init__` module:

```
1 from mypackage.models import DBSession
2
3 from pyramid.events import subscriber
4 from pyramid.events import NewRequest
5
6 def cleanup_callback(request):
7     DBSession.remove()
8
9 @subscriber(NewRequest)
10 def add_cleanup_callback(event):
11     event.request.add_finished_callback(cleanup_callback)
```

Registering the `cleanup_callback` finished callback at the start of a request (by causing the `add_cleanup_callback` to receive a `pyramid.events.NewRequest` event at the start of each request) will cause the `DBSession` to be removed whenever request processing has ended. Note that in the example above, for the `pyramid.events.subscriber` decorator to work, the `pyramid.config.Configurator.scan()` method must be called against your `mypackage` package during application initialization.



This is only an example. In particular, it is not necessary to cause `DBSession.remove` to be called in an application generated from any Pyramid scaffold, because these all use the `pyramid_tm` package. The cleanup done by `DBSession.remove` is unnecessary when `pyramid_tm middleware` is configured into the application.

More Details

More detail about the request object API is available as follows.

- `pyramid.request.Request` API documentation
- WebOb documentation. All methods and attributes of a `webob.Request` documented within the WebOb documentation will work with request objects created by Pyramid.

Response

The Pyramid response object can be imported as `pyramid.response.Response`. This class is a subclass of the `webob.Response` class. The subclass does not add or change any functionality, so the WebOb Response documentation will be completely relevant for this class as well.

A response object has three fundamental parts:

`response.status` The response code plus reason message, like `200 OK`. To set the code without a message, use `status_int`, i.e., `response.status_int = 200`.

`response.headerlist` A list of all the headers, like `[('Content-Type', 'text/html')]`. There's a case-insensitive *multidict* in `response.headers` that also allows you to access these same headers.

`response.app_iter` An iterable (such as a list or generator) that will produce the content of the response. This is also accessible as `response.body` (a string), `response.text` (a unicode object, informed by `response.charset`), and `response.body_file` (a file-like object; writing to it appends to `app_iter`).

Everything else in the object typically derives from this underlying state. Here are some highlights:

response.content_type The content type *not* including the `charset` parameter.

Typical use: `response.content_type = 'text/html'`.

Default value: `response.content_type = 'text/html'`.

response.charset The `charset` parameter of the content-type, it also informs encoding in `response.text`. `response.content_type_params` is a dictionary of all the parameters.

response.set_cookie(key, value, max_age=None, path='/', ...) Set a cookie. The keyword arguments control the various cookie parameters. The `max_age` argument is the length for the cookie to live in seconds (you may also use a `timedelta` object). The `Expires` key will also be set based on the value of `max_age`.

response.delete_cookie(key, path='/', domain=None) Delete a cookie from the client. This sets `max_age` to 0 and the cookie value to ''.

response.cache_expires(seconds=0) This makes the response cacheable for the given number of seconds, or if `seconds` is 0 then the response is uncacheable (this also sets the `Expires` header).

response(envIRON, start_response) The response object is a WSGI application. As an application, it acts according to how you create it. It *can* do conditional responses if you pass `conditional_response=True` when instantiating (or set that attribute later). It can also do HEAD and Range requests.

Headers

Like the request, most HTTP response headers are available as properties. These are parsed, so you can do things like `response.last_modified = os.path.getmtime(filename)`.

The details are available in the `webob.response` API documentation.

Instantiating the Response

Of course most of the time you just want to *make* a response. Generally any attribute of the response can be passed in as a keyword argument to the class, e.g.:


```
1 from pyramid.response import Response
2 response = Response(body='hello world!', content_type='text/plain')
```

The status defaults to '200 OK'.

The value of `content_type` defaults to `webob.response.Response.default_content_type`, which is `text/html`. You can subclass `pyramid.response.Response` and set `default_content_type` to override this behavior.

Exception Responses

To facilitate error responses like 404 Not Found, the module `pyramid.httpexceptions` contains classes for each kind of error response. These include boring but appropriate error bodies. The exceptions exposed by this module, when used under Pyramid, should be imported from the `pyramid.httpexceptions` module. This import location contains subclasses and replacements that mirror those in the `webob.exc` module.

Each class is named `pyramid.httpexceptions.HTTP*`, where `*` is the reason for the error. For instance, `pyramid.httpexceptions.HTTPNotFound` subclasses `pyramid.response.Response`, so you can manipulate the instances in the same way. A typical example is:

```
1 from pyramid.httpexceptions import HTTPNotFound
2 from pyramid.httpexceptions import HTTPMovedPermanently
3
4 response = HTTPNotFound('There is no such resource')
5 # or:
6 response = HTTPMovedPermanently(location=new_url)
```

More Details

More details about the response object API are available in the `pyramid.response` documentation. More details about exception responses are in the `pyramid.httpexceptions` API documentation. The WebOb documentation is also useful.

Sessions

A *session* is a namespace which is valid for some period of continual activity that can be used to represent a user's interaction with a web application.

This chapter describes how to configure sessions, what session implementations Pyramid provides out of the box, how to store and retrieve data from sessions, and two session-specific features: flash messages, and cross-site request forgery attack prevention.

Using the Default Session Factory

In order to use sessions, you must set up a *session factory* during your Pyramid configuration.

A very basic, insecure sample session factory implementation is provided in the Pyramid core. It uses a cookie to store session information. This implementation has the following limitations:

- The session information in the cookies used by this implementation is *not* encrypted, so it can be viewed by anyone with access to the cookie storage of the user's browser or anyone with access to the network along which the cookie travels.
- The maximum number of bytes that are storable in a serialized representation of the session is fewer than 4000. This is suitable only for very small data sets.

It is digitally signed, however, and thus its data cannot easily be tampered with.

You can configure this session factory in your Pyramid application by using the `pyramid.config.Configurator.set_session_factory()` method.

```
1 from pyramid.session import SignedCookieSessionFactory
2 my_session_factory = SignedCookieSessionFactory('itsaseekreet')
3
4 from pyramid.config import Configurator
5 config = Configurator()
6 config.set_session_factory(my_session_factory)
```



By default the `SignedCookieSessionFactory()` implementation is *unencrypted*. You should not use it when you keep sensitive information in the session object, as the information can be easily read by both users of your application and third parties who have access to your users' network traffic. And, if you use this sessioning implementation, and you inadvertently create a cross-site scripting vulnerability in your application, because the session data is stored unencrypted in a cookie, it will also be easier for evildoers to obtain the current user's cross-site scripting token. In short, use a different session factory implementation (preferably one which keeps session data on the server) for anything but the most basic of applications where "session security doesn't matter", and you are sure your application has no cross-site scripting vulnerabilities.

Using a Session Object

Once a session factory has been configured for your application, you can access session objects provided by the session factory via the `session` attribute of any *request* object. For example:

```
1 from pyramid.response import Response
2
3 def myview(request):
4     session = request.session
5     if 'abc' in session:
6         session['fred'] = 'yes'
7     session['abc'] = '123'
8     if 'fred' in session:
9         return Response('Fred was in the session')
10    else:
11        return Response('Fred was not in the session')
```

The first time this view is invoked produces `Fred was not in the session`. Subsequent invocations produce `Fred was in the session`, assuming of course that the client side maintains the session's identity across multiple requests.

You can use a session much like a Python dictionary. It supports all dictionary methods, along with some extra attributes and methods.

Extra attributes:

created An integer timestamp indicating the time that this session was created.

new A boolean. If `new` is `True`, this session is new. Otherwise, it has been constituted from data that was already serialized.

Extra methods:

changed() Call this when you mutate a mutable value in the session namespace. See the gotchas below for details on when and why you should call this.

invalidate() Call this when you want to invalidate the session (dump all data, and perhaps set a clearing cookie).

The formal definition of the methods and attributes supported by the session object are in the *pyramid.interfaces.ISession* documentation.

Some gotchas:

- Keys and values of session data must be *pickleable*. This means, typically, that they are instances of basic types of objects, such as strings, lists, dictionaries, tuples, integers, etc. If you place an object in a session data key or value that is not pickleable, an error will be raised when the session is serialized.
- If you place a mutable value (for example, a list or a dictionary) in a session object, and you subsequently mutate that value, you must call the `changed()` method of the session object. In this case, the session has no way to know that it was modified. However, when you modify a session object directly, such as setting a value (i.e., `__setitem__`), or removing a key (e.g., `del` or `pop`), the session will automatically know that it needs to re-serialize its data, thus calling `changed()` is unnecessary. There is no harm in calling `changed()` in either case, so when in doubt, call it after you've changed sessioning data.

Using Alternate Session Factories

The following session factories exist at the time of this writing.

Session Factory	Back-end	Description
pyramid_redis_sessions	Redis	Server-side session library for Pyramid, using Redis for storage.
pyramid_beaker	Beaker	Session factory for Pyramid backed by the Beaker sessioning system.

Creating Your Own Session Factory

If none of the default or otherwise available sessioning implementations for Pyramid suit you, you may create your own session object by implementing a *session factory*. Your session factory should return a *session*. The interfaces for both types are available in *pyramid.interfaces.ISessionFactory* and *pyramid.interfaces.ISession*. You might use the cookie implementation in the *pyramid.session* module as inspiration.

Flash Messages

“Flash messages” are simply a queue of message strings stored in the *session*. To use flash messaging, you must enable a *session factory* as described in *Using the Default Session Factory* or *Using Alternate Session Factories*.

Flash messaging has two main uses: to display a status message only once to the user after performing an internal redirect, and to allow generic code to log messages for single-time display without having direct access to an HTML template. The user interface consists of a number of methods of the *session* object.

Using the `session.flash` Method

To add a message to a flash message queue, use a session object’s `flash()` method:

```
request.session.flash('mymessage')
```

The `flash()` method appends a message to a flash queue, creating the queue if necessary.

`flash()` accepts three arguments:

`flash` (*message*, *queue*='', *allow_duplicate*=True)

The *message* argument is required. It represents a message you wish to later display to a user. It is usually a string but the *message* you provide is not modified in any way.

The *queue* argument allows you to choose a queue to which to append the message you provide. This can be used to push different kinds of messages into flash storage for later display in different places on a page. You can pass any name for your queue, but it must be a string. Each queue is independent, and can be popped by `pop_flash()` or examined via `peek_flash()` separately. *queue* defaults to the empty string. The empty string represents the default flash message queue.

```
request.session.flash(msg, 'myappsqueue')
```

The *allow_duplicate* argument defaults to True. If this is False, and you attempt to add a message value which is already present in the queue, it will not be added.

Using the `session.pop_flash` Method

Once one or more messages have been added to a flash queue by the `session.flash()` API, the `session.pop_flash()` API can be used to pop an entire queue and return it for use.

To pop a particular queue of messages from the flash object, use the session object's `pop_flash()` method. This returns a list of the messages that were added to the flash queue, and empties the queue.

`pop_flash(queue='')`

```
>>> request.session.flash('info message')
>>> request.session.pop_flash()
['info message']
```

Calling `session.pop_flash()` again like above without a corresponding call to `session.flash()` will return an empty list, because the queue has already been popped.

```
>>> request.session.flash('info message')
>>> request.session.pop_flash()
['info message']
>>> request.session.pop_flash()
[]
```

Using the `session.peak_flash` Method

Once one or more messages have been added to a flash queue by the `session.flash()` API, the `session.peak_flash()` API can be used to “peek” at that queue. Unlike `session.pop_flash()`, the queue is not popped from flash storage.

`peak_flash(queue='')`

```
>>> request.session.flash('info message')
>>> request.session.peak_flash()
['info message']
>>> request.session.peak_flash()
['info message']
>>> request.session.pop_flash()
['info message']
>>> request.session.peak_flash()
[]
```

Preventing Cross-Site Request Forgery Attacks

Cross-site request forgery attacks are a phenomenon whereby a user who is logged in to your website might inadvertently load a URL because it is linked from, or embedded in, an attacker's website. If the URL is one that may modify or delete data, the consequences can be dire.

You can avoid most of these attacks by issuing a unique token to the browser and then requiring that it be present in all potentially unsafe requests. Pyramid sessions provide facilities to create and check CSRF tokens.

To use CSRF tokens, you must first enable a *session factory* as described in *Using the Default Session Factory* or *Using Alternate Session Factories*.

Using the `session.get_csrf_token` Method

To get the current CSRF token from the session, use the `session.get_csrf_token()` method.

```
token = request.session.get_csrf_token()
```

The `session.get_csrf_token()` method accepts no arguments. It returns a CSRF *token* string. If `session.get_csrf_token()` or `session.new_csrf_token()` was invoked previously for this session, then the existing token will be returned. If no CSRF token previously existed for this session, then a new token will be set into the session and returned. The newly created token will be opaque and randomized.

You can use the returned token as the value of a hidden field in a form that posts to a method that requires elevated privileges, or supply it as a request header in AJAX requests.

For example, include the CSRF token as a hidden field:

```
<form method="post" action="/myview">
  <input type="hidden" name="csrf_token" value="{request.session.get_csrf_
  token()}">
  <input type="submit" value="Delete Everything">
</form>
```

Or include it as a header in a jQuery AJAX request:

```
var csrfToken = ${request.session.get_csrf_token()};
$.ajax({
  type: "POST",
  url: "/myview",
  headers: { 'X-CSRF-Token': csrfToken }
}).done(function() {
  alert("Deleted");
});
```

The handler for the URL that receives the request should then require that the correct CSRF token is supplied.

Using the `session.new_csrf_token` Method

To explicitly create a new CSRF token, use the `session.new_csrf_token()` method. This differs only from `session.get_csrf_token()` inasmuch as it clears any existing CSRF token, creates a new CSRF token, sets the token into the session, and returns the token.

```
token = request.session.new_csrf_token()
```

Checking CSRF Tokens Manually

In request handling code, you can check the presence and validity of a CSRF token with `pyramid.session.check_csrf_token()`. If the token is valid, it will return `True`, otherwise it will raise `HTTPBadRequest`. Optionally, you can specify `raises=False` to have the check return `False` instead of raising an exception.

By default, it checks for a POST parameter named `csrf_token` or a header named `X-CSRF-Token`.

```
from pyramid.session import check_csrf_token

def myview(request):
    # Require CSRF Token
    check_csrf_token(request)

    # ...
```


Checking CSRF Tokens Automatically

New in version 1.7.

Pyramid supports automatically checking CSRF tokens on requests with an unsafe method as defined by RFC2616. Any other request may be checked manually. This feature can be turned on globally for an application using the `pyramid.config.Configurator.set_default_csrf_options()` directive. For example:

```
from pyramid.config import Configurator

config = Configurator()
config.set_default_csrf_options(require_csrf=True)
```

CSRF checking may be explicitly enabled or disabled on a per-view basis using the `require_csrf` view option. A value of `True` or `False` will override the default set by `set_default_csrf_options`. For example:

```
@view_config(route_name='hello', require_csrf=False)
def myview(request):
    # ...
```

When CSRF checking is active, the token and header used to find the supplied CSRF token will be `csrf_token` and `X-CSRF-Token`, respectively, unless otherwise overridden by `set_default_csrf_options`. The token is checked against the value in `request.POST` which is the submitted form body. If this value is not present, then the header will be checked.

In addition to token based CSRF checks, if the request is using HTTPS then the automatic CSRF checking will also check the referrer of the request to ensure that it matches one of the trusted origins. By default the only trusted origin is the current host, however additional origins may be configured by setting `pyramid.csrf_trusted_origins` to a list of domain names (and ports if they are non standard). If a host in the list of domains starts with a `.` then that will allow all subdomains as well as the domain without the `..`


If CSRF checks fail then a `pyramid.exceptions.BadCSRFToken` or `pyramid.exceptions.BadCSRFOrigin` exception will be raised. This exception may be caught and handled by an *exception view* but, by default, will result in a 400 Bad Request response being sent to the client.

Checking CSRF Tokens with a View Predicate

Deprecated since version 1.7: Use the `require_csrf` option or read *Checking CSRF Tokens Automatically* instead to have `pyramid.exceptions.BadCSRFToken` exceptions raised.

A convenient way to require a valid CSRF token for a particular view is to include `check_csrf=True` as a view predicate. See `pyramid.config.Configurator.add_view()`.

```
@view_config(request_method='POST', check_csrf=True, ...)
def myview(request):
    ...
```

 A mismatch of a CSRF token is treated like any other predicate miss, and the predicate system, when it doesn't find a view, raises `HTTPNotFound` instead of `HTTPBadRequest`, so `check_csrf=True` behavior is different from calling `pyramid.session.check_csrf_token()`.

Using Events

An *event* is an object broadcast by the Pyramid framework at interesting points during the lifetime of an application. You don't need to use events in order to create most Pyramid applications, but they can be useful when you want to perform slightly advanced operations. For example, subscribing to an event can allow you to run some code as the result of every new request.

Events in Pyramid are always broadcast by the framework. However, they only become useful when you register a *subscriber*. A subscriber is a function that accepts a single argument named *event*:

```
1 def mysubscriber(event):
2     print(event)
```

The above is a subscriber that simply prints the event to the console when it's called.

The mere existence of a subscriber function, however, is not sufficient to arrange for it to be called. To arrange for the subscriber to be called, you'll need to use the `pyramid.config.Configurator.add_subscriber()` method or you'll need to use the `pyramid.events.subscriber()` decorator to decorate a function found via a *scan*.

Configuring an Event Listener Imperatively

You can imperatively configure a subscriber function to be called for some event type via the `add_subscriber()` method:

```
1 from pyramid.events import NewRequest
2
3 from subscribers import mysubscriber
4
5 # "config" below is assumed to be an instance of a
6 # pyramid.config.Configurator object
7
8 config.add_subscriber(mysubscriber, NewRequest)
```

The first argument to `add_subscriber()` is the subscriber function (or a *dotted Python name* which refers to a subscriber callable); the second argument is the event type.

See also:

See also *Configurator*.

Configuring an Event Listener Using a Decorator

You can configure a subscriber function to be called for some event type via the `pyramid.events.subscriber()` function.

```
1 from pyramid.events import NewRequest
2 from pyramid.events import subscriber
3
4 @subscriber(NewRequest)
5 def mysubscriber(event):
6     event.request.foo = 1
```

When the `subscriber()` decorator is used, a *scan* must be performed against the package containing the decorated function for the decorator to have any effect.

Either of the above registration examples implies that every time the Pyramid framework emits an event object that supplies an `pyramid.events.NewRequest` interface, the `mysubscriber` function will be called with an *event* object.

As you can see, a subscription is made in terms of a *class* (such as `pyramid.events.NewResponse`). The event object sent to a subscriber will always be an object that possesses an *interface*. For `pyramid.events.NewResponse`, that interface is `pyramid.interfaces.INewResponse`. The interface documentation provides information about available attributes and methods of the event objects.

The return value of a subscriber function is ignored. Subscribers to the same event type are not guaranteed to be called in any particular order relative to each other.

All the concrete Pyramid event types are documented in the `pyramid.events` API documentation.

An Example

If you create event listener functions in a `subscribers.py` file in your application like so:

```
1 def handle_new_request(event):
2     print('request', event.request)
3
4 def handle_new_response(event):
5     print('response', event.response)
```

You may configure these functions to be called at the appropriate times by adding the following code to your application's configuration startup:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_subscriber('myproject.subscribers.handle_new_request',
4                       'pyramid.events.NewRequest')
5 config.add_subscriber('myproject.subscribers.handle_new_response',
6                       'pyramid.events.NewResponse')
```

Either mechanism causes the functions in `subscribers.py` to be registered as event subscribers. Under this configuration, when the application is run, each time a new request or response is detected, a message will be printed to the console.

Each of our subscriber functions accepts an `event` object and prints an attribute of the event object. This begs the question: how can we know which attributes a particular event has?

We know that `pyramid.events.NewRequest` event objects have a `request` attribute, which is a *request* object, because the interface defined at `pyramid.interfaces.INewRequest` says it must. Likewise, we know that `pyramid.events.NewResponse` events have a `response` attribute, which is a response object constructed by your application, because the interface defined at `pyramid.interfaces.INewResponse` says it must (`pyramid.events.NewResponse` objects also have a `request`).

Creating Your Own Events

In addition to using the events that the Pyramid framework creates, you can create your own events for use in your application. This can be useful to decouple parts of your application.

For example, suppose your application has to do many things when a new document is created. Rather than putting all this logic in the view that creates the document, you can create the document in your view and then fire a custom event. Subscribers to the custom event can take other actions, such as indexing the document, sending email, or sending a message to a remote system.

An event is simply an object. There are no required attributes or method for your custom events. In general, your events should keep track of the information that subscribers will need. Here are some example custom event classes:

```
1 class DocCreated(object):
2     def __init__(self, doc, request):
3         self.doc = doc
4         self.request = request
5
6 class UserEvent(object):
7     def __init__(self, user):
8         self.user = user
9
10 class UserLoggedIn(UserEvent):
11     pass
```

Some Pyramid applications choose to define custom events classes in an `events` module.

You can subscribe to custom events in the same way that you subscribe to Pyramid events—either imperatively or with a decorator. You can also use custom events with *subscriber predicates*. Here's an example of subscribing to a custom event with a decorator:

```
1 from pyramid.events import subscriber
2 from .events import DocCreated
3 from .index import index_doc
4
5 @subscriber(DocCreated)
6 def index_doc(event):
7     # index the document using our application's index_doc function
8     index_doc(event.doc, event.request)
```

The above example assumes that the application defines a `DocCreated` event class and an `index_doc` function.

To fire your custom events use the `pyramid.registry.Registry.notify()` method, which is most often accessed as `request.registry.notify`. For example:


```
1 from .events import DocCreated
2
3 def new_doc_view(request):
4     doc = MyDoc()
5     event = DocCreated(doc, request)
6     request.registry.notify(event)
7     return {'document': doc}
```

This example view will notify all subscribers to the custom `DocCreated` event.

Note that when you fire an event, all subscribers are run synchronously so it's generally not a good idea to create event handlers that may take a long time to run. Although event handlers could be used as a central place to spawn tasks on your own message queues.

Environment Variables and `.ini` File Settings

Pyramid behavior can be configured through a combination of operating system environment variables and `.ini` configuration file application section settings. The meaning of the environment variables and the configuration file settings overlap.

 Where a configuration file setting exists with the same meaning as an environment variable, and both are present at application startup time, the environment variable setting takes precedence.

The term “configuration file setting name” refers to a key in the `.ini` configuration for your application. The configuration file setting names documented in this chapter are reserved for Pyramid use. You should not use them to indicate application-specific configuration settings.

Reloading Templates

When this value is true, templates are automatically reloaded whenever they are modified without restarting the application, so you can see changes to templates take effect immediately during development. This flag is meaningful to Chameleon and Mako templates, as well as most third-party template rendering extensions.

Environment Variable Name	Config File Setting Name
PYRAMID_RELOAD_TEMPLATES	pyramid.reload_templates or reload_templates


Reloading Assets

Don't cache any asset file data when this value is true.

See also:

See also *Overriding Assets*.

Environment Variable Name	Config File Setting Name
PYRAMID_RELOAD_ASSETS	pyramid.reload_assets or reload_assets

 For backwards compatibility purposes, aliases can be used for configuring asset reloading: `PYRAMID_RELOAD_RESOURCES` (envvar) and `pyramid.reload_resources` (config file).

Debugging Authorization

Print view authorization failure and success information to stderr when this value is true.

See also:

See also *Debugging View Authorization Failures*.

Environment Variable Name	Config File Setting Name
PYRAMID_DEBUG_AUTHORIZATION	pyramid.debug_authorization or debug_authorization

Debugging Not Found Errors

Print view-related `NotFound` debug messages to `stderr` when this value is true.

See also:

See also *NotFound Errors*.

Environment Variable Name	Config File Setting Name
<code>PYRAMID_DEBUG_NOTFOUND</code>	<code>pyramid.debug_notfound</code> or <code>debug_notfound</code>

Debugging Route Matching

Print debugging messages related to *url dispatch* route matching when this value is true.

See also:

See also *Debugging Route Matching*.

Environment Variable Name	Config File Setting Name
<code>PYRAMID_DEBUG_ROUTE MATCH</code>	<code>pyramid.debug_routematch</code> or <code>debug_routematch</code>

Preventing HTTP Caching

Prevent the `http_cache` view configuration argument from having any effect globally in this process when this value is true. No HTTP caching-related response headers will be set by the Pyramid `http_cache` view configuration feature when this is true.

See also:

See also *Influencing HTTP Caching*.

Environment Variable Name	Config File Setting Name
<code>PYRAMID_PREVENT_HTTP_CACHE</code>	<code>pyramid.prevent_http_cache</code> or <code>prevent_http_cache</code>

Preventing Cache Busting

Prevent the `cachebust` static view configuration argument from having any effect globally in this process when this value is true. No cache buster will be configured or used when this is true.

New in version 1.6.

See also:

See also *Cache Busting*.

Environment Variable Name	Config File Setting Name
PYRAMID_PREVENT_CACHEBUST	pyramid.prevent_cachebust or prevent_cachebust

Debugging All

Turns on all `debug*` settings.

Environment Variable Name	Config File Setting Name
PYRAMID_DEBUG_ALL	pyramid.debug_all or debug_all

Reloading All

Turns on all `reload*` settings.

Environment Variable Name	Config File Setting Name
PYRAMID_RELOAD_ALL	pyramid.reload_all or reload_all

Default Locale Name

The value supplied here is used as the default locale name when a *locale negotiator* is not registered.

See also:

See also *Localization-Related Deployment Settings*.

Environment Variable Name	Config File Setting Name
PYRAMID_DEFAULT_LOCALE_NAME	pyramid.default_locale_name or default_locale_name

Including Packages

`pyramid.includes` instructs your application to include other packages. Using the setting is equivalent to using the `pyramid.config.Configurator.include()` method.

Config File Setting Name
<code>pyramid.includes</code>

The value assigned to `pyramid.includes` should be a sequence. The sequence can take several different forms.

1. It can be a string.

If it is a string, the package names can be separated by spaces:

```
package1 package2 package3
```

The package names can also be separated by carriage returns:

```
package1
package2
package3
```

2. It can be a Python list, where the values are strings:

```
['package1', 'package2', 'package3']
```

Each value in the sequence should be a *dotted Python name*.

`pyramid.includes` VS. `pyramid.config.Configurator.include()`

Two methods exist for including packages: `pyramid.includes` and `pyramid.config.Configurator.include()`. This section explains their equivalence.

Using PasteDeploy

Using the following `pyramid.includes` setting in the PasteDeploy `.ini` file in your application:

```
[app:main]
pyramid.includes = pyramid_debugtoolbar
                  pyramid_tm
```

Is equivalent to using the following statements in your configuration code:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     config = Configurator(settings=settings)
5     # ...
6     config.include('pyramid_debugtoolbar')
7     config.include('pyramid_tm')
8     # ...
```

It is fine to use both or either form.

Plain Python

Using the following `pyramid.includes` setting in your plain-Python Pyramid application:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     settings = {'pyramid.includes': 'pyramid_debugtoolbar pyramid_tm'}
5     config = Configurator(settings=settings)
```

Is equivalent to using the following statements in your configuration code:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     settings = {}
5     config = Configurator(settings=settings)
6     config.include('pyramid_debugtoolbar')
7     config.include('pyramid_tm')
```

It is fine to use both or either form.

Explicit Tween Configuration

This value allows you to perform explicit *tween* ordering in your configuration. Tweens are bits of code used by add-on authors to extend Pyramid. They form a chain, and require ordering.

Ideally you won't need to use the `pyramid.tweens` setting at all. Tweens are generally ordered and included “implicitly” when an add-on package which registers a tween is “included”. Packages are included when you name a `pyramid.includes` setting in your configuration or when you call `pyramid.config.Configurator.include()`.

Authors of included add-ons provide “implicit” tween configuration ordering hints to Pyramid when their packages are included. However, the implicit tween ordering is only best-effort. Pyramid will attempt to provide an implicit order of tweens as best it can using hints provided by add-on authors, but because it's only best-effort, if very precise tween ordering is required, the only surefire way to get it is to use an explicit tween order. You may be required to inspect your tween ordering (see *Displaying “Tweens”*) and add a `pyramid.tweens` configuration value at the behest of an add-on author.

Config File Setting Name
<code>pyramid.tweens</code>

The value assigned to `pyramid.tweens` should be a sequence. The sequence can take several different forms.

1. It can be a string.

If it is a string, the tween names can be separated by spaces:

```
pkg.tween_factory1 pkg.tween_factory2 pkg.tween_factory3
```

The tween names can also be separated by carriage returns:

```
pkg.tween_factory1
pkg.tween_factory2
pkg.tween_factory3
```

2. It can be a Python list, where the values are strings:

```
['pkg.tween_factory1', 'pkg.tween_factory2', 'pkg.tween_factory3']
```

Each value in the sequence should be a *dotted Python name*.

PasteDeploy Configuration vs. Plain-Python Configuration

Using the following `pyramid.tweens` setting in the PasteDeploy `.ini` file in your application:

```
[app:main]
pyramid.tweens = pyramid_debugtoolbar.toolbar.tween_factory
                  pyramid.tweens.excview_tween_factory
                  pyramid_tm.tm_tween_factory
```

Is equivalent to using the following statements in your configuration code:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     settings['pyramid.tweens'] = [
5         'pyramid_debugtoolbar.toolbar.tween_factory',
6         'pyramid.tweens.excview_tween_factory',
7         'pyramid_tm.tm_tween_factory',
8     ]
9     config = Configurator(settings=settings)
```

It is fine to use both or either form.

Examples

Let's presume your configuration file is named `MyProject.ini`, and there is a section representing your application named `[app:main]` within the file that represents your Pyramid application. The configuration file settings documented in the above "Config File Setting Name" column would go in the `[app:main]` section. Here's an example of such a section:

```
1 [app:main]
2 use = egg:MyProject
3 pyramid.reload_templates = true
4 pyramid.debug_authorization = true
```


You can also use environment variables to accomplish the same purpose for settings documented as such. For example, you might start your Pyramid application using the following command line:

```
$ PYRAMID_DEBUG_AUTHORIZATION=1 PYRAMID_RELOAD_TEMPLATES=1 \
  $VENV/bin/pserve MyProject.ini
```

If you started your application this way, your Pyramid application would behave in the same manner as if you had placed the respective settings in the `[app:main]` section of your application's `.ini` file.

If you want to turn all debug settings (every setting that starts with `pyramid.debug_`) on in one fell swoop, you can use `PYRAMID_DEBUG_ALL=1` as an environment variable setting or you may use `pyramid.debug_all=true` in the config file. Note that this does not affect settings that do not start with `pyramid.debug_*` such as `pyramid.reload_templates`.

If you want to turn all `pyramid.reload` settings (every setting that starts with `pyramid.reload_`) on in one fell swoop, you can use `PYRAMID_RELOAD_ALL=1` as an environment variable setting or you may use `pyramid.reload_all=true` in the config file. Note that this does not affect settings that do not start with `pyramid.reload_*` such as `pyramid.debug_notfound`.

 Specifying configuration settings via environment variables is generally most useful during development, where you may wish to augment or override the more permanent settings in the configuration file. This is useful because many of the reload and debug settings may have performance or security (i.e., disclosure) implications that make them undesirable in a production environment.

Understanding the Distinction Between `reload_templates` and `reload_assets`

The difference between `pyramid.reload_assets` and `pyramid.reload_templates` is a bit subtle. Templates are themselves also treated by Pyramid as asset files (along with other static files), so the distinction can be confusing. It's helpful to read *Overriding Assets* for some context about assets in general.

When `pyramid.reload_templates` is true, Pyramid takes advantage of the underlying templating system's ability to check for file modifications to an individual template file. When `pyramid.reload_templates` is true, but `pyramid.reload_assets` is *not* true, the template filename returned by the `pkg_resources` package (used under the hood by asset resolution) is cached by Pyramid on the first request. Subsequent requests for the same template file will return a cached template filename. The underlying templating system checks for modifications to this particular file for every request. Setting `pyramid.reload_templates` to `True` doesn't affect performance dramatically (although it should still not be used in production because it has some effect).

However, when `pyramid.reload_assets` is true, Pyramid will not cache the template filename, meaning you can see the effect of changing the content of an overridden asset directory for templates without restarting the server after every change. Subsequent requests for the same template file may return different filenames based on the current state of overridden asset directories. Setting `pyramid.reload_assets` to `True` affects performance *dramatically*, slowing things down by an order of magnitude for each template rendering. However, it's convenient to enable when moving files around in overridden asset directories. `pyramid.reload_assets` makes the system *very slow* when templates are in use. Never set `pyramid.reload_assets` to `True` on a production system.

Adding a Custom Setting

From time to time, you may need to add a custom setting to your application. Here's how:

- If you're using an `.ini` file, change the `.ini` file, adding the setting to the `[app:foo]` section representing your Pyramid application. For example:

```
[app:main]
# .. other settings
debug_frobnosticator = True
```

- In the `main()` function that represents the place that your Pyramid WSGI application is created, anticipate that you'll be getting this key/value pair as a setting and do any type conversion necessary.

If you've done any type conversion of your custom value, reset the converted values into the `settings` dictionary *before* you pass the dictionary as settings to the *Configurator*. For example:

```
def main(global_config, **settings):
    # ...
    from pyramid.settings import asbool
    debug_frobnosticator = asbool(settings.get(
        'debug_frobnosticator', 'false'))
    settings['debug_frobnosticator'] = debug_frobnosticator
    config = Configurator(settings=settings)
```



It's especially important that you mutate the `settings` dictionary with the converted version of the variable *before* passing it to the *Configurator*: the configurator makes a *copy* of `settings`, it doesn't use the one you pass directly.

- When creating an `includeme` function that will be later added to your application's configuration you may access the `settings` dictionary through the instance of the *Configurator* that is passed into the function as its only argument. For Example:

```
def includeme(config):
    settings = config.registry.settings
    debug_frobnosticator = settings['debug_frobnosticator']
```

- In the runtime code from where you need to access the new settings value, find the value in the `registry.settings` dictionary and use it. In *view* code (or any other code that has access to the request), the easiest way to do this is via `request.registry.settings`. For example:

```
settings = request.registry.settings
debug_frobnosticator = settings['debug_frobnosticator']
```

If you wish to use the value in code that does not have access to the request and you wish to use the value, you'll need to use the `pyramid.threadlocal.get_current_registry()` API to obtain the current registry, then ask for its settings attribute. For example:

```
registry = pyramid.threadlocal.get_current_registry()
settings = registry.settings
debug_frobnosticator = settings['debug_frobnosticator']
```

Logging

Pyramid allows you to make use of the Python standard library logging module. This chapter describes how to configure logging and how to send log messages to loggers that you've configured.



This chapter assumes you've used a *scaffold* to create a project which contains `development.ini` and `production.ini` files which help configure logging. All of the scaffolds which ship with Pyramid do this. If you're not using a scaffold, or if you've used a third-party scaffold which does not create these files, the configuration information in this chapter may not be applicable.

Logging Configuration

A Pyramid project created from a *scaffold* is configured to allow you to send messages to Python standard library logging package loggers from within your application. In particular, the *PasteDeploy* `development.ini` and `production.ini` files created when you use a scaffold include a basic configuration for the Python logging package.

PasteDeploy `.ini` files use the Python standard library `ConfigParser` format. This is the same format used as the Python logging module's Configuration file format. The application-related and logging-related sections in the configuration file can coexist peacefully, and the logging-related sections in the file are used from when you run `pserve`.

The `pserve` command calls the `pyramid.paster.setup_logging()` function, a thin wrapper around the `logging.config.fileConfig()` using the specified `.ini` file, if it contains a

[loggers] section (all of the scaffold-generated .ini files do). setup_logging reads the logging configuration from the ini file upon which pserve was invoked.

Default logging configuration is provided in both the default development.ini and the production.ini file. The logging configuration in the development.ini file is as follows:

```
1 # Begin logging configuration
2
3 [loggers]
4 keys = root, {{package_logger}}
5
6 [handlers]
7 keys = console
8
9 [formatters]
10 keys = generic
11
12 [logger_root]
13 level = INFO
14 handlers = console
15
16 [logger_{{package_logger}}]
17 level = DEBUG
18 handlers =
19 qualname = {{package}}
20
21 [handler_console]
22 class = StreamHandler
23 args = (sys.stderr,)
24 level = NOTSET
25 formatter = generic
26
27 [formatter_generic]
28 format = %(asctime)s %(levelname)-5.5s [% (name)s] [% (threadName)s]
29 ↪ % (message)s
30 # End logging configuration
```

The production.ini file uses the WARN level in its logger configuration, but it is otherwise identical.

The name {{package_logger}} above will be replaced with the name of your project's *package*, which is derived from the name you provide to your project. For instance, if you do:

```
1 pcreate -s starter MyApp
```

The logging configuration will literally be:

```

1  # Begin logging configuration
2
3  [loggers]
4  keys = root, myapp
5
6  [handlers]
7  keys = console
8
9  [formatters]
10 keys = generic
11
12 [logger_root]
13 level = INFO
14 handlers = console
15
16 [logger_myapp]
17 level = DEBUG
18 handlers =
19 qualname = myapp
20
21 [handler_console]
22 class = StreamHandler
23 args = (sys.stderr,)
24 level = NOTSET
25 formatter = generic
26
27 [formatter_generic]
28 format = %(asctime)s %(levelname)-5.5s [% (name)s] [% (threadName)s]
29         ↳ %(message)s
30 # End logging configuration

```

In this logging configuration:

- a logger named `root` is created that logs messages at a level above or equal to the `INFO` level to `stderr`, with the following format:

```
2007-08-17 15:04:08,704 INFO [packagename] Loading resource, id: 86
```

- a logger named `myapp` is configured that logs messages sent at a level above or equal to `DEBUG` to `stderr` in the same format as the root logger.

The root logger will be used by all applications in the Pyramid process that ask for a logger (via `logging.getLogger`) that has a name which begins with anything except your project's package name (e.g., `myapp`). The logger with the same name as your package name is reserved for your own usage in your Pyramid application. Its existence means that you can log to a known logging location from any Pyramid application generated via a scaffold.

Pyramid and many other libraries (such as Beaker, SQLAlchemy, Paste) log a number of messages to the root logger for debugging purposes. Switching the root logger level to `DEBUG` reveals them:

```
[logger_root]
#level = INFO
level = DEBUG
handlers = console
```

Some scaffolds configure additional loggers for additional subsystems they use (such as SQLAlchemy). Take a look at the `production.ini` and `development.ini` files rendered when you create a project from a scaffold.

Sending Logging Messages

Python's special `__name__` variable refers to the current module's fully qualified name. From any module in a package named `myapp`, the `__name__` builtin variable will always be something like `myapp`, or `myapp.subpackage` or `myapp.package.subpackage` if your project is named `myapp`. Sending a message to this logger will send it to the `myapp` logger.

To log messages to the package-specific logger configured in your `.ini` file, simply create a logger object using the `__name__` builtin and call methods on it.

```
1 import logging
2 log = logging.getLogger(__name__)
3
4 def myview(request):
5     content_type = 'text/plain'
6     content = 'Hello World!'
7     log.debug('Returning: %s (content-type: %s)', content, content_type)
8     request.response.content_type = content_type
9     return request.response
```

This will result in the following printed to the console, on `stderr`:

```
16:20:20,440 DEBUG [myapp.views] Returning: Hello World!
              (content-type: text/plain)
```

Filtering log messages

Often there's too much log output to sift through, such as when switching the root logger's level to `DEBUG`. For example, you're diagnosing database connection issues in your application and only want to see SQLAlchemy's `DEBUG` messages in relation to database connection pooling. You can leave the root logger's level at the less verbose `INFO` level and set that particular SQLAlchemy logger to `DEBUG` on its own, apart from the root logger:

```
[logger_sqlalchemy.pool]
level = DEBUG
handlers =
qualname = sqlalchemy.pool
```

then add it to the list of loggers:

```
[loggers]
keys = root, myapp, sqlalchemy.pool
```

No handlers need to be configured for this logger as by default non-root loggers will propagate their log records up to their parent logger's handlers. The root logger is the top level parent of all loggers.

This technique is used in the default `development.ini`. The root logger's level is set to `INFO`, whereas the application's log level is set to `DEBUG`:

```
# Begin logging configuration

[loggers]
keys = root, myapp

[logger_myapp]
level = DEBUG
handlers =
qualname = myapp
```

All of the child loggers of the `myapp` logger will inherit the `DEBUG` level unless they're explicitly set differently. Meaning the `myapp.views`, `myapp.models`, and all your app's modules' loggers by default have an effective level of `DEBUG` too.

For more advanced filtering, the logging module provides a `logging.Filter` object; however it cannot be used directly from the configuration file.

Advanced Configuration

To capture log output to a separate file, use `logging.FileHandler` (or `logging.handlers.RotatingFileHandler`):

```
[handler_filelog]
class = FileHandler
args = ('%(here)s/myapp.log', 'a')
level = INFO
formatter = generic
```

Before it's recognized, it needs to be added to the list of handlers:

```
[handlers]
keys = console, myapp, filelog
```

and finally utilized by a logger.

```
[logger_root]
level = INFO
handlers = console, filelog
```

These final three lines of configuration direct all of the root logger's output to the `myapp.log` as well as the console.

Logging Exceptions

To log or email exceptions generated by your Pyramid application, use the *pyramid_exclog* package. Details about its configuration are in its documentation.

Request Logging with Paste's TransLogger

The *WSGI* design is modular. Waitress logs error conditions, debugging output, etc., but not web traffic. For web traffic logging, Paste provides the TransLogger *middleware*. TransLogger produces logs in the Apache Combined Log Format. But TransLogger does not write to files; the Python logging system must be configured to do this. The Python `logging.FileHandler` logging handler can be used alongside TransLogger to create an `access.log` file similar to Apache's.

Like any standard *middleware* with a Paste entry point, TransLogger can be configured to wrap your application using `.ini` file syntax. First rename your Pyramid `.ini` file's `[app:main]` section to `[app:mypyramidapp]`, then add a `[filter:translogger]` section, then use a `[pipeline:main]` section file to form a WSGI pipeline with both the translogger and your application in it. For instance, change from this:

```
[app:main]
use = egg:MyProject
```

To this:

```
[app:mypyramidapp]
use = egg:MyProject

[filter:translogger]
use = egg:Paste#translogger
setup_console_handler = False

[pipeline:main]
pipeline = translogger
          mypyramidapp
```

Using PasteDeploy this way to form and serve a pipeline is equivalent to wrapping your app in a TransLogger instance via the bottom of the `main` function of your project's `__init__.py` file:

```
...
app = config.make_wsgi_app()
from paste.translogger import TransLogger
app = TransLogger(app, setup_console_handler=False)
return app
```



TransLogger will automatically setup a logging handler to the console when called with no arguments, so it “just works” in environments that don’t configure logging. Since our logging handlers are configured, we disable the automation via `setup_console_handler = False`.

With the filter in place, TransLogger’s logger (named the `wsgi` logger) will propagate its log messages to the parent logger (the root logger), sending its output to the console when we request a page:

```
00:50:53,694 INFO [myapp.views] Returning: Hello World!
          (content-type: text/plain)
00:50:53,695 INFO [wsgi] 192.168.1.111 - - [11/Aug/2011:20:09:33 -0700]
→ "GET /hello
HTTP/1.1" 404 - "-"
"Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.6) Gecko/
→20070725
Firefox/2.0.0.6"
```

To direct TransLogger to an `access.log` `FileHandler`, we need the following to add a `FileHandler` (named `accesslog`) to the list of handlers, and ensure that the `wsgi` logger is configured and uses this handler accordingly:

```
# Begin logging configuration

[loggers]
keys = root, myapp, wsgi

[handlers]
keys = console, accesslog

[logger_wsgi]
level = INFO
handlers = accesslog
qualname = wsgi
propagate = 0

[handler_accesslog]
class = FileHandler
args = ('%(here)s/access.log', 'a')
level = INFO
formatter = generic
```

As mentioned above, non-root loggers by default propagate their log records to the root logger's handlers (currently the console handler). Setting `propagate` to 0 (False) here disables this; so the `wsgi` logger directs its records only to the `accesslog` handler.

Finally, there's no need to use the generic formatter with TransLogger as TransLogger itself provides all the information we need. We'll use a formatter that passes through the log messages as is. Add a new formatter called `accesslog` by including the following in your configuration file:

```
[formatters]
keys = generic, accesslog
```

```
[formatter_accesslog]
format = %(message)s
```

Finally alter the existing configuration to wire this new `accesslog` formatter into the `FileHandler`:

```
[handler_accesslog]
class = FileHandler
args = ('%(here)s/access.log', 'a')
level = INFO
formatter = accesslog
```

PasteDeploy Configuration Files

Packages generated via a *scaffold* make use of a system created by Ian Bicking named *PasteDeploy*. PasteDeploy defines a way to declare *WSGI* application configuration in an `.ini` file.

Pyramid uses this configuration file format as input to its *WSGI* server runner `pserve`, as well as other commands such as `pviews`, `pshell`, `proutes`, and `ptweens`.

PasteDeploy is not a particularly integral part of Pyramid. It's possible to create a Pyramid application which does not use PasteDeploy at all. We show a Pyramid application that doesn't use PasteDeploy in *Creating Your First Pyramid Application*. However, all Pyramid scaffolds render PasteDeploy configuration files, to provide new developers with a standardized way of setting deployment values, and to provide new users with a standardized way of starting, stopping, and debugging an application.

This chapter is not a replacement for documentation about PasteDeploy; it only contextualizes the use of PasteDeploy within Pyramid. For detailed documentation, see <http://pythonpaste.org/deploy/>.

PasteDeploy

PasteDeploy is the system that Pyramid uses to allow *deployment settings* to be specified using an `.ini` configuration file format. It also allows the `pserve` command to work. Its configuration format provides a convenient place to define application *deployment settings* and *WSGI* server settings, and its server runner allows you to stop and start a Pyramid application easily.

Entry Points and PasteDeploy .ini Files

In the *Creating a Pyramid Project* chapter, we breezed over the meaning of a configuration line in the `deployment.ini` file. This was the `use = egg:MyProject` line in the `[app:main]` section. We breezed over it because it's pretty confusing and "too much information" for an introduction to the system. We'll try to give it a bit of attention here. Let's see the config file again:

```
1  ###
2  # app configuration
3  # http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
   ↪environment.html
4  ###
5
6  [app:main]
7  use = egg:MyProject
8
9  pyramid.reload_templates = true
10 pyramid.debug_authorization = false
11 pyramid.debug_notfound = false
12 pyramid.debug_routematch = false
13 pyramid.default_locale_name = en
14 pyramid.includes =
15     pyramid_debugtoolbar
16
17 # By default, the toolbar only appears for clients from IP addresses
18 # '127.0.0.1' and '::1'.
19 # debugtoolbar.hosts = 127.0.0.1 ::1
20
21 ###
22 # wsgi server configuration
23 ###
24
25 [server:main]
26 use = egg:waitress#main
27 host = 127.0.0.1
28 port = 6543
29
30 ###
31 # logging configuration
32 # http://docs.pylonsproject.org/projects/pyramid/en/1.7-branch/narr/
   ↪logging.html
33 ###
34
35 [loggers]
36 keys = root, myproject
37
```

```

38 [handlers]
39 keys = console
40
41 [formatters]
42 keys = generic
43
44 [logger_root]
45 level = INFO
46 handlers = console
47
48 [logger_myproject]
49 level = DEBUG
50 handlers =
51 qualname = myproject
52
53 [handler_console]
54 class = StreamHandler
55 args = (sys.stderr,)
56 level = NOTSET
57 formatter = generic
58
59 [formatter_generic]
60 format = %(asctime)s %(levelname)-5.5s [%(name)s:%(lineno)s] [
    ↪ %(threadName)s] %(message)s

```

The line in `[app:main]` above that says `use = egg:MyProject` is actually shorthand for a longer spelling: `use = egg:MyProject#main`. The `#main` part is omitted for brevity, as `#main` is a default defined by PasteDeploy. `egg:MyProject#main` is a string which has meaning to PasteDeploy. It points at a *setuptools* entry point named `main` defined in the `MyProject` project.

Take a look at the generated `setup.py` file for this project.

```

1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 with open(os.path.join(here, 'README.txt')) as f:
7     README = f.read()
8 with open(os.path.join(here, 'CHANGES.txt')) as f:
9     CHANGES = f.read()
10
11 requires = [
12     'pyramid',
13     'pyramid_chameleon',

```

```
14     'pyramid_debugtoolbar',
15     'waitress',
16 ]
17
18 tests_require = [
19     'WebTest >= 1.3.1', # py3 compat
20     'pytest', # includes virtualenv
21     'pytest-cov',
22 ]
23
24 setup(name='MyProject',
25       version='0.0',
26       description='MyProject',
27       long_description=README + '\n\n' + CHANGES,
28       classifiers=[
29         "Programming Language :: Python",
30         "Framework :: Pyramid",
31         "Topic :: Internet :: WWW/HTTP",
32         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
33     ],
34     author='',
35     author_email='',
36     url='',
37     keywords='web pyramid pylons',
38     packages=find_packages(),
39     include_package_data=True,
40     zip_safe=False,
41     extras_require={
42         'testing': tests_require,
43     },
44     install_requires=requires,
45     entry_points="""\
46 [paste.app_factory]
47 main = myproject:main
48 """,
49 )
```

Note that `entry_points` is assigned a string which looks a lot like an `.ini` file. This string representation of an `.ini` file has a section named `[paste.app_factory]`. Within this section, there is a key named `main` (the entry point name) which has a value `myproject:main`. The key `main` is what our `egg:MyProject#main` value of the `use` section in our config file is pointing at, although it is actually shortened to `egg:MyProject` there. The value represents a *dotted Python name* path, which refers to a callable in our `myproject` package's `__init__.py` module.

The `egg:` prefix in `egg:MyProject` indicates that this is an entry point *URI* specifier, where the

“scheme” is “egg”. An “egg” is created when you run `setup.py install` or `setup.py develop` within your project.

In English, this entry point can thus be referred to as a “PasteDeploy application factory in the `MyProject` project which has the entry point named `main` where the entry point refers to a `main` function in the `mypackage` module”. Indeed, if you open up the `__init__.py` module generated within any scaffold-generated package, you’ll see a `main` function. This is the function called by *PasteDeploy* when the `pserve` command is invoked against our application. It accepts a global configuration object and *returns* an instance of our application.

[DEFAULT] Section of a PasteDeploy .ini File

You can add a `[DEFAULT]` section to your PasteDeploy `.ini` file. Such a section should consist of global parameters that are shared by all the applications, servers, and *middleware* defined within the configuration file. The values in a `[DEFAULT]` section will be passed to your application’s `main` function as `global_config` (see the reference to the `main` function in `__init__.py`).

Command-Line Pyramid

Your Pyramid application can be controlled and inspected using a variety of command-line utilities. These utilities are documented in this chapter.

Displaying Matching Views for a Given URL

See also:

See also the output of `pviews -help`.

For a big application with several views, it can be hard to keep the view configuration details in your head, even if you defined all the views yourself. You can use the `pviews` command in a terminal window to print a summary of matching routes and views for a given URL in your application. The `pviews` command accepts two arguments. The first argument to `pviews` is the path to your application’s `.ini` file and section name inside the `.ini` file which points to your application. This should be of the format `config_file#section_name`. The second argument is the URL to test for matching views. The `section_name` may be omitted; if it is, it’s considered to be `main`.

Here is an example for a simple view configuration using *traversal*:

```

1 $ $VENV/bin/pviews development.ini#tutorial /FrontPage
2
3 URL = /FrontPage
4
5     context: <tutorial.models.Page object at 0xa12536c>
6     view name:
7
8     View:
9     -----
10    tutorial.views.view_page
11    required permission = view

```

The output always has the requested URL at the top and below that all the views that matched with their view configuration details. In this example only one view matches, so there is just a single *View* section. For each matching view, the full code path to the associated view callable is shown, along with any permissions and predicates that are part of that view configuration.

A more complex configuration might generate something like this:

```

1 $ $VENV/bin/pviews development.ini#shootout /about
2
3 URL = /about
4
5     context: <shootout.models.RootFactory object at 0xa56668c>
6     view name: about
7
8     Route:
9     -----
10    route name: about
11    route pattern: /about
12    route path: /about
13    subpath:
14    route predicates (request method = GET)
15
16    View:
17    -----
18    shootout.views.about_view
19    required permission = view
20    view predicates (request_param testing, header X/header)
21
22    Route:
23    -----
24    route name: about_post
25    route pattern: /about
26    route path: /about

```

```

27     subpath:
28         route predicates (request method = POST)
29
30         View:
31         -----
32         shootout.views.about_view_post
33         required permission = view
34         view predicates (request_param test)
35
36         View:
37         -----
38         shootout.views.about_view_post2
39         required permission = view
40         view predicates (request_param test2)

```

In this case, we are dealing with a *URL dispatch* application. This specific URL has two matching routes. The matching route information is displayed first, followed by any views that are associated with that route. As you can see from the second matching route output, a route can be associated with more than one view.

For a URL that doesn't match any views, `pviews` will simply print out a *Not found* message.

The Interactive Shell

See also:

See also the output of `pshell -help`.

Once you've installed your program for development using `pip install -e .`, you can use an interactive Python shell to execute expressions in a Python environment exactly like the one that will be used when your application runs "for real". To do so, use the `pshell` command line utility.

The argument to `pshell` follows the format `config_file#section_name` where `config_file` is the path to your application's `.ini` file and `section_name` is the app section name inside the `.ini` file which points to your application. For example, your application `.ini` file might have an `[app:main]` section that looks like so:

```

1  [app:main]
2  use = egg:MyProject
3  pyramid.reload_templates = true
4  pyramid.debug_authorization = false
5  pyramid.debug_notfound = false
6  pyramid.debug_templates = true
7  pyramid.default_locale_name = en

```

If so, you can use the following command to invoke a debug shell using the name `main` as a section name:

```
$ $VENV/bin/pshell starter/development.ini#main
Python 2.6.5 (r265:79063, Apr 29 2010, 00:31:32)
[GCC 4.4.3] on linux2
Type "help" for more information.

Environment:
  app           The WSGI application.
  registry      Active Pyramid registry.
  request       Active request object.
  root          Root of the default resource tree.
  root_factory  Default root factory used to create `root`.

>>> root
<myproject.resources.MyResource object at 0x445270>
>>> registry
<Registry myproject>
>>> registry.settings['pyramid.debug_notfound']
False
>>> from myproject.views import my_view
>>> from pyramid.request import Request
>>> r = Request.blank('/')
>>> my_view(r)
{'project': 'myproject'}
```

The WSGI application that is loaded will be available in the shell as the `app` global. Also, if the application that is loaded is the Pyramid app with no surrounding *middleware*, the `root` object returned by the default *root factory*, `registry`, and `request` will be available.

You can also simply rely on the `main` default section name by omitting any hash after the filename:

```
$ $VENV/bin/pshell starter/development.ini
```

Press `Ctrl-D` to exit the interactive shell (or `Ctrl-Z` on Windows).

Extending the Shell

It is convenient when using the interactive shell often to have some variables significant to your application already loaded as globals when you start the `pshell`. To facilitate this, `pshell` will look for a special `[pshell]` section in your INI file and expose the subsequent key/value pairs to the shell. Each key

is a variable name that will be global within the pshell session; each value is a *dotted Python name*. If specified, the special key `setup` should be a *dotted Python name* pointing to a callable that accepts the dictionary of globals that will be loaded into the shell. This allows for some custom initializing code to be executed each time the pshell is run. The `setup` callable can also be specified from the commandline using the `--setup` option which will override the key in the INI file.

For example, you want to expose your model to the shell along with the database session so that you can mutate the model on an actual database. Here, we'll assume your model is stored in the `myapp.models` package.

```
1 [pshell]
2 setup = myapp.lib.pshell.setup
3 m = myapp.models
4 session = myapp.models.DBSession
5 t = transaction
```

By defining the `setup` callable, we will create the module `myapp.lib.pshell` containing a callable named `setup` that will receive the global environment before it is exposed to the shell. Here we mutate the environment's request as well as add a new value containing a WebTest version of the application to which we can easily submit requests.

```
1 # myapp/lib/pshell.py
2 from webtest import TestApp
3
4 def setup(env):
5     env['request'].host = 'www.example.com'
6     env['request'].scheme = 'https'
7     env['testapp'] = TestApp(env['app'])
```

When this INI file is loaded, the extra variables `m`, `session` and `t` will be available for use immediately. Since a `setup` callable was also specified, it is executed and a new variable `testapp` is exposed, and the request is configured to generate urls from the host `http://www.example.com`. For example:

```
$ $VENV/bin/pshell starter/development.ini
Python 2.6.5 (r265:79063, Apr 29 2010, 00:31:32)
[GCC 4.4.3] on linux2
Type "help" for more information.

Environment:
  app           The WSGI application.
  registry      Active Pyramid registry.
  request       Active request object.
  root          Root of the default resource tree.
```



```
root_factory Default root factory used to create `root`.
testapp      <webtest.TestApp object at ...>

Custom Variables:
  m          myapp.models
  session    myapp.models.DBSession
  t          transaction

>>> testapp.get('/')
<200 OK text/html body='<!DOCTYPE...l>\n'/3337>
>>> request.route_url('home')
'https://www.example.com/'
```

Alternative Shells

The `pshell` command can be easily extended with alternate REPLs if the default python REPL is not satisfactory. Assuming you have a binding installed such as `pyramid_ipython` it will normally be auto-selected and used. You may also specifically invoke your choice with the `-p` choice or `--python-shell` choice option.

```
$ $VENV/bin/pshell -p ipython development.ini#MyProject
```

You may use the `--list-shells` option to see the available shells.

```
$ $VENV/bin/pshell --list-shells
Available shells:
  bpython
  ipython
  python
```

If you want to use a shell that isn't supported out of the box, you can introduce a new shell by registering an entry point in your `setup.py`:

```
setup(
    entry_points={
        'pyramid.pshell_runner': [
            'myshell=my_app:ptpython_shell_factory',
        ],
    },
)
```

And then your shell factory should return a function that accepts two arguments, `env` and `help`, which would look like this:

```
from ptpython.repl import embed

def ptpython_shell_runner(env, help):
    print(help)
    return embed(locals=env)
```

Changed in version 1.6: User-defined shells may be registered using entry points. Prior to this the only supported shells were `ipython`, `bpython` and `python`.

`ipython` and `bpython` have been moved into their respective packages `pyramid_ipython` and `pyramid_bpython`.

Setting a Default Shell

You may use the `default_shell` option in your `[pshell]` ini section to specify a list of preferred shells.

```
1 [pshell]
2 default_shell = ptpython ipython bpython
```

New in version 1.6.

Displaying All Application Routes

See also:

See also the output of `proutes -help`.

You can use the `proutes` command in a terminal window to print a summary of routes related to your application. Much like the `pshell` command (see *The Interactive Shell*), the `proutes` command accepts one argument with the format `config_file#section_name`. The `config_file` is the path to your application's `.ini` file, and `section_name` is the app section name inside the `.ini` file which points to your application. By default, the `section_name` is `main` and can be omitted.

For example:

```

1 $ $VENV/bin/proutes development.ini
2 Name                Pattern                View
   ↳                  Method
3 ----                -
   ↳                  -
4 debugtoolbar         /_debug_toolbar/*subpath          <wsgiapp>
   ↳                  *
5 __static/            /static/*subpath                  dummy_
   ↳starter:static/          *
6 __static2/           /static2/*subpath                 /var/www/static/
   ↳                  *
7 __pdt_images/        /pdt_images/*subpath             pyramid_
   ↳debugtoolbar:static/img/  *
8 a                    /                                  <unknown>
   ↳                  *
9 no_view_attached     /                                  <unknown>
   ↳                  *
10 route_and_view_attached /                                  appl.standard_views.
   ↳route_and_view_attached  *
11 method_conflicts    /conflicts                        appl.standard_
   ↳conflicts                <route mismatch>
12 multiview            /multiview                        appl.standard_views.
   ↳multiview                GET,PATCH
13 not_post             /not_post                          appl.standard_views.
   ↳multiview                !POST,*

```

`proutes` generates a table with four columns: *Name*, *Pattern*, *View*, and *Method*. The items listed in the Name column are route names, the items listed in the Pattern column are route patterns, the items listed in the View column are representations of the view callable that will be invoked when a request matches the associated route pattern, and the items listed in the Method column are the request methods that are associated with the route name. The View column may show `<unknown>` if no associated view callable could be found. The Method column, for the route name, may show either `<route mismatch>` if the view callable does not accept any of the route's request methods, or `*` if the view callable will accept any of the route's request methods. If no routes are configured within your application, nothing will be printed to the console when `proutes` is executed.

It is convenient when using the `proutes` command often to configure which columns and the order you would like to view them. To facilitate this, `proutes` will look for a special `[proutes]` section in your `.ini` file and use those as defaults.

For example you may remove the request method and place the view first:

```

1 [proutes]
2 format = view

```

```

3         name
4         pattern

```

You can also separate the formats with commas or spaces:

```

1     [proutes]
2     format = view name pattern
3
4     [proutes]
5     format = view, name, pattern

```

If you want to temporarily configure the columns and order, there is the argument `--format`, which is a comma separated list of columns you want to include. The current available formats are `name`, `pattern`, `view`, and `method`.

Displaying “Tweens”

See also:

See also the output of `ptweens --help`.

A *tween* is a bit of code that sits between the main Pyramid application request handler and the WSGI application which calls it. A user can get a representation of both the implicit tween ordering (the ordering specified by calls to `pyramid.config.Configurator.add_tween()`) and the explicit tween ordering (specified by the `pyramid.tweens` configuration setting) using the `ptweens` command. Tween factories will show up represented by their standard Python dotted name in the `ptweens` output.

For example, here’s the `ptweens` command run against a system configured without any explicit tweens:

```

1 $ $VENV/bin/ptweens development.ini
2 "pyramid.tweens" config value NOT set (implicitly ordered tweens used)
3
4 Implicit Tween Chain
5
6 Position      Name                                          Alias
7 -----
8 -
9 0             pyramid_debugtoolbar.toolbar.toolbar_tween_factory  pdbt
10 1             pyramid.tweens.excview_tween_factory          excview
11 -                                                     MAIN

```

Here’s the `ptweens` command run against a system configured *with* explicit tweens defined in its `development.ini` file:

```

1 $ ptweens development.ini
2 "pyramid.tweens" config value set (explicitly ordered tweens used)
3
4 Explicit Tween Chain (used)
5
6 Position      Name
7 -----
8 -             INGRESS
9 0             starter.tween_factory2
10 1            starter.tween_factory1
11 2            pyramid.tweens.excview_tween_factory
12 -            MAIN
13
14 Implicit Tween Chain (not used)
15
16 Position      Name
17 -----
18 -             INGRESS
19 0            pyramid_debugtoolbar.toolbar.toolbar_tween_factory
20 1            pyramid.tweens.excview_tween_factory
21 -            MAIN

```

Here's the application configuration section of the `development.ini` used by the above `ptweens` command which reports that the explicit tween chain is used:

```

1 [app:main]
2 use = egg:starter
3 reload_templates = true
4 debug_authorization = false
5 debug_notfound = false
6 debug_routematch = false
7 debug_templates = true
8 default_locale_name = en
9 pyramid.include = pyramid_debugtoolbar
10 pyramid.tweens = starter.tween_factory2
11                  starter.tween_factory1
12                  pyramid.tweens.excview_tween_factory

```

See *Registering Tweens* for more information about tweens.

Invoking a Request

See also:

See also the output of *prequest --help*.

You can use the *prequest* command-line utility to send a request to your application and see the response body without starting a server.

There are two required arguments to *prequest*:

- The config file/section: follows the format `config_file#section_name`, where `config_file` is the path to your application's `.ini` file and `section_name` is the app section name inside the `.ini` file. The `section_name` is optional; it defaults to `main`. For example: `development.ini`.
- The path: this should be the non-URL-quoted path element of the URL to the resource you'd like to be rendered on the server. For example, `/`.

For example:

```
$ $VENV/bin/prequest development.ini /
```

This will print the body of the response to the console on which it was invoked.

Several options are supported by *prequest*. These should precede any config file name or URL.

prequest has a `-d` (i.e., `--display-headers`) option which prints the status and headers returned by the server before the output:

```
$ $VENV/bin/prequest -d development.ini /
```

This will print the status, headers, and the body of the response to the console.

You can add request header values by using the `--header` option:

```
$ $VENV/bin/prequest --header=Host:example.com development.ini /
```

Headers are added to the WSGI environment by converting them to their CGI/WSGI equivalents (e.g., `Host=example.com` will insert the `HTTP_HOST` header variable as the value `example.com`). Multiple `--header` options can be supplied. The special header value `content-type` sets the `CONTENT_TYPE` in the WSGI environment.

By default, *prequest* sends a GET request. You can change this by using the `-m` (aka `--method`) option. GET, HEAD, POST, and DELETE are currently supported. When you use POST, the standard input of the *prequest* process is used as the POST body:

```
$ $VENV/bin/prequest -mPOST development.ini / < somefile
```

Using Custom Arguments to Python when Running p* Scripts

New in version 1.5.

Each of Pyramid's console scripts (`pserve`, `pviews`, etc.) can be run directly using `python3 -m`, allowing custom arguments to be sent to the Python interpreter at runtime. For example:

```
python3 -m pyramid.scripts.pserve development.ini
```

Showing All Installed Distributions and Their Versions

New in version 1.5.

See also:

See also the output of `pdistreport --help`.

You can use the `pdistreport` command to show the Pyramid version in use, the Python version in use, and all installed versions of Python distributions in your Python environment:

```
$ $VENV/bin/pdistreport
Pyramid version: 1.5dev
Platform Linux-3.2.0-51-generic-x86_64-with-debian-wheezy-sid
Packages:
  authapp 0.0
    /home/chris/projects/foo/src/authapp
  beautifulsoup4 4.1.3
    /home/chris/projects/foo/lib/python2.7/site-packages/beautifulsoup4-4.
    ↪1.3-py2.7.egg
  ... more output ...
```

`pdistreport` takes no options. Its output is useful to paste into a pastebin when you are having problems and need someone with more familiarity with Python packaging and distribution than you have to look at your environment.

Writing a Script

All web applications are, at their hearts, systems which accept a request and return a response. When a request is accepted by a Pyramid application, the system receives state from the request which is later relied on by your application code. For example, one *view callable* may assume it's working against a request that has a `request.matchdict` of a particular composition, while another assumes a different composition of the `matchdict`.

In the meantime, it's convenient to be able to write a Python script that can work “in a Pyramid environment”, for instance to update database tables used by your Pyramid application. But a “real” Pyramid environment doesn't have a completely static state independent of a request; your application (and Pyramid itself) is almost always reliant on being able to obtain information from a request. When you run a Python script that simply imports code from your application and tries to run it, there just is no request data, because there isn't any real web request. Therefore some parts of your application and some Pyramid APIs will not work.

For this reason, Pyramid makes it possible to run a script in an environment much like the environment produced when a particular *request* reaches your Pyramid application. This is achieved by using the `pyramid.paster.bootstrap()` command in the body of your script.

New in version 1.1: `pyramid.paster.bootstrap()`

In the simplest case, `pyramid.paster.bootstrap()` can be used with a single argument, which accepts the *PasteDeploy* `.ini` file representing your Pyramid application's configuration as a single argument:

```
from pyramid.paster import bootstrap
env = bootstrap('/path/to/my/development.ini')
print(env['request'].route_url('home'))
```

`pyramid.paster.bootstrap()` returns a dictionary containing framework-related information. This dictionary will always contain a *request* object as its `request` key.

The following keys are available in the `env` dictionary returned by `pyramid.paster.bootstrap()`:

request

A `pyramid.request.Request` object implying the current request state for your script.

app

The *WSGI* application object generated by bootstrapping.

root

The *resource* root of your Pyramid application. This is an object generated by the *root factory* configured in your application.

registry

The *application registry* of your Pyramid application.

closer

A parameterless callable that can be used to pop an internal Pyramid threadlocal stack (used by `pyramid.threadlocal.get_current_registry()` and `pyramid.threadlocal.get_current_request()`) when your scripting job is finished.

Let's assume that the `/path/to/my/development.ini` file used in the example above looks like so:

```
[pipeline:main]
pipeline = translogger
          another

[filter:translogger]
filter_app_factory = egg:Paste#translogger
setup_console_handler = False
logger_name = wsgi

[app:another]
use = egg:MyProject
```

The configuration loaded by the above bootstrap example will use the configuration implied by the `[pipeline:main]` section of your configuration file by default. Specifying `/path/to/my/development.ini` is logically equivalent to specifying `/path/to/my/development.ini#main`. In this case, we'll be using a configuration that includes an app object which is wrapped in the Paste “translogger” *middleware* (which logs requests to the console).

You can also specify a particular *section* of the PasteDeploy `.ini` file to load instead of `main`:

```
from pyramid.paster import bootstrap
env = bootstrap('/path/to/my/development.ini#another')
print(env['request'].route_url('home'))
```

The above example specifies the `another` app, pipeline, or composite section of your PasteDeploy configuration file. The app object present in the `env` dictionary returned by `pyramid.paster.bootstrap()` will be a Pyramid *router*.

Changing the Request

By default, Pyramid will generate a request object in the `env` dictionary for the URL `http://localhost:80/`. This means that any URLs generated by Pyramid during the execution of your script will be anchored here. This is generally not what you want.

So how do we make Pyramid generate the correct URLs?

Assuming that you have a route configured in your application like so:

```
config.add_route('verify', '/verify/{code}')
```

You need to inform the Pyramid environment that the WSGI application is handling requests from a certain base. For example, we want to simulate mounting our application at `https://example.com/prefix`, to ensure that the generated URLs are correct for our deployment. This can be done by either mutating the resulting request object, or more simply by constructing the desired request and passing it into `bootstrap()`:

```
from pyramid.paster import bootstrap
from pyramid.request import Request

request = Request.blank('/', base_url='https://example.com/prefix')
env = bootstrap('/path/to/my/development.ini#another', request=request)
print(env['request'].application_url)
# will print 'https://example.com/prefix'
```

Now you can readily use Pyramid's APIs for generating URLs:

```
env['request'].route_url('verify', code='1337')
# will return 'https://example.com/prefix/verify/1337'
```

Cleanup

When your scripting logic finishes, it's good manners to call the `closer` callback:

```
from pyramid.paster import bootstrap
env = bootstrap('/path/to/my/development.ini')

# .. do stuff ...

env['closer']()
```

Setting Up Logging

By default, `pyramid.paster.bootstrap()` does not configure logging parameters present in the configuration file. If you'd like to configure logging based on `[logger]` and related sections in the configuration file, use the following command:

```
import pyramid.paster
pyramid.paster.setup_logging('/path/to/my/development.ini')
```

See *Logging* for more information on logging within Pyramid.

Making Your Script into a Console Script

A “console script” is *setuptools* terminology for a script that gets installed into the `bin` directory of a Python *virtual environment* (or “base” Python environment) when a *distribution* which houses that script is installed. Because it's installed into the `bin` directory of a virtual environment when the distribution is installed, it's a convenient way to package and distribute functionality that you can call from the command-line. It's often more convenient to create a console script than it is to create a `.py` script and instruct people to call it with the “right” Python interpreter. A console script generates a file that lives in `bin`, and when it's invoked it will always use the “right” Python environment, which means it will always be invoked in an environment where all the libraries it needs (such as Pyramid) are available.

In general, you can make your script into a console script by doing the following:

- Use an existing distribution (such as one you've already created via `pcreate`) or create a new distribution that possesses at least one package or module. It should, within any module within the distribution, house a callable (usually a function) that takes no arguments and which runs any of the code you wish to run.
- Add a `[console_scripts]` section to the `entry_points` argument of the distribution which creates a mapping between a script name and a dotted name representing the callable you added to your distribution.
- Run `pip install -e .` or `pip install .` to get your distribution reinstalled. When you reinstall your distribution, a file representing the script that you named in the last step will be in the `bin` directory of the virtual environment in which you installed the distribution. It will be executable. Invoking it from a terminal will execute your callable.

As an example, let's create some code that can be invoked by a console script that prints the deployment settings of a Pyramid application. To do so, we'll pretend you have a distribution with a package in it named `myproject`. Within this package, we'll pretend you've added a `scripts.py` module which contains the following code:

```

1  # myproject.scripts module
2
3  import optparse
4  import sys
5  import textwrap
6
7  from pyramid.paster import bootstrap
8
9  def settings_show():
10     description = """\
11     Print the deployment settings for a Pyramid application.  Example:
12     'show_settings deployment.ini'
13     """
14     usage = "usage: %prog config_uri"
15     parser = optparse.OptionParser(
16         usage=usage,
17         description=textwrap.dedent(description)
18     )
19     parser.add_option(
20         '-o', '--omit',
21         dest='omit',
22         metavar='PREFIX',
23         type='string',
24         action='append',
25         help=("Omit settings which start with PREFIX (you can use this "
26              "option multiple times)")
27     )
28
29     options, args = parser.parse_args(sys.argv[1:])
30     if not len(args) >= 1:
31         print('You must provide at least one argument')
32         return 2
33     config_uri = args[0]
34     omit = options.omit
35     if omit is None:
36         omit = []
37     env = bootstrap(config_uri)
38     settings, closer = env['registry'].settings, env['closer']
39     try:
40         for k, v in settings.items():
41             if any([k.startswith(x) for x in omit]):
42                 continue
43             print('%-40s    %-20s' % (k, v))
44     finally:
45         closer()

```

This script uses the Python `optparse` module to allow us to make sense out of extra arguments passed to the script. It uses the `pyramid.paster.bootstrap()` function to get information about the application defined by a config file, and prints the deployment settings defined in that config file.

After adding this script to the package, you'll need to tell your distribution's `setup.py` about its existence. Within your distribution's top-level directory, your `setup.py` file will look something like this:

```
1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 with open(os.path.join(here, 'README.txt')) as f:
7     README = f.read()
8 with open(os.path.join(here, 'CHANGES.txt')) as f:
9     CHANGES = f.read()
10
11 requires = ['pyramid', 'pyramid_debugtoolbar']
12
13 tests_require = [
14     'WebTest >= 1.3.1', # py3 compat
15     'pytest', # includes virtualenv
16     'pytest-cov',
17 ]
18
19 setup(name='MyProject',
20       version='0.0',
21       description='My project',
22       long_description=README + '\n\n' + CHANGES,
23       classifiers=[
24         "Programming Language :: Python",
25         "Framework :: Pyramid",
26         "Topic :: Internet :: WWW/HTTP",
27         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
28     ],
29     author='',
30     author_email='',
31     url='',
32     keywords='web pyramid pylons',
33     packages=find_packages(),
34     include_package_data=True,
35     zip_safe=False,
36     install_requires=requires,
37     extras_require={
38         'testing': tests_require,
39     },
40     entry_points = """\
```

```

41     [paste.app_factory]
42     main = myproject:main
43     """
44 )

```

We're going to change the `setup.py` file to add a `[console_scripts]` section within the `entry_points` string. Within this section, you should specify a `scriptname = dotted.path.to:yourfunction` line. For example:

```

[console_scripts]
show_settings = myproject.scripts:settings_show

```

The `show_settings` name will be the name of the script that is installed into `bin`. The colon (`:`) between `myproject.scripts` and `settings_show` above indicates that `myproject.scripts` is a Python module, and `settings_show` is the function in that module which contains the code you'd like to run as the result of someone invoking the `show_settings` script from their command line.

The result will be something like:

```

1  import os
2
3  from setuptools import setup, find_packages
4
5  here = os.path.abspath(os.path.dirname(__file__))
6  with open(os.path.join(here, 'README.txt')) as f:
7      README = f.read()
8  with open(os.path.join(here, 'CHANGES.txt')) as f:
9      CHANGES = f.read()
10
11  requires = ['pyramid', 'pyramid_debugtoolbar']
12
13  tests_require = [
14      'WebTest >= 1.3.1', # py3 compat
15      'pytest', # includes virtualenv
16      'pytest-cov',
17  ]
18
19  setup(name='MyProject',
20        version='0.0',
21        description='My project',
22        long_description=README + '\n\n' + CHANGES,
23        classifiers=[
24            "Programming Language :: Python",
25            "Framework :: Pyramid",

```

```
26         "Topic :: Internet :: WWW/HTTP",
27         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
28     ],
29     author='',
30     author_email='',
31     url='',
32     keywords='web pyramid pylons',
33     packages=find_packages(),
34     include_package_data=True,
35     zip_safe=False,
36     install_requires=requires,
37     extras_require={
38         'testing': tests_require,
39     },
40     entry_points = """\
41     [paste.app_factory]
42     main = myproject:main
43     [console_scripts]
44     show_settings = myproject.scripts:settings_show
45     """,
46 )
```

Once you’ve done this, invoking `$VENV/bin/pip install -e .` will install a file named `show_settings` into the `$someenv/bin` directory with a small bit of Python code that points to your entry point. It will be executable. Running it without any arguments will print an error and exit. Running it with a single argument that is the path of a config file will print the settings. Running it with an `--omit=foo` argument will omit the settings that have keys that start with `foo`. Running it with two “omit” options (e.g., `--omit=foo --omit=bar`) will omit all settings that have keys that start with either `foo` or `bar`:

```
$ $VENV/bin/show_settings development.ini --omit=pyramid --
↳omit=debugtoolbar
debug_routematch                False
debug_templates                 True
reload_templates                True
mako.directories                []
debug_notfound                  False
default_locale_name             en
reload_resources                 False
debug_authorization             False
reload_assets                   False
prevent_http_cache              False
```

Pyramid’s `pserve`, `pcreate`, `pshell`, `prequest`, `ptweens`, and other `p*` scripts are implemented as console scripts. When you invoke one of those, you are using a console script.

Internationalization and Localization

Internationalization (i18n) is the act of creating software with a user interface that can potentially be displayed in more than one language or cultural context. *Localization* (l10n) is the process of displaying the user interface of an internationalized application in a *particular* language or cultural context.

Pyramid offers internationalization and localization subsystems that can be used to translate the text of buttons, error messages, and other software- and template-defined values into the native language of a user of your application.

Creating a Translation String


While you write your software, you can insert specialized markup into your Python code that makes it possible for the system to translate text values into the languages used by your application’s users. This markup creates a *translation string*. A translation string is an object that behaves mostly like a normal Unicode object, except that it also carries around extra information related to its job as part of the Pyramid translation machinery.

Using the `TranslationString` Class

The most primitive way to create a translation string is to use the `pyramid.i18n.TranslationString` callable:

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add')
```

This creates a Unicode-like object that is a `TranslationString`.

 For people more familiar with *Zope* i18n, a `TranslationString` is a lot like a `zope.i18nmessageid.Message` object. It is not a subclass, however. For people more familiar with *Pylons* or *Django* i18n, using a `TranslationString` is a lot like using “lazy” versions of related gettext APIs.

The first argument to `TranslationString` is the `msgid`; it is required. It represents the key into the translation mappings provided by a particular localization. The `msgid` argument must be a Unicode object or an ASCII string. The `msgid` may optionally contain *replacement markers*. For instance:


```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add ${number}')
```

Within the string above, `${number}` is a replacement marker. It will be replaced by whatever is in the *mapping* for a translation string. The mapping may be supplied at the same time as the replacement marker itself:

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add ${number}', mapping={'number':1})
```

Any number of replacement markers can be present in the *msgid* value, any number of times. Only markers which can be replaced by the values in the *mapping* will be replaced at translation time. The others will not be interpolated and will be output literally.

A translation string should also usually carry a *domain*. The domain represents a translation category to disambiguate it from other translations of the same *msgid*, in case they conflict.

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add ${number}', mapping={'number':1},
3                       domain='form')
```

The above translation string named a domain of *form*. A *translator* function will often use the domain to locate the right translator file on the filesystem which contains translations for a given domain. In this case, if it were trying to translate our *msgid* to German, it might try to find a translation from a *gettext* file within a *translation directory* like this one:

```
locale/de/LC_MESSAGES/form.mo
```

In other words, it would want to take translations from the *form.mo* translation file in the German language.

Finally, the *TranslationString* constructor accepts a *default* argument. If a *default* argument is supplied, it replaces usages of the *msgid* as the *default value* for the translation string. When *default* is *None*, the *msgid* value passed to a *TranslationString* is used as an implicit message identifier. Message identifiers are matched with translations in translation files, so it is often useful to create translation strings with “opaque” message identifiers unrelated to their default text:


```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('add-number', default='Add ${number}',
3                       domain='form', mapping={'number':1})
```

When default text is used, Default text objects may contain replacement values.

Using the `TranslationStringFactory` Class

Another way to generate a translation string is to use the *TranslationStringFactory* object. This object is a *translation string factory*. Basically a translation string factory presets the domain value of any *translation string* generated by using it. For example:

```
1 from pyramid.i18n import TranslationStringFactory
2 _ = TranslationStringFactory('pyramid')
3 ts = _('add-number', default='Add ${number}', mapping={'number':1})
```

 We assigned the translation string factory to the name `_`. This is a convention which will be supported by translation file generation tools.


After assigning `_` to the result of a *TranslationStringFactory()*, the subsequent result of calling `_` will be a *TranslationString* instance. Even though a domain value was not passed to `_` (as would have been necessary if the *TranslationString* constructor were used instead of a translation string factory), the domain attribute of the resulting translation string will be `pyramid`. As a result, the previous code example is completely equivalent (except for spelling) to:

```
1 from pyramid.i18n import TranslationString as _
2 ts = _('add-number', default='Add ${number}', mapping={'number':1},
3       domain='pyramid')
```

You can set up your own translation string factory much like the one provided above by using the *TranslationStringFactory* class. For example, if you'd like to create a translation string factory which presets the domain value of generated translation strings to `form`, you'd do something like this:

```
1 from pyramid.i18n import TranslationStringFactory
2 _ = TranslationStringFactory('form')
3 ts = _('add-number', default='Add ${number}', mapping={'number':1})
```

Creating a unique domain for your application via a translation string factory is best practice. Using your own unique translation domain allows another person to reuse your application without needing to merge your translation files with their own. Instead they can just include your package's *translation directory* via the *pyramid.config.Configurator.add_translation_dirs()* method.

 For people familiar with Zope internationalization, a *TranslationStringFactory* is a lot like a *zope.i18nmessageid.MessageFactory* object. It is not a subclass, however.

Working with `gettext` Translation Files

The basis of Pyramid translation services is GNU *gettext*. Once your application source code files and templates are marked up with translation markers, you can work on translations by creating various kinds of `gettext` files.



The steps a developer must take to work with *gettext message catalog* files within a Pyramid application are very similar to the steps a *Pylons* developer must take to do the same. See the Pylons Internationalization and Localization documentation for more information.

GNU `gettext` uses three types of files in the translation framework, `.pot` files, `.po` files, and `.mo` files.

`.pot` (Portable Object Template) files

A `.pot` file is created by a program which searches through your project's source code and which picks out every *message identifier* passed to one of the `_()` functions (e.g., *translation string* constructions). The list of all message identifiers is placed into a `.pot` file, which serves as a template for creating `.po` files.

`.po` (Portable Object) files

The list of messages in a `.pot` file are translated by a human to a particular language; the result is saved as a `.po` file.

`.mo` (Machine Object) files

A `.po` file is turned into a machine-readable binary file, which is the `.mo` file. Compiling the translations to machine code makes the localized program start faster.

The tools for working with *gettext* translation files related to a Pyramid application are *Lingua* and *Gettext*. *Lingua* can scrape `i18n` references out of Python and Chameleon files and create the `.pot` file. *Gettext* includes `msgmerge` tool to update a `.po` file from an updated `.pot` file and `msgfmt` to compile `.po` files to `.mo` files.

Installing Lingua and Gettext

In order for the commands related to working with `gettext` translation files to work properly, you will need to have *Lingua* and *Gettext* installed into the same environment in which Pyramid is installed.

Installation on UNIX

Gettext is often already installed on UNIX systems. You can check if it is installed by testing if the `msgfmt` command is available. If it is not available you can install it through the packaging system from your OS; the package name is almost always `gettext`. For example on a Debian or Ubuntu system run this command:

```
$ sudo apt-get install gettext
```

Installing Lingua is done with the Python packaging tools. If the *virtual environment* into which you've installed your Pyramid application lives at the environment variable `$VENV`, you can install Lingua like so:

```
$ $VENV/bin/pip install lingua
```

Installation on Windows

There are several ways to install Gettext on Windows: it is included in the Cygwin collection, or you can use the installer from the GnuWin32, or compile it yourself. Make sure the installation path is added to your `$PATH`.

Installing Lingua is done with the Python packaging tools. If the *virtual environment* into which you've installed your Pyramid application lives at the environment variable `%VENV%`, you can install Lingua like so:

```
c:\> %VENV%\Scripts\pip install lingua
```

Extracting Messages from Code and Templates

Once Lingua is installed, you may extract a message catalog template from the code and *Chameleon* templates which reside in your Pyramid application. You run a `pot-create` command to extract the messages:

```
$ cd /file/path/to/myapplication_setup.py
$ mkdir -p myapplication/locale
$ $VENV/bin/pot-create -o myapplication/locale/myapplication.pot src
```

The message catalog `.pot` template will end up in `myapplication/locale/myapplication.pot`.

Initializing a Message Catalog File

Once you’ve extracted messages into a `.pot` file (see *Extracting Messages from Code and Templates*), to begin localizing the messages present in the `.pot` file, you need to generate at least one `.po` file. A `.po` file represents translations of a particular set of messages to a particular locale. Initialize a `.po` file for a specific locale from a pre-generated `.pot` template by using the `msginit` command from Gettext:

```
$ cd /file/path/to/myapplication_setup.py
$ cd myapplication/locale
$ mkdir -p es/LC_MESSAGES
$ msginit -l es -o es/LC_MESSAGES/myapplication.po
```

This will create a new message catalog `.po` file in `myapplication/locale/es/LC_MESSAGES/myapplication.po`.

Once the file is there, it can be worked on by a human translator. One tool which may help with this is Poedit.

Note that Pyramid itself ignores the existence of all `.po` files. For a running application to have translations available, a `.mo` file must exist. See *Compiling a Message Catalog File*.

Updating a Catalog File

If more translation strings are added to your application, or translation strings change, you will need to update existing `.po` files based on changes to the `.pot` file, so that the new and changed messages can also be translated or re-translated.

First, regenerate the `.pot` file as per *Extracting Messages from Code and Templates*. Then use the `msgmerge` command from Gettext.

```
$ cd /file/path/to/myapplication_setup.py
$ cd myapplication/locale
$ msgmerge --update es/LC_MESSAGES/myapplication.po myapplication.pot
```

Compiling a Message Catalog File

Finally, to prepare an application for performing actual runtime translations, compile `.po` files to `.mo` files using the `msgfmt` command from Gettext:


```
$ cd /file/path/to/myapplication_setup.py
$ msgfmt -o myapplication/locale/es/LC_MESSAGES/myapplication.mo \
    myapplication/locale/es/LC_MESSAGES/myapplication.po
```

This will create a `.mo` file for each `.po` file in your application. As long as the *translation directory* in which the `.mo` file ends up in is configured into your application (see *Adding a Translation Directory*), these translations will be available to Pyramid.

Using a Localizer

A *localizer* is an object that allows you to perform translation or pluralization “by hand” in an application. You may use the `pyramid.request.Request.localizer` attribute to obtain a *localizer*. The localizer object will be configured to produce translations implied by the active *locale negotiator*, or a default localizer object if no explicit locale negotiator is registered.

```
1 def aview(request):
2     localizer = request.localizer
```

 If you need to create a localizer for a locale, use the `pyramid.i18n.make_localizer()` function.

Performing a Translation

A *localizer* has a `translate` method which accepts either a *translation string* or a Unicode string and which returns a Unicode object representing the translation. Generating a translation in a view component of an application might look like so:

```
1 from pyramid.i18n import TranslationString
2
3 ts = TranslationString('Add ${number}', mapping={'number':1},
4                       domain='pyramid')
5
6 def aview(request):
7     localizer = request.localizer
8     translated = localizer.translate(ts) # translation string
9     # ... use translated ...
```

The `request.localizer` attribute will be a `pyramid.i18n.Localizer` object bound to the locale name represented by the request. The translation returned from its `pyramid.i18n.Localizer.translate()` method will depend on the `domain` attribute of the provided translation string as well as the locale of the localizer.



If you're using *Chameleon* templates, you don't need to pre-translate translation strings this way. See *Chameleon Template Support for Translation Strings*.

Performing a Pluralization

A *localizer* has a `pluralize` method with the following signature:

```
1 def pluralize(singular, plural, n, domain=None, mapping=None):
2     ...
```

The simplest case is the `singular` and `plural` arguments being passed as Unicode literals. This returns the appropriate literal according to the locale pluralization rules for the number `n`, and interpolates `mapping`.

```
1 def aview(request):
2     localizer = request.localizer
3     translated = localizer.pluralize('Item', 'Items', 1, 'mydomain')
4     # ... use translated ...
```

However, for support of other languages, the `singular` argument should be a Unicode value representing a *message identifier*. In this case the `plural` value is ignored. `domain` should be a *translation domain*, and `mapping` should be a dictionary that is used for *replacement value* interpolation of the translated string.

The value of `n` will be used to find the appropriate plural form for the current language, and `pluralize` will return a Unicode translation for the message id `singular`. The message file must have defined `singular` as a translation with plural forms.

The argument provided as `singular` may be a *translation string* object, but the `domain` and `mapping` information attached is ignored.

```

1 def aview(request):
2     localizer = request.localizer
3     num = 1
4     translated = localizer.pluralize('item_plural', '${number} items',
5                                     num, 'mydomain', mapping={'number':num})

```

The corresponding message catalog must have language plural definitions and plural alternatives set.

```

1 "Plural-Forms: nplurals=3; plural=n==0 ? 0 : n==1 ? 1 : 2;"
2
3 msgid "item_plural"
4 msgid_plural ""
5 msgstr[0] "No items"
6 msgstr[1] "${number} item"
7 msgstr[2] "${number} items"

```

More information on complex plurals can be found in the gettext documentation.

Obtaining the Locale Name for a Request

You can obtain the locale name related to a request by using the `pyramid.request.Request.locale_name()` attribute of the request.

```

1 def aview(request):
2     locale_name = request.locale_name

```

The locale name of a request is dynamically computed; it will be the locale name negotiated by the currently active *locale negotiator*, or the *default locale name* if the locale negotiator returns `None`. You can change the default locale name by changing the `pyramid.default_locale_name` setting. See *Default Locale Name*.

Once `locale_name()` is first run, the locale name is stored on the request object. Subsequent calls to `locale_name()` will return the stored locale name without invoking the *locale negotiator*. To avoid this caching, you can use the `pyramid.i18n.negotiate_locale_name()` function:

```

1 from pyramid.i18n import negotiate_locale_name
2
3 def aview(request):
4     locale_name = negotiate_locale_name(request)

```

You can also obtain the locale name related to a request using the `locale_name` attribute of a *localizer*.


```
1 def aview(request):
2     localizer = request.localizer
3     locale_name = localizer.locale_name
```

Obtaining the locale name as an attribute of a localizer is equivalent to obtaining a locale name by asking for the `locale_name()` attribute.

Performing Date Formatting and Currency Formatting

Pyramid does not itself perform date and currency formatting for different locales. However, *Babel* can help you do this via the `babel.core.Locale` class. The Babel documentation for this class provides minimal information about how to perform date and currency related locale operations. See *Installing Lingua and Gettext* for information about how to install Babel.

The `babel.core.Locale` class requires a *locale name* as an argument to its constructor. You can use Pyramid APIs to obtain the locale name for a request to pass to the `babel.core.Locale` constructor. See *Obtaining the Locale Name for a Request*. For example:

```
1 from babel.core import Locale
2
3 def aview(request):
4     locale_name = request.locale_name
5     locale = Locale(locale_name)
```

Chameleon Template Support for Translation Strings

When a *translation string* is used as the subject of textual rendering by a *Chameleon* template renderer, it will automatically be translated to the requesting user's language if a suitable translation exists. This is true of both the ZPT and text variants of the Chameleon template renderers.

For example, in a Chameleon ZPT template, the translation string represented by “some_translation_string” in each example below will go through translation before being rendered:

```
1 <span tal:content="some_translation_string"/>
```

```
1 <span tal:replace="some_translation_string"/>
```

```
1 <span>${some_translation_string}</span>
```

```
1 <a tal:attributes="href some_translation_string">Click here</a>
```

The features represented by attributes of the `i18n` namespace of Chameleon will also consult the Pyramid translations. See <http://chameleon.readthedocs.org/en/latest/reference.html#translation-i18n>.

i Unlike when Chameleon is used outside of Pyramid, when it is used *within* Pyramid, it does not support use of the `zope.i18n` translation framework. Applications which use Pyramid should use the features documented in this chapter rather than `zope.i18n`.

Third party Pyramid template renderers might not provide this support out of the box and may need special code to do an equivalent. For those, you can always use the more manual translation facility described in *Performing a Translation*.

Mako Pyramid i18n Support

There exists a recipe within the *Pyramid Community Cookbook* named Mako Internationalization which explains how to add idiomatic i18n support to *Mako* templates.

Jinja2 Pyramid i18n Support

The add-on `pyramid_jinja2` provides a scaffold with an example of how to use internationalization with Jinja2 in Pyramid. See the documentation sections *Internalization (i18n)* and *Paster Template I18N*.

Localization-Related Deployment Settings

A Pyramid application will have a `pyramid.default_locale_name` setting. This value represents the *default locale name* used when the *locale negotiator* returns `None`. Pass it to the *Configurator* constructor at startup time:

```
1 from pyramid.config import Configurator
2 config = Configurator(settings={'pyramid.default_locale_name': 'de'})
```

You may alternately supply a `pyramid.default_locale_name` via an application's `.ini` file:

```
1 [app:main]
2 use = egg:MyProject
3 pyramid.reload_templates = true
4 pyramid.debug_authorization = false
5 pyramid.debug_notfound = false
6 pyramid.default_locale_name = de
```

If this value is not supplied via the Configurator constructor or via a config file, it will default to `en`.

If this setting is supplied within the Pyramid application `.ini` file, it will be available as a settings key:

```
1 from pyramid.threadlocal import get_current_registry
2 settings = get_current_registry().settings
3 default_locale_name = settings['pyramid.default_locale_name']
```

“Detecting” Available Languages

Other systems provide an API that returns the set of “available languages” as indicated by the union of all languages in all translation directories on disk at the time of the call to the API.

It is by design that Pyramid doesn’t supply such an API. Instead the application itself is responsible for knowing the “available languages”. The rationale is this: any particular application deployment must always know which languages it should be translatable to anyway, regardless of which translation files are on disk.

Here’s why: it’s not a given that because translations exist in a particular language within the registered set of translation directories that this particular deployment wants to allow translation to that language. For example, some translations may exist but they may be incomplete or incorrect. Or there may be translations to a language but not for all translation domains.

Any nontrivial application deployment will always need to be able to selectively choose to allow only some languages even if that set of languages is smaller than all those detected within registered translation directories. The easiest way to allow for this is to make the application entirely responsible for knowing which languages are allowed to be translated to instead of relying on the framework to divine this information from translation directory file info.

You can set up a system to allow a deployer to select available languages based on convention by using the `pyramid.settings` mechanism.

Allow a deployer to modify your application’s `.ini` file:

```
1 [app:main]
2 use = egg:MyProject
3 # ...
4 available_languages = fr de en ru
```

Then as a part of the code of a custom *locale negotiator*:

```
1 from pyramid.settings import aslist
2
3 def my_locale_negotiator(request):
4     languages = aslist(request.registry.settings['available_languages'])
5     # ...
```

This is only a suggestion. You can create your own “available languages” configuration scheme as necessary.

Activating Translation

By default, a Pyramid application performs no translation. To turn translation on you must:

- add at least one *translation directory* to your application.
- ensure that your application sets the *locale name* correctly.

Adding a Translation Directory

gettext is the underlying machinery behind the Pyramid translation machinery. A translation directory is a directory organized to be useful to *gettext*. A translation directory usually includes a listing of language directories, each of which itself includes an `LC_MESSAGES` directory. Each `LC_MESSAGES` directory should contain one or more `.mo` files. Each `.mo` file represents a *message catalog*, which is used to provide translations to your application.

Adding a *translation directory* registers all of its constituent *message catalog* files within your Pyramid application to be available to use for translation services. This includes all of the `.mo` files found within all `LC_MESSAGES` directories within each locale directory in the translation directory.

You can add a translation directory imperatively by using the `pyramid.config.Configurator.add_translation_dirs()` during application startup. For example:


```
1 from pyramid.config import Configurator
2 config.add_translation_dirs('my.application:locale/',
3                             'another.application:locale/')
```

A message catalog in a translation directory added via `add_translation_dirs()` will be merged into translations from a message catalog added earlier if both translation directories contain translations for the same locale and *translation domain*.

Setting the Locale

When the *default locale negotiator* (see *The Default Locale Negotiator*) is in use, you can inform Pyramid of the current locale name by doing any of these things before any translations need to be performed:

- Set the `__LOCALE__` attribute of the request to a valid locale name (usually directly within view code), e.g., `request.__LOCALE__ = 'de'`.
- Ensure that a valid locale name value is in the `request.params` dictionary under the key named `__LOCALE__`. This is usually the result of passing a `__LOCALE__` value in the query string or in the body of a form post associated with a request. For example, visiting `http://my.application?__LOCALE__=de`.
- Ensure that a valid locale name value is in the `request.cookies` dictionary under the key named `__LOCALE__`. This is usually the result of setting a `__LOCALE__` cookie in a prior response, e.g., `response.set_cookie('__LOCALE__', 'de')`.

 If this locale negotiation scheme is inappropriate for a particular application, you can configure a custom *locale negotiator* function into that application as required. See *Using a Custom Locale Negotiator*.

Locale Negotiators

A *locale negotiator* informs the operation of a *localizer* by telling it what *locale name* is related to a particular request. A locale negotiator is a bit of code which accepts a request and which returns a *locale name*. It is consulted when `pyramid.i18n.Localizer.translate()` or `pyramid.i18n.Localizer.pluralize()` is invoked. It is also consulted when `locale_name()` is accessed or when `negotiate_locale_name()` is invoked.

The Default Locale Negotiator

Most applications can make use of the default locale negotiator, which requires no additional coding or configuration.

The default locale negotiator implementation named `default_locale_negotiator` uses the following set of steps to determine the locale name.

- First the negotiator looks for the `__LOCALE__` attribute of the request object (possibly set directly by view code or by a listener for an *event*).
- Then it looks for the `request.params['_LOCALE_']` value.
- Then it looks for the `request.cookies['_LOCALE_']` value.
- If no locale can be found via the request, it falls back to using the *default locale name* (see *Localization-Related Deployment Settings*).
- Finally if the default locale name is not explicitly set, it uses the locale name `en`.

Using a Custom Locale Negotiator

Locale negotiation is sometimes policy-laden and complex. If the (simple) default locale negotiation scheme described in *Activating Translation* is inappropriate for your application, you may create a special *locale negotiator*. Subsequently you may override the default locale negotiator by adding your newly created locale negotiator to your application's configuration.

A locale negotiator is simply a callable which accepts a request and returns a single *locale name* or `None` if no locale can be determined.

Here's an implementation of a simple locale negotiator:

```
1 def my_locale_negotiator(request):  
2     locale_name = request.params.get('my_locale')  
3     return locale_name
```

If a locale negotiator returns `None`, it signifies to Pyramid that the default application locale name should be used.

You may add your newly created locale negotiator to your application's configuration by passing an object which can act as the negotiator (or a *dotted Python name* referring to the object) as the `locale_negotiator` argument of the *Configurator* instance during application startup. For example:

```
1 from pyramid.config import Configurator
2 config = Configurator(locale_negotiator=my_locale_negotiator)
```

Alternatively, use the `pyramid.config.Configurator.set_locale_negotiator()` method.

For example:

```
1 from pyramid.config import Configurator
2 config = Configurator()
3 config.set_locale_negotiator(my_locale_negotiator)
```

Virtual Hosting

“Virtual hosting” is, loosely, the act of serving a Pyramid application or a portion of a Pyramid application under a URL space that it does not “naturally” inhabit.

Pyramid provides facilities for serving an application under a URL “prefix”, as well as serving a *portion* of a *traversal* based application under a root URL.

Hosting an Application Under a URL Prefix

Pyramid supports a common form of virtual hosting whereby you can host a Pyramid application as a “subset” of some other site (e.g., under `http://example.com/mypyramidapplication/` as opposed to under `http://example.com/`).


If you use a “pure Python” environment, this functionality can be provided by Paste’s `urlmap` “composite” WSGI application. Alternatively, you can use `mod_wsgi` to serve your application, which handles this virtual hosting translation for you “under the hood”.

If you use the `urlmap` composite application “in front” of a Pyramid application or if you use `mod_wsgi` to serve up a Pyramid application, nothing special needs to be done within the application for URLs to be generated that contain a prefix. `paste.urlmap` and `mod_wsgi` manipulate the *WSGI* environment in such a way that the `PATH_INFO` and `SCRIPT_NAME` variables are correct for some given prefix.

Here’s an example of a PasteDeploy configuration snippet that includes a `urlmap` composite.

```
1 [app:mypyramidapp]
2 use = egg:mypyramidapp
3
4 [composite:main]
5 use = egg:Paste#urlmap
6 /pyramidapp = mypyramidapp
```

This “roots” the Pyramid application at the prefix `/pyramidapp` and serves up the composite as the “main” application in the file.

 If you’re using an Apache server to proxy to a Paste urlmap composite, you may have to use the `ProxyPreserveHost` directive to pass the original `HTTP_HOST` header along to the application, so URLs get generated properly. As of this writing the urlmap composite does not seem to respect the `HTTP_X_FORWARDED_HOST` parameter, which will contain the original host header even if `HTTP_HOST` is incorrect.

If you use `mod_wsgi`, you do not need to use a `composite` application in your `.ini` file. The `WSGIScriptAlias` configuration setting in a `mod_wsgi` configuration does the work for you:

```
1 WSGIScriptAlias /pyramidapp /Users/chrism/projects/modwsgi/env/pyramid.wsgi
```

In the above configuration, we root a Pyramid application at `/pyramidapp` within the Apache configuration.

Virtual Root Support


Pyramid also supports “virtual roots”, which can be used in *traversal*-based (but not *URL dispatch*-based) applications.

Virtual root support is useful when you’d like to host some resource in a Pyramid resource tree as an application under a URL pathname that does not include the resource path itself. For example, you might want to serve the object at the traversal path `/cms` as an application reachable via `http://example.com/` (as opposed to `http://example.com/cms`).

To specify a virtual root, cause an environment variable to be inserted into the WSGI environ named `HTTP_X_VHM_ROOT` with a value that is the absolute pathname to the resource object in the resource tree that should behave as the “root” resource. As a result, the traversal machinery will respect this value during traversal (prepending it to the `PATH_INFO` before traversal starts), and the `pyramid.request.Request.resource_url()` API will generate the “correct” virtually-rooted URLs.

An example of an Apache `mod_proxy` configuration that will host the `/cms` subobject as `http://www.example.com/` using this facility is below:


```
1 NameVirtualHost *:80
2
3 <VirtualHost *:80>
4     ServerName www.example.com
5     RewriteEngine On
6     RewriteRule ^/(.*) http://127.0.0.1:6543/$1 [L,P]
7     ProxyPreserveHost on
8     RequestHeader add X-Vhm-Root /cms
9 </VirtualHost>
```

 Use of the `RequestHeader` directive requires that the Apache `mod_headers` module be available in the Apache environment you’re using.

For a Pyramid application running under *mod_wsgi*, the same can be achieved using `SetEnv`:

```
1 <Location />
2     SetEnv HTTP_X_VHM_ROOT /cms
3 </Location>
```

Setting a virtual root has no effect when using an application based on *URL dispatch*.

Further Documentation and Examples

The API documentation in *pyramid.traversal* documents a *pyramid.traversal.virtual_root()* API. When called, it returns the virtual root object (or the physical root object if no virtual root has been specified).

Running a Pyramid Application under mod_wsgi has detailed information about using *mod_wsgi* to serve Pyramid applications.

Unit, Integration, and Functional Testing

Unit testing is, not surprisingly, the act of testing a “unit” in your application. In this context, a “unit” is often a function or a method of a class instance. The unit is also referred to as a “unit under test”.

The goal of a single unit test is to test **only** some permutation of the “unit under test”. If you write a unit test that aims to verify the result of a particular codepath through a Python function, you need only be

concerned about testing the code that *lives in the function body itself*. If the function accepts a parameter that represents a complex application “domain object” (such as a resource, a database connection, or an SMTP server), the argument provided to this function during a unit test *need not be* and likely *should not be* a “real” implementation object. For example, although a particular function implementation may accept an argument that represents an SMTP server object, and the function may call a method of this object when the system is operating normally that would result in an email being sent, a unit test of this codepath of the function does *not* need to test that an email is actually sent. It just needs to make sure that the function calls the method of the object provided as an argument that *would* send an email if the argument happened to be the “real” implementation of an SMTP server object.

An *integration test*, on the other hand, is a different form of testing in which the interaction between two or more “units” is explicitly tested. Integration tests verify that the components of your application work together. You *might* make sure that an email was actually sent in an integration test.

A *functional test* is a form of integration test in which the application is run “literally”. You would *have to* make sure that an email was actually sent in a functional test, because it tests your code end to end.

It is often considered best practice to write each type of tests for any given codebase. Unit testing often provides the opportunity to obtain better “coverage”: it’s usually possible to supply a unit under test with arguments and/or an environment which causes *all* of its potential codepaths to be executed. This is usually not as easy to do with a set of integration or functional tests, but integration and functional testing provides a measure of assurance that your “units” work together, as they will be expected to when your application is run in production.

The suggested mechanism for unit and integration testing of a Pyramid application is the Python `unittest` module. Although this module is named `unittest`, it is actually capable of driving both unit and integration tests. A good `unittest` tutorial is available within Dive Into Python by Mark Pilgrim.

Pyramid provides a number of facilities that make unit, integration, and functional tests easier to write. The facilities become particularly useful when your code calls into Pyramid-related framework functions.

Test Set Up and Tear Down

Pyramid uses a “global” (actually *thread local*) data structure to hold two items: the current *request* and the current *application registry*. These data structures are available via the `pyramid.threadlocal.get_current_request()` and `pyramid.threadlocal.get_current_registry()` functions, respectively. See *Thread Locals* for information about these functions and the data structures they return.

If your code uses these `get_current_*` functions or calls Pyramid code which uses `get_current_*` functions, you will need to call `pyramid.testing.setUp()` in your test setup

and you will need to call `pyramid.testing.tearDown()` in your test teardown. `setUp()` pushes a registry onto the *thread local* stack, which makes the `get_current_*` functions work. It returns a *Configurator* object which can be used to perform extra configuration required by the code under test. `tearDown()` pops the thread local stack.

Normally when a *Configurator* is used directly with the `main` block of a Pyramid application, it defers performing any “real work” until its `.commit` method is called (often implicitly by the `pyramid.config.Configurator.make_wsgi_app()` method). The *Configurator* returned by `setUp()` is an *autocommitting* *Configurator*, however, which performs all actions implied by methods called on it immediately. This is more convenient for unit testing purposes than needing to call `pyramid.config.Configurator.commit()` in each test after adding extra configuration statements.

The use of the `setUp()` and `tearDown()` functions allows you to supply each unit test method in a test case with an environment that has an isolated registry and an isolated request for the duration of a single test. Here’s an example of using this feature:

```
1 import unittest
2 from pyramid import testing
3
4 class MyTest(unittest.TestCase):
5     def setUp(self):
6         self.config = testing.setUp()
7
8     def tearDown(self):
9         testing.tearDown()
```

The above will make sure that `get_current_registry()` called within a test case method of `MyTest` will return the *application registry* associated with the `config` *Configurator* instance. Each test case method attached to `MyTest` will use an isolated registry.

The `setUp()` and `tearDown()` functions accept various arguments that influence the environment of the test. See the `pyramid.testing` API for information about the extra arguments supported by these functions.

If you also want to make `get_current_request()` return something other than `None` during the course of a single test, you can pass a *request* object into the `pyramid.testing.setUp()` within the `setUp` method of your test:

```
1 import unittest
2 from pyramid import testing
3
4 class MyTest(unittest.TestCase):
5     def setUp(self):
6         request = testing.DummyRequest()
```

```
7         self.config = testing.setUp(request=request)
8
9     def tearDown(self):
10         testing.tearDown()
```

If you pass a *request* object into `pyramid.testing.setUp()` within your test case's `setUp`, any test method attached to the `MyTest` test case that directly or indirectly calls `get_current_request()` will receive the request object. Otherwise, during testing, `get_current_request()` will return `None`. We use a “dummy” request implementation supplied by `pyramid.testing.DummyRequest` because it's easier to construct than a “real” Pyramid request object.

Test setup using a context manager

An alternative style of setting up a test configuration is to use the `with` statement and `pyramid.testing.testConfig()` to create a context manager. The context manager will call `pyramid.testing.setUp()` before the code under test and `pyramid.testing.tearDown()` afterwards.

This style is useful for small self-contained tests. For example:

```
1 import unittest
2
3 class MyTest(unittest.TestCase):
4
5     def test_my_function(self):
6         from pyramid import testing
7         with testing.testConfig() as config:
8             config.add_route('bar', '/bar/{id}')
9             my_function_which_needs_route_bar()
```

What?


Thread local data structures are always a bit confusing, especially when they're used by frameworks. Sorry. So here's a rule of thumb: if you don't *know* whether you're calling code that uses the `get_current_registry()` or `get_current_request()` functions, or you don't care about any of this, but you still want to write test code, just always call `pyramid.testing.setUp()` in your test's `setUp` method and `pyramid.testing.tearDown()` in your tests' `tearDown` method. This won't really hurt anything if the application you're testing does not call any `get_current*` function.

Using the Configurator and `pyramid.testing` APIs in Unit Tests

The Configurator API and the `pyramid.testing` module provide a number of functions which can be used during unit testing. These functions make *configuration declaration* calls to the current *application registry*, but typically register a “stub” or “dummy” feature in place of the “real” feature that the code would call if it was being run normally.

For example, let’s imagine you want to unit test a Pyramid view function.

```
1 from pyramid.httpexceptions import HTTPForbidden
2
3 def view_fn(request):
4     if request.has_permission('edit'):
5         raise HTTPForbidden
6     return {'greeting': 'hello'}
```

 This code implies that you have defined a renderer imperatively in a relevant `pyramid.config.Configurator` instance, otherwise it would fail when run normally.

Without doing anything special during a unit test, the call to `has_permission()` in this view function will always return a `True` value. When a Pyramid application starts normally, it will populate an *application registry* using *configuration declaration* calls made against a *Configurator*. But if this application registry is not created and populated (e.g., by initializing the configurator with an authorization policy), like when you invoke application code via a unit test, Pyramid API functions will tend to either fail or return default results. So how do you test the branch of the code in this view function that raises `HTTPForbidden`?

The testing API provided by Pyramid allows you to simulate various application registry registrations for use under a unit testing framework without needing to invoke the actual application configuration implied by its main function. For example, if you wanted to test the above `view_fn` (assuming it lived in the package named `my.package`), you could write a `unittest.TestCase` that used the testing API.

```
1 import unittest
2 from pyramid import testing
3
4 class MyTest(unittest.TestCase):
5     def setUp(self):
6         self.config = testing.setUp()
7
8     def tearDown(self):
9         testing.tearDown()
```

```

10
11     def test_view_fn_forbidden(self):
12         from pyramid.httpexceptions import HTTPForbidden
13         from my.package import view_fn
14         self.config.testing_securitypolicy(userid='hank',
15                                           permissive=False)
16         request = testing.DummyRequest()
17         request.context = testing.DummyResource()
18         self.assertRaises(HTTPForbidden, view_fn, request)
19
20     def test_view_fn_allowed(self):
21         from my.package import view_fn
22         self.config.testing_securitypolicy(userid='hank',
23                                           permissive=True)
24         request = testing.DummyRequest()
25         request.context = testing.DummyResource()
26         response = view_fn(request)
27         self.assertEqual(response, {'greeting': 'hello'})

```

In the above example, we create a `MyTest` test case that inherits from `unittest.TestCase`. If it's in our Pyramid application, it will be found when `py.test` is run. It has two test methods.

The first test method, `test_view_fn_forbidden` tests the `view_fn` when the authentication policy forbids the current user the `edit` permission. Its third line registers a “dummy” “non-permissive” authorization policy using the `testing_securitypolicy()` method, which is a special helper method for unit testing.

We then create a `pyramid.testing.DummyRequest` object which simulates a `WebOb` request object API. A `pyramid.testing.DummyRequest` is a request object that requires less setup than a “real” Pyramid request. We call the function being tested with the manufactured request. When the function is called, `pyramid.request.Request.has_permission()` will call the “dummy” authentication policy we've registered through `testing_securitypolicy()`, which denies access. We check that the view function raises a `HTTPForbidden` error.

The second test method, named `test_view_fn_allowed`, tests the alternate case, where the authentication policy allows access. Notice that we pass different values to `testing_securitypolicy()` to obtain this result. We assert at the end of this that the view function returns a value.

Note that the test calls the `pyramid.testing.setUp()` function in its `setUp` method and the `pyramid.testing.tearDown()` function in its `tearDown` method. We assign the result of `pyramid.testing.setUp()` as `config` on the `unittest` class. This is a *Configurator* object and all methods of the configurator can be called as necessary within tests. If you use any of the *Configurator* APIs during testing, be sure to use this pattern in your test case's `setUp` and `tearDown`; these methods make sure you're using a “fresh” *application registry* per test run.

See the *pyramid.testing* chapter for the entire Pyramid-specific testing API. This chapter describes APIs for registering a security policy, registering resources at paths, registering event listeners, registering views and view permissions, and classes representing “dummy” implementations of a request and a resource.

See also:

See also the various methods of the *Configurator* documented in *pyramid.config* that begin with the `testing_` prefix.

Creating Integration Tests

In Pyramid, a *unit test* typically relies on “mock” or “dummy” implementations to give the code under test enough context to run.

“Integration testing” implies another sort of testing. In the context of a Pyramid integration test, the test logic exercises the functionality of the code under test *and* its integration with the rest of the Pyramid framework.

Creating an integration test for a Pyramid application usually means invoking the application’s `includeme` function via `pyramid.config.Configurator.include()` within the test’s setup code. This causes the entire Pyramid environment to be set up, simulating what happens when your application is run “for real”. This is a heavy-hammer way of making sure that your tests have enough context to run properly, and tests your code’s integration with the rest of Pyramid.

See also:

See also *Including Configuration from External Sources*

Writing unit tests that use the *Configurator* API to set up the right “mock” registrations is often preferred to creating integration tests. Unit tests will run faster (because they do less for each test) and are usually easier to reason about.

Creating Functional Tests

Functional tests test your literal application.

In Pyramid, functional tests are typically written using the *WebTest* package, which provides APIs for invoking HTTP(S) requests to your application. We also like `py.test` and `pytest-cov` to provide simple testing and coverage reports.

Regardless of which testing *package* you use, be sure to add a `tests_require` dependency on that package to your application’s `setup.py` file. Using the project `MyProject` generated by the starter scaffold as described in *Creating a Pyramid Project*, we would insert the following code immediately following the `requires` block in the file `MyProject/setup.py`.

```

11 requires = [
12     'pyramid',
13     'pyramid_chameleon',
14     'pyramid_debugtoolbar',
15     'waitress',
16 ]
17
18 tests_require = [
19     'WebTest >= 1.3.1', # py3 compat
20     'pytest', # includes virtualenv
21     'pytest-cov',
22 ]

```

Remember to change the dependency.

```

40     zip_safe=False,
41     extras_require={
42         'testing': tests_require,
43     },
44     install_requires=requires,

```

As always, whenever you change your dependencies, make sure to run the correct `pip install -e` command.

```
$VENV/bin/pip install -e ".[testing]"
```

In your `MyPackage` project, your *package* is named `myproject` which contains a `views` module, which in turn contains a *view* function `my_view` that returns an HTML body when the root URL is invoked:

```

1 from pyramid.view import view_config
2
3
4 @view_config(route_name='home', renderer='templates/mytemplate.pt
  ↪')
5 def my_view(request):
6     return {'project': 'MyProject'}

```

The following example functional test demonstrates invoking the above *view*:


```
1 class FunctionalTests(unittest.TestCase):
2     def setUp(self):
3         from myproject import main
4         app = main({})
5         from webtest import TestApp
6         self.testapp = TestApp(app)
7
8     def test_root(self):
9         res = self.testapp.get('/', status=200)
10        self.assertTrue(b'Pyramid' in res.body)
```

When this test is run, each test method creates a “real” *WSGI* application using the `main` function in your `myproject.__init__` module, using *WebTest* to wrap that *WSGI* application. It assigns the result to `self.testapp`. In the test named `test_root`, the `TestApp`’s `GET` method is used to invoke the root URL. Finally, an assertion is made that the returned HTML contains the text `Pyramid`.

See the *WebTest* documentation for further information about the methods available to a `webtest.app.TestApp` instance.

Resources

A *resource* is an object that represents a “place” in a tree related to your application. Every Pyramid application has at least one resource object: the *root* resource. Even if you don’t define a root resource manually, a default one is created for you. The root resource is the root of a *resource tree*. A resource tree is a set of nested dictionary-like objects which you can use to represent your website’s structure.

In an application which uses *traversal* to map URLs to code, the resource tree structure is used heavily to map each URL to a *view callable*. When *traversal* is used, Pyramid will walk through the resource tree by traversing through its nested dictionary structure in order to find a *context* resource. Once a context resource is found, the context resource and data in the request will be used to find a *view callable*.

In an application which uses *URL dispatch*, the resource tree is only used indirectly, and is often “invisible” to the developer. In URL dispatch applications, the resource “tree” is often composed of only the root resource by itself. This root resource sometimes has security declarations attached to it, but is not required to have any. In general, the resource tree is much less important in applications that use URL dispatch than applications that use traversal.

In “Zope-like” Pyramid applications, resource objects also often store data persistently, and offer methods related to mutating that persistent data. In these kinds of applications, resources not only represent the site structure of your website, but they become the *domain model* of the application.

Also:

- The `context` and `containment` predicate arguments to `add_view()` (or a `view_config()` decorator) reference a resource class or resource *interface*.
- A *root factory* returns a resource.
- A resource is exposed to *view* code as the *context* of a view.
- Various helpful Pyramid API methods expect a resource as an argument (e.g., `resource_url()` and others).

Defining a Resource Tree

When *traversal* is used (as opposed to a purely *URL dispatch* based application), Pyramid expects to be able to traverse a tree composed of resources (the *resource tree*). Traversal begins at a root resource, and descends into the tree recursively, trying each resource's `__getitem__` method to resolve a path segment to another resource object. Pyramid imposes the following policy on resource instances in the tree:

- A container resource (a resource which contains other resources) must supply a `__getitem__` method which is willing to resolve a Unicode name to a sub-resource. If a sub-resource by a particular name does not exist in a container resource, the `__getitem__` method of the container resource must raise a `KeyError`. If a sub-resource by that name *does* exist, the container's `__getitem__` should return the sub-resource.
- Leaf resources, which do not contain other resources, must not implement a `__getitem__`, or if they do, their `__getitem__` method must always raise a `KeyError`.

See *Traversal* for more information about how traversal works against resource instances.

Here's a sample resource tree, represented by a variable named `root`:

```
1 class Resource(dict):
2     pass
3
4 root = Resource({'a':Resource({'b':Resource({'c':Resource()})})})
```

The resource tree we've created above is represented by a dictionary-like root object which has a single child named `'a'`. `'a'` has a single child named `'b'`, and `'b'` has a single child named `'c'`, which has no children. It is therefore possible to access the `'c'` leaf resource like so:

```
1 root['a']['b']['c']
```

If you returned the above `root` object from a *root factory*, the path `/a/b/c` would find the `'c'` object in the resource tree as the result of *traversal*.

In this example, each of the resources in the tree is of the same class. This is not a requirement. Resource elements in the tree can be of any type. We used a single class to represent all resources in the tree for the sake of simplicity, but in a “real” app, the resources in the tree can be arbitrary.

Although the example tree above can service a traversal, the resource instances in the above example are not aware of *location*, so their utility in a “real” application is limited. To make best use of built-in Pyramid API facilities, your resources should be “location-aware”. The next section details how to make resources location-aware.

Location-Aware Resources

In order for certain Pyramid location, security, URL-generation, and traversal APIs to work properly against the resources in a resource tree, all resources in the tree must be *location-aware*. This means they must have two attributes: `__parent__` and `__name__`.

The `__parent__` attribute of a location-aware resource should be a reference to the resource’s parent resource instance in the tree. The `__name__` attribute should be the name with which a resource’s parent refers to the resource via `__getitem__`.

The `__parent__` of the root resource should be `None` and its `__name__` should be the empty string. For instance:

```
1 class MyRootResource(object):
2     __name__ = ''
3     __parent__ = None
```

A resource returned from the root resource’s `__getitem__` method should have a `__parent__` attribute that is a reference to the root resource, and its `__name__` attribute should match the name by which it is reachable via the root resource’s `__getitem__`. A container resource within the root resource should have a `__getitem__` that returns resources with a `__parent__` attribute that points at the container, and these sub-objects should have a `__name__` attribute that matches the name by which they are retrieved from the container via `__getitem__`. This pattern continues recursively “up” the tree from the root.

The `__parent__` attributes of each resource form a linked list that points “downwards” toward the root. This is analogous to the `..` entry in filesystem directories. If you follow the `__parent__` values from any resource in the resource tree, you will eventually come to the root resource, just like if you keep executing the `cd ..` filesystem command, eventually you will reach the filesystem root directory.



If your root resource has a `__name__` argument that is not `None` or the empty string, URLs returned by the `resource_url()` function, and paths generated by the `resource_path()` and `resource_path_tuple()` APIs, will be generated improperly. The value of `__name__` will be prepended to every path and URL generated (as opposed to a single leading slash or empty tuple element).

For your convenience

If you’d rather not manage the `__name__` and `__parent__` attributes of your resources “by hand”, an add-on package named `pyramid_traversalwrapper` can help.

In order to use this helper feature, you must first install the `pyramid_traversalwrapper` package (available via PyPI), then register its `ModelGraphTraverser` as the traversal policy, rather than the default `Pyramid` traverser. The package contains instructions for doing so.

Once `Pyramid` is configured with this feature, you will no longer need to manage the `__parent__` and `__name__` attributes on resource objects “by hand”. Instead, as necessary during traversal, `Pyramid` will wrap each resource (even the root resource) in a `LocationProxy`, which will dynamically assign a `__name__` and a `__parent__` to the traversed resource, based on the last traversed resource and the name supplied to `__getitem__`. The root resource will have a `__name__` attribute of `None` and a `__parent__` attribute of `None`.

Applications which use tree-walking `Pyramid` APIs require location-aware resources. These APIs include (but are not limited to) `resource_url()`, `find_resource()`, `find_root()`, `find_interface()`, `resource_path()`, `resource_path_tuple()`, `traverse()`, `virtual_root()`, and (usually) `has_permission()` and `principals_allowed_by_permission()`.

In general, since so much `Pyramid` infrastructure depends on location-aware resources, it’s a good idea to make each resource in your tree location-aware.

Generating the URL of a Resource

If your resources are *location-aware*, you can use the `pyramid.request.Request.resource_url()` API to generate a URL for the resource. This URL will use the resource's position in the parent tree to create a resource path, and it will prefix the path with the current application URL to form a fully-qualified URL with the scheme, host, port, and path. You can also pass extra arguments to `resource_url()` to influence the generated URL.

The simplest call to `resource_url()` looks like this:

```
1 url = request.resource_url(resource)
```

The `request` in the above example is an instance of a Pyramid *request* object.

If the resource referred to as `resource` in the above example was the root resource, and the host that was used to contact the server was `example.com`, the URL generated would be `http://example.com/`. However, if the resource was a child of the root resource named `a`, the generated URL would be `http://example.com/a/`.

A slash is appended to all resource URLs when `resource_url()` is used to generate them in this simple manner, because resources are “places” in the hierarchy, and URLs are meant to be clicked on to be visited. Relative URLs that you include on HTML pages rendered as the result of the default view of a resource are more apt to be relative to these resources than relative to their parent.

You can also pass extra elements to `resource_url()`:

```
1 url = request.resource_url(resource, 'foo', 'bar')
```

If the resource referred to as `resource` in the above example was the root resource, and the host that was used to contact the server was `example.com`, the URL generated would be `http://example.com/foo/bar`. Any number of extra elements can be passed to `resource_url()` as extra positional arguments. When extra elements are passed, they are appended to the resource's URL. A slash is not appended to the final segment when elements are passed.

You can also pass a query string:

```
1 url = request.resource_url(resource, query={'a': '1'})
```

If the resource referred to as `resource` in the above example was the root resource, and the host that was used to contact the server was `example.com`, the URL generated would be `http://example.com/?a=1`.

When a *virtual root* is active, the URL generated by `resource_url()` for a resource may be “shorter” than its physical tree path. See *Virtual Root Support* for more information about virtually rooting a resource.

For more information about generating resource URLs, see the documentation for `pyramid.request.Request.resource_url()`.

Overriding Resource URL Generation

If a resource object implements a `__resource_url__` method, this method will be called when `resource_url()` is called to generate a URL for the resource, overriding the default URL returned for the resource by `resource_url()`.

The `__resource_url__` hook is passed two arguments: `request` and `info`. `request` is the `request` object passed to `resource_url()`. `info` is a dictionary with the following keys:

physical_path A string representing the “physical path” computed for the resource, as defined by `pyramid.traversal.resource_path(resource)`. It will begin and end with a slash.

virtual_path A string representing the “virtual path” computed for the resource, as defined by *Virtual Root Support*. This will be identical to the physical path if virtual rooting is not enabled. It will begin and end with a slash.

app_url A string representing the application URL generated during `request.resource_url`. It will not end with a slash. It represents a potentially customized URL prefix, containing potentially custom scheme, host and port information passed by the user to `request.resource_url`. It should be preferred over use of `request.application_url`.

The `__resource_url__` method of a resource should return a string representing a URL. If it cannot override the default, it should return `None`. If it returns `None`, the default URL will be returned.

Here’s an example `__resource_url__` method.

```
1 class Resource(object):
2     def __resource_url__(self, request, info):
3         return info['app_url'] + info['virtual_path']
```

The above example actually just generates and returns the default URL, which would have been what was generated by the default `resource_url` machinery, but your code can perform arbitrary logic as necessary. For example, your code may wish to override the hostname or port number of the generated URL.

Note that the URL generated by `__resource_url__` should be fully qualified, should end in a slash, and should not contain any query string or anchor elements (only path elements) to work with `resource_url()`.

Generating the Path To a Resource

`pyramid.traversal.resource_path()` returns a string object representing the absolute physical path of the resource object based on its position in the resource tree. Each segment of the path is separated with a slash character.

```
1 from pyramid.traversal import resource_path
2 url = resource_path(resource)
```

If `resource` in the example above was accessible in the tree as `root['a']['b']`, the above example would generate the string `/a/b`.

Any positional arguments passed in to `resource_path()` will be appended as path segments to the end of the resource path.

```
1 from pyramid.traversal import resource_path
2 url = resource_path(resource, 'foo', 'bar')
```

If `resource` in the example above was accessible in the tree as `root['a']['b']`, the above example would generate the string `/a/b/foo/bar`.

The resource passed in must be *location-aware*.

The presence or absence of a *virtual root* has no impact on the behavior of `resource_path()`.

Finding a Resource by Path

If you have a string path to a resource, you can grab the resource from that place in the application's resource tree using `pyramid.traversal.find_resource()`.

You can resolve an absolute path by passing a string prefixed with a `/` as the `path` argument:

```
1 from pyramid.traversal import find_resource
2 url = find_resource(anyresource, '/path')
```

Or you can resolve a path relative to the resource that you pass in to `pyramid.traversal.find_resource()` by passing a string that isn't prefixed by `/`:

```
1 from pyramid.traversal import find_resource
2 url = find_resource(anyresource, 'path')
```

Often the paths you pass to `find_resource()` are generated by the `resource_path()` API. These APIs are “mirrors” of each other.

If the path cannot be resolved when calling `find_resource()` (if the respective resource in the tree does not exist), a `KeyError` will be raised.

See the `pyramid.traversal.find_resource()` documentation for more information about resolving a path to a resource.

Obtaining the Lineage of a Resource

`pyramid.location.lineage()` returns a generator representing the *lineage* of the *location*-aware *resource* object.

The `lineage()` function returns the resource that is passed into it, then each parent of the resource in order. For example, if the resource tree is composed like so:

```
1 class Thing(object): pass
2
3 thing1 = Thing()
4 thing2 = Thing()
5 thing2.__parent__ = thing1
```

Calling `lineage(thing2)` will return a generator. When we turn it into a list, we will get:

```
1 list(lineage(thing2))
2 [ <Thing object at thing2>, <Thing object at thing1> ]
```

The generator returned by `lineage()` first returns unconditionally the resource that was passed into it. Then, if the resource supplied a `__parent__` attribute, it returns the resource represented by `resource.__parent__`. If *that* resource has a `__parent__` attribute, it will return that resource's parent, and so on, until the resource being inspected either has no `__parent__` attribute or has a `__parent__` attribute of `None`.

See the documentation for `pyramid.location.lineage()` for more information.

Determining if a Resource is in the Lineage of Another Resource

Use the `pyramid.location.inside()` function to determine if one resource is in the *lineage* of another resource.

For example, if the resource tree is:

```
1 class Thing(object): pass
2
3 a = Thing()
4 b = Thing()
5 b.__parent__ = a
```


Calling `inside(b, a)` will return `True`, because `b` has a lineage that includes `a`. However, calling `inside(a, b)` will return `False` because `a` does not have a lineage that includes `b`.

The argument list for `inside()` is `(resource1, resource2)`. `resource1` is “inside” `resource2` if `resource2` is a *lineage* ancestor of `resource1`. It is a lineage ancestor if its parent (or one of its parent’s parents, etc.) is an ancestor.

See `pyramid.location.inside()` for more information.

Finding the Root Resource

Use the `pyramid.traversal.find_root()` API to find the *root* resource. The root resource is the resource at the root of the *resource tree*. The API accepts a single argument: `resource`. This is a resource that is *location*-aware. It can be any resource in the tree for which you want to find the root.

For example, if the resource tree is:

```
1 class Thing(object): pass
2
3 a = Thing()
4 b = Thing()
5 b.__parent__ = a
```

Calling `find_root(b)` will return `a`.

The root resource is also available as `request.root` within *view callable* code.

The presence or absence of a *virtual root* has no impact on the behavior of `find_root()`. The root object returned is always the *physical* root object.

Resources Which Implement Interfaces

Resources can optionally be made to implement an *interface*. An interface is used to tag a resource object with a “type” that later can be referred to within *view configuration* and by `pyramid.traversal.find_interface()`.

Specifying an interface instead of a class as the `context` or `containment` predicate arguments within *view configuration* statements makes it possible to use a single view callable for more than one class of resource objects. If your application is simple enough that you see no reason to want to do this, you can skip reading this section of the chapter.

For example, here’s some code which describes a blog entry which also declares that the blog entry implements an *interface*.

```

1 import datetime
2 from zope.interface import implementer
3 from zope.interface import Interface
4
5 class IBlogEntry(Interface):
6     pass
7
8 @implementer(IBlogEntry)
9 class BlogEntry(object):
10     def __init__(self, title, body, author):
11         self.title = title
12         self.body = body
13         self.author = author
14         self.created = datetime.datetime.now()

```

This resource consists of two things: the class which defines the resource constructor as the class `BlogEntry`, and an *interface* attached to the class via an `implementer` class decorator using the `IBlogEntry` interface as its sole argument.

The interface object used must be an instance of a class that inherits from `zope.interface.Interface`.

A resource class may implement zero or more interfaces. You specify that a resource implements an interface by using the `zope.interface.implementer()` function as a class decorator. The above `BlogEntry` resource implements the `IBlogEntry` interface.

You can also specify that a particular resource *instance* provides an interface as opposed to its class. When you declare that a class implements an interface, all instances of that class will also provide that interface. However, you can also just say that a single object provides the interface. To do so, use the `zope.interface.directlyProvides()` function:

```

1 import datetime
2 from zope.interface import directlyProvides
3 from zope.interface import Interface
4
5 class IBlogEntry(Interface):
6     pass
7
8 class BlogEntry(object):
9     def __init__(self, title, body, author):
10         self.title = title
11         self.body = body
12         self.author = author
13         self.created = datetime.datetime.now()

```

```
14 |
15 | entry = BlogEntry('title', 'body', 'author')
16 | directlyProvides(entry, IBlogEntry)
```

`zope.interface.directlyProvides()` will replace any existing interface that was previously provided by an instance. If a resource object already has instance-level interface declarations that you don't want to replace, use the `zope.interface.alsoProvides()` function:

```
1 | import datetime
2 | from zope.interface import alsoProvides
3 | from zope.interface import directlyProvides
4 | from zope.interface import Interface
5 |
6 | class IBlogEntry1(Interface):
7 |     pass
8 |
9 | class IBlogEntry2(Interface):
10 |     pass
11 |
12 | class BlogEntry(object):
13 |     def __init__(self, title, body, author):
14 |         self.title = title
15 |         self.body = body
16 |         self.author = author
17 |         self.created = datetime.datetime.now()
18 |
19 | entry = BlogEntry('title', 'body', 'author')
20 | directlyProvides(entry, IBlogEntry1)
21 | alsoProvides(entry, IBlogEntry2)
```

`zope.interface.alsoProvides()` will augment the set of interfaces directly provided by an instance instead of overwriting them like `zope.interface.directlyProvides()` does.

For more information about how resource interfaces can be used by view configuration, see *Using Resource Interfaces in View Configuration*.

Finding a Resource with a Class or Interface in Lineage

Use the `find_interface()` API to locate a parent that is of a particular Python class, or which implements some *interface*.

For example, if your resource tree is composed as follows:

```
1 class Thing1(object): pass
2 class Thing2(object): pass
3
4 a = Thing1()
5 b = Thing2()
6 b.__parent__ = a
```

Calling `find_interface(a, Thing1)` will return the `a` resource because `a` is of class `Thing1` (the resource passed as the first argument is considered first, and is returned if the class or interface specification matches).

Calling `find_interface(b, Thing1)` will return the `a` resource because `a` is of class `Thing1` and `a` is the first resource in `b`'s lineage of this class.

Calling `find_interface(b, Thing2)` will return the `b` resource.

The second argument to `find_interface` may also be a *interface* instead of a class. If it is an interface, each resource in the lineage is checked to see if the resource implements the specified interface (instead of seeing if the resource is of a class).

See also:

See also *Resources Which Implement Interfaces*.

Pyramid API Functions That Act Against Resources

A resource object is used as the *context* provided to a view. See *Traversal* and *URL Dispatch* for more information about how a resource object becomes the context.

The APIs provided by *pyramid.traversal* are used against resource objects. These functions can be used to find the “path” of a resource, the root resource in a resource tree, or to generate a URL for a resource.

The APIs provided by *pyramid.location* are used against resources. These can be used to walk down a resource tree, or conveniently locate one resource “inside” another.

Some APIs on the *pyramid.request.Request* accept a resource object as a parameter. For example, the *has_permission()* API accepts a resource object as one of its arguments; the ACL is obtained from this resource or one of its ancestors. Other security related APIs on the *pyramid.request.Request* class also accept *context* as an argument, and a context is always a resource.

Hello Traversal World

Traversal is an alternative to URL dispatch which allows Pyramid applications to map URLs to code.

If code speaks louder than words, maybe this will help. Here is a single-file Pyramid application that uses traversal:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 class Resource(dict):
6     pass
7
8 def get_root(request):
9     return Resource({'a': Resource({'b': Resource({'c': Resource()}))})
10
11 def hello_world_of_resources(context, request):
12     output = "Here's a resource and its children: %s" % context
13     return Response(output)
14
15 if __name__ == '__main__':
16     config = Configurator(root_factory=get_root)
17     config.add_view(hello_world_of_resources, context=Resource)
18     app = config.make_wsgi_app()
19     server = make_server('0.0.0.0', 8080, app)
20     server.serve_forever()
21
22
```

You may notice that this application is intentionally very similar to the “hello world” application from *Creating Your First Pyramid Application*.

On lines 5-6, we create a trivial *resource* class that’s just a dictionary subclass.

On lines 8-9, we hard-code a *resource tree* in our *root factory* function.

On lines 11-13, we define a single *view callable* that can display a single instance of our `Resource` class, passed as the `context` argument.

The rest of the file sets up and serves our Pyramid WSGI app. Line 18 is where our view gets configured for use whenever the traversal ends with an instance of our `Resource` class.

Interestingly, there are no URLs explicitly configured in this application. Instead, the URL space is defined entirely by the keys in the resource tree.

Example requests

If this example is running on `http://localhost:8080`, and the user browses to `http://localhost:8080/a/b`, Pyramid will call `get_root(request)` to get the root resource, then traverse the tree from there by key; starting from the root, it will find the child with key "a", then its child with key "b"; then use that as the `context` argument for calling `hello_world_of_resources`.

Or, if the user browses to `http://localhost:8080/`, Pyramid will stop at the root—the outermost `Resource` instance, in this case—and use that as the `context` argument to the same view.

Or, if the user browses to a key that doesn't exist in this resource tree, like `http://localhost:8080/xyz` or `http://localhost:8080/a/b/c/d`, the traversal will end by raising a `KeyError`, and Pyramid will turn that into a 404 HTTP response.

A more complicated application could have many types of resources, with different view callables defined for each type, and even multiple views for each type.

See also:

Full technical details may be found in *Traversal*.

For more about *why* you might use traversal, see *Much Ado About Traversal*.

Much Ado About Traversal

(Or, why you should care about it.)



This chapter was adapted, with permission, from a blog post by Rob Miller.

Traversal is an alternative to *URL dispatch* which allows Pyramid applications to map URLs to code.



Ex-Zope users who are already familiar with traversal and view lookup conceptually may want to skip directly to the *Traversal* chapter, which discusses technical details. This chapter is mostly aimed at people who have previous *Pylons* experience or experience in another framework which does not provide traversal, and need an introduction to the “why” of traversal.

Some folks who have been using Pylons and its Routes-based URL matching for a long time are being exposed for the first time, via Pyramid, to new ideas such as “*traversal*” and “*view lookup*” as a way to route incoming HTTP requests to callable code. Some of the same folks believe that traversal is hard to understand. Others question its usefulness; URL matching has worked for them so far, so why should they even consider dealing with another approach, one which doesn’t fit their brain and which doesn’t provide any immediately obvious value?

You can be assured that if you don’t want to understand traversal, you don’t have to. You can happily build Pyramid applications with only *URL dispatch*. However, there are some straightforward, real-world use cases that are much more easily served by a traversal-based approach than by a pattern-matching mechanism. Even if you haven’t yet hit one of these use cases yourself, understanding these new ideas is worth the effort for any web developer so you know when you might want to use them. *Traversal* is actually a straightforward metaphor easily comprehended by anyone who’s ever used a run-of-the-mill file system with folders and files.

URL Dispatch

Let’s step back and consider the problem we’re trying to solve. An HTTP request for a particular path has been routed to our web application. The requested path will possibly invoke a specific *view callable* function defined somewhere in our app. We’re trying to determine *which* callable function, if any, should be invoked for a given requested URL.

Many systems, including Pyramid, offer a simple solution. They offer the concept of “URL matching”. URL matching approaches this problem by parsing the URL path and comparing the results to a set of registered “patterns”, defined by a set of regular expressions or some other URL path templating syntax. Each pattern is mapped to a callable function somewhere; if the request path matches a specific pattern, the associated function is called. If the request path matches more than one pattern, some conflict resolution scheme is used, usually a simple order precedence so that the first match will take priority over any subsequent matches. If a request path doesn’t match any of the defined patterns, a “404 Not Found” response is returned.

In Pyramid, we offer an implementation of URL matching which we call *URL dispatch*. Using Pyramid syntax, we might have a match pattern such as `/ {userid} / photos / {photoid}`, mapped to a `photo_view()` function defined somewhere in our code. Then a request for a path such as `/ joeschmoe / photos / photo1` would be a match, and the `photo_view()` function would be invoked to handle the request. Similarly, `/ {userid} / blog / {year} / {month} / {postid}` might map to a `blog_post_view()` function, so `/ joeschmoe / blog / 2010 / 12 / urlmatching` would trigger the function, which presumably would know how to find and render the `urlmatching` blog post.

Historical Refresher

Now that we’ve refreshed our understanding of *URL dispatch*, we’ll dig in to the idea of traversal. Before we do, though, let’s take a trip down memory lane. If you’ve been doing web work for a while, you may remember a time when we didn’t have fancy web frameworks like *Pylons* and *Pyramid*. Instead, we had general purpose HTTP servers that primarily served files off of a file system. The “root” of a given site mapped to a particular folder somewhere on the file system. Each segment of the request URL path represented a subdirectory. The final path segment would be either a directory or a file, and once the server found the right file it would package it up in an HTTP response and send it back to the client. So serving up a request for `/joeschmoe/photos/photo1` literally meant that there was a `joeschmoe` folder somewhere, which contained a `photos` folder, which in turn contained a `photo1` file. If at any point along the way we find that there is not a folder or file matching the requested path, we return a 404 response.

As the web grew more dynamic, however, a little bit of extra complexity was added. Technologies such as CGI and HTTP server modules were developed. Files were still looked up on the file system, but if the file ended with (for example) `.cgi` or `.php`, or if it lived in a special folder, instead of simply sending the file to the client the server would read the file, execute it using an interpreter of some sort, and then send the output from this process to the client as the final result. The server configuration specified which files would trigger some dynamic code, with the default case being to just serve the static file.

Traversal (a.k.a., Resource Location)

Believe it or not, if you understand how serving files from a file system works, you understand traversal. And if you understand that a server might do something different based on what type of file a given request specifies, then you understand view lookup.

The major difference between file system lookup and traversal is that a file system lookup steps through nested directories and files in a file system tree, while traversal steps through nested dictionary-type objects in a *resource tree*. Let’s take a detailed look at one of our example paths, so we can see what I mean.

The path `/joeschmoe/photos/photo1`, has four segments: `/`, `joeschmoe`, `photos` and `photo1`. With file system lookup we might have a root folder (`/`) containing a nested folder (`joeschmoe`), which contains another nested folder (`photos`), which finally contains a JPG file (`photo1`). With traversal, we instead have a dictionary-like root object. Asking for the `joeschmoe` key gives us another dictionary-like object. Asking in turn for the `photos` key gives us yet another mapping object, which finally (hopefully) contains the resource that we’re looking for within its values, referenced by the `photo1` key.

In pure Python terms, then, the traversal or “resource location” portion of satisfying the `/joeschmoe/photos/photo1` request will look something like this pseudocode:


```
get_root() ['joeschmoe'] ['photos'] ['photo1']
```

`get_root()` is some function that returns a root traversal *resource*. If all of the specified keys exist, then the returned object will be the resource that is being requested, analogous to the JPG file that was retrieved in the file system example. If a `KeyError` is generated anywhere along the way, Pyramid will return 404. (This isn't precisely true, as you'll see when we learn about view lookup below, but the basic idea holds.)

What Is a “Resource”?

“Files on a file system I understand”, you might say. “But what are these nested dictionary things? Where do these objects, these ‘resources’, live? What *are* they?”

Since Pyramid is not a highly opinionated framework, it makes no restriction on how a *resource* is implemented; a developer can implement them as they wish. One common pattern used is to persist all of the resources, including the root, in a database as a graph. The root object is a dictionary-like object. Dictionary-like objects in Python supply a `__getitem__` method which is called when key lookup is done. Under the hood, when `adict` is a dictionary-like object, Python translates `adict['a']` to `adict.__getitem__('a')`. Try doing this in a Python interpreter prompt if you don't believe us:

```
>>> adict = {}
>>> adict['a'] = 1
>>> adict['a']
1
>>> adict.__getitem__('a')
1
```

The dictionary-like root object stores the ids of all of its subresources as keys, and provides a `__getitem__` implementation that fetches them. So `get_root()` fetches the unique root object, while `get_root() ['joeschmoe']` returns a different object, also stored in the database, which in turn has its own subresources and `__getitem__` implementation, and so on. These resources might be persisted in a relational database, one of the many “NoSQL” solutions that are becoming popular these days, or anywhere else; it doesn't matter. As long as the returned objects provide the dictionary-like API (i.e., as long as they have an appropriately implemented `__getitem__` method), then traversal will work.

In fact, you don't need a “database” at all. You could use plain dictionaries, with your site's URL structure hard-coded directly in the Python source. Or you could trivially implement a set of objects with `__getitem__` methods that search for files in specific directories, and thus precisely recreate the traditional mechanism of having the URL path mapped directly to a folder structure on the file system. Traversal is in fact a superset of file system lookup.



See the chapter entitled *Resources* for a more technical overview of resources.

View Lookup

At this point we’re nearly there. We’ve covered traversal, which is the process by which a specific resource is retrieved according to a specific URL path. But what is “view lookup”?

The need for view lookup is simple: there is more than one possible action that you might want to take after finding a *resource*. With our photo example, for instance, you might want to view the photo in a page, but you might also want to provide a way for the user to edit the photo and any associated metadata. We’ll call the former the `view` view, and the latter will be the `edit` view. (Original, I know.) Pyramid has a centralized *view application registry* where named views can be associated with specific resource types. So in our example, we’ll assume that we’ve registered `view` and `edit` views for photo objects, and that we’ve specified the `view` view as the default, so that `/joeschmoe/photos/photo1/view` and `/joeschmoe/photos/photo1` are equivalent. The `edit` view would sensibly be provided by a request for `/joeschmoe/photos/photo1/edit`.

Hopefully it’s clear that the first portion of the `edit` view’s URL path is going to resolve to the same resource as the non-`edit` version, specifically the resource returned by `get_root() ['joeschmoe'] ['photos'] ['photo1']`. But traversal ends there; the `photo1` resource doesn’t have an `edit` key. In fact, it might not even be a dictionary-like object, in which case `photo1['edit']` would be meaningless. When the Pyramid resource location has been resolved to a *leaf* resource, but the entire request path has not yet been expended, the *very next* path segment is treated as a *view name*. The registry is then checked to see if a view of the given name has been specified for a resource of the given type. If so, the view callable is invoked, with the resource passed in as the related context object (also available as `request.context`). If a view callable could not be found, Pyramid will return a “404 Not Found” response.

You might conceptualize a request for `/joeschmoe/photos/photo1/edit` as ultimately converted into the following piece of Pythonic pseudocode:

```
context = get_root() ['joeschmoe'] ['photos'] ['photo1']
view_callable = get_view(context, 'edit')
request.context = context
view_callable(request)
```

The `get_root` and `get_view` functions don’t really exist. Internally, Pyramid does something more complicated. But the example above is a reasonable approximation of the view lookup algorithm in pseudocode.

Use Cases

Why should we care about traversal? URL matching is easier to explain, and it's good enough, right?

In some cases, yes, but certainly not in all cases. So far we've had very structured URLs, where our paths have had a specific, small number of pieces, like this:

```
/{userid}/{typename}/{objectid}/{view_name}]
```

In all of the examples thus far, we've hard coded the `typename` value, assuming that we'd know at development time what names were going to be used ("photos", "blog", etc.). But what if we don't know what these names will be? Or, worse yet, what if we don't know *anything* about the structure of the URLs inside a user's folder? We could be writing a CMS where we want the end user to be able to arbitrarily add content and other folders inside his folder. He might decide to nest folders dozens of layers deep. How will you construct matching patterns that could account for every possible combination of paths that might develop?


It might be possible, but it certainly won't be easy. The matching patterns are going to become complex quickly as you try to handle all of the edge cases.

With traversal, however, it's straightforward. Twenty layers of nesting would be no problem. Pyramid will happily call `__getitem__` as many times as it needs to, until it runs out of path segments or until a resource raises a `KeyError`. Each resource only needs to know how to fetch its immediate children, and the traversal algorithm takes care of the rest. Also, since the structure of the resource tree can live in the database and not in the code, it's simple to let users modify the tree at runtime to set up their own personalized "directory" structures.

Another use case in which traversal shines is when there is a need to support a context-dependent security policy. One example might be a document management infrastructure for a large corporation, where members of different departments have varying access levels to the various other departments' files. Reasonably, even specific files might need to be made available to specific individuals. Traversal does well here if your resources actually represent the data objects related to your documents, because the idea of a resource authorization is baked right into the code resolution and calling process. Resource objects can store ACLs, which can be inherited and/or overridden by the subresources.

If each resource can thus generate a context-based ACL, then whenever view code is attempting to perform a sensitive action, it can check against that ACL to see whether the current user should be allowed to perform the action. In this way you achieve so called "instance based" or "row level" security which is considerably harder to model using a traditional tabular approach. Pyramid actively supports such a scheme, and in fact if you register your views with guarded permissions and use an authorization policy, Pyramid can check against a resource's ACL when deciding whether or not the view itself is available to the current user.

In summary, there are entire classes of problems that are more easily served by traversal and view lookup than by *URL dispatch*. If your problems don't require it, great, stick with *URL dispatch*. But if you're using Pyramid and you ever find that you *do* need to support one of these use cases, you'll be glad you have traversal in your toolkit.

 It is even possible to mix and match *traversal* with *URL dispatch* in the same Pyramid application. See the *Combining Traversal and URL Dispatch* chapter for details.


Traversal

This chapter explains the technical details of how traversal works in Pyramid.

For a quick example, see *Hello Traversal World*.

For more about *why* you might use traversal, see *Much Ado About Traversal*.

A *traversal* uses the URL (Universal Resource Locator) to find a *resource* located in a *resource tree*, which is a set of nested dictionary-like objects. Traversal is done by using each segment of the path portion of the URL to navigate through the *resource tree*. You might think of this as looking up files and directories in a file system. Traversal walks down the path until it finds a published resource, analogous to a file system “directory” or “file”. The resource found as the result of a traversal becomes the *context* of the *request*. Then, the *view lookup* subsystem is used to find some view code willing to “publish” this resource by generating a *response*.

 Using *Traversal* to map a URL to code is optional. If you're creating your first Pyramid application, it probably makes more sense to use *URL dispatch* to map URLs to code instead of traversal, as new Pyramid developers tend to find URL dispatch slightly easier to understand. If you use URL dispatch, you needn't read this chapter.

Traversal Details

Traversal is dependent on information in a *request* object. Every *request* object contains URL path information in the `PATH_INFO` portion of the *WSGI* environment. The `PATH_INFO` string is the portion of a request's URL following the hostname and port number, but before any query string elements or fragment element. For example the `PATH_INFO` portion of the URL `http://example.com:8080/a/b/c?foo=1` is `/a/b/c`.

Traversal treats the `PATH_INFO` segment of a URL as a sequence of path segments. For example, the `PATH_INFO` string `/a/b/c` is converted to the sequence `['a', 'b', 'c']`.

This path sequence is then used to descend through the *resource tree*, looking up a resource for each path segment. Each lookup uses the `__getitem__` method of a resource in the tree.

For example, if the path info sequence is `['a', 'b', 'c']`:

- *Traversal* starts by acquiring the *root* resource of the application by calling the *root factory*. The *root factory* can be configured to return whatever object is appropriate as the traversal root of your application.
- Next, the first element (`'a'`) is popped from the path segment sequence and is used as a key to lookup the corresponding resource in the root. This invokes the root resource's `__getitem__` method using that value (`'a'`) as an argument.
- If the root resource “contains” a resource with key `'a'`, its `__getitem__` method will return it. The *context* temporarily becomes the “A” resource.
- The next segment (`'b'`) is popped from the path sequence, and the “A” resource's `__getitem__` is called with that value (`'b'`) as an argument; we'll presume it succeeds.
- The “A” resource's `__getitem__` returns another resource, which we'll call “B”. The *context* temporarily becomes the “B” resource.

Traversal continues until the path segment sequence is exhausted or a path element cannot be resolved to a resource. In either case, the *context* resource is the last object that the traversal successfully resolved. If any resource found during traversal lacks a `__getitem__` method, or if its `__getitem__` method raises a `KeyError`, traversal ends immediately, and that resource becomes the *context*.

The results of a *traversal* also include a *view name*. If traversal ends before the path segment sequence is exhausted, the *view name* is the *next* remaining path segment element. If the *traversal* expends all of the path segments, then the *view name* is the empty string (`''`).

The combination of the context resource and the *view name* found via traversal is used later in the same request by the *view lookup* subsystem to find a *view callable*. How Pyramid performs view lookup is explained within the *View Configuration* chapter.

The Resource Tree

The resource tree is a set of nested dictionary-like resource objects that begins with a *root* resource. In order to use *traversal* to resolve URLs to code, your application must supply a *resource tree* to Pyramid.

In order to supply a root resource for an application the Pyramid *Router* is configured with a call-back known as a *root factory*. The root factory is supplied by the application at startup time as the `root_factory` argument to the *Configurator*.

The root factory is a Python callable that accepts a *request* object, and returns the root object of the *resource tree*. A function or class is typically used as an application's root factory. Here's an example of a simple root factory class:

```
1 class Root(dict):
2     def __init__(self, request):
3         pass
```


Here's an example of using this root factory within startup configuration, by passing it to an instance of a *Configurator* named `config`:

```
1 config = Configurator(root_factory=Root)
```

The `root_factory` argument to the *Configurator* constructor registers this root factory to be called to generate a root resource whenever a request enters the application. The root factory registered this way is also known as the global root factory. A root factory can alternatively be passed to the *Configurator* as a *dotted Python name* which can refer to a root factory defined in a different module.

If no *root factory* is passed to the Pyramid *Configurator* constructor, or if the `root_factory` value specified is `None`, a *default root factory* is used. The default root factory always returns a resource that has no child resources; it is effectively empty.

Usually a root factory for a traversal-based application will be more complicated than the above `Root` class. In particular it may be associated with a database connection or another persistence mechanism. The above `Root` class is analogous to the default root factory present in Pyramid. The default root factory is very simple and not very useful.

 If the items contained within the resource tree are “persistent” (they have state that lasts longer than the execution of a single process), they become analogous to the concept of *domain model* objects used by many other frameworks.

The resource tree consists of *container* resources and *leaf* resources. There is only one difference between a *container* resource and a *leaf* resource: *container* resources possess a `__getitem__` method (making it “dictionary-like”) while *leaf* resources do not. The `__getitem__` method was chosen as the signifying difference between the two types of resources because the presence of this method is how Python itself typically determines whether an object is “containerish” or not (dictionary objects are “containerish”).

Each container resource is presumed to be willing to return a child resource or raise a `KeyError` based on a name passed to its `__getitem__`.

Leaf-level instances must not have a `__getitem__`. If instances that you’d like to be leaves already happen to have a `__getitem__` through some historical inequity, you should subclass these resource types and cause their `__getitem__` methods to simply raise a `KeyError`. Or just disuse them and think up another strategy.

Usually the traversal root is a *container* resource, and as such it contains other resources. However, it doesn’t *need* to be a container. Your resource tree can be as shallow or as deep as you require.

In general, the resource tree is traversed beginning at its root resource using a sequence of path elements described by the `PATH_INFO` of the current request. If there are path segments, the root resource’s `__getitem__` is called with the next path segment, and it is expected to return another resource. The resulting resource’s `__getitem__` is called with the very next path segment, and it is expected to return another resource. This happens *ad infinitum* until all path segments are exhausted.

The Traversal Algorithm

This section will attempt to explain the Pyramid traversal algorithm. We’ll provide a description of the algorithm, a diagram of how the algorithm works, and some example traversal scenarios that might help you understand how the algorithm operates against a specific resource tree.

We’ll also talk a bit about *view lookup*. The *View Configuration* chapter discusses *view lookup* in detail, and it is the canonical source for information about views. Technically, *view lookup* is a Pyramid subsystem that is separated from traversal entirely. However, we’ll describe the fundamental behavior of view lookup in the examples in the next few sections to give you an idea of how traversal and view lookup cooperate, because they are almost always used together.

A Description of the Traversal Algorithm

When a user requests a page from your traversal-powered application, the system uses this algorithm to find a *context* resource and a *view name*.

1. The request for the page is presented to the Pyramid *router* in terms of a standard *WSGI* request, which is represented by a *WSGI* environment and a *WSGI* `start_response` callable.
2. The router creates a *request* object based on the *WSGI* environment.
3. The *root factory* is called with the *request*. It returns a *root* resource.
4. The router uses the *WSGI* environment's `PATH_INFO` information to determine the path segments to traverse. The leading slash is stripped off `PATH_INFO`, and the remaining path segments are split on the slash character to form a traversal sequence.

The traversal algorithm by default attempts to first URL-unquote and then Unicode-decode each path segment derived from `PATH_INFO` from its natural byte string (`str` type) representation. URL unquoting is performed using the Python standard library `urllib.unquote` function. Conversion from a URL-decoded string into Unicode is attempted using the UTF-8 encoding. If any URL-unquoted path segment in `PATH_INFO` is not decodable using the UTF-8 decoding, a `TypeError` is raised. A segment will be fully URL-unquoted and UTF8-decoded before it is passed in to the `__getitem__` of any resource during traversal.

Thus a request with a `PATH_INFO` variable of `/a/b/c` maps to the traversal sequence `[u'a', u'b', u'c']`.

5. *Traversal* begins at the root resource returned by the root factory. For the traversal sequence `[u'a', u'b', u'c']`, the root resource's `__getitem__` is called with the name `'a'`. Traversal continues through the sequence. In our example, if the root resource's `__getitem__` called with the name `a` returns a resource (a.k.a. resource “A”), that resource's `__getitem__` is called with the name `'b'`. If resource “A” returns a resource “B” when asked for `'b'`, resource B's `__getitem__` is then asked for the name `'c'`, and may return resource “C”.
6. Traversal ends when either (a) the entire path is exhausted, (b) when any resource raises a `KeyError` from its `__getitem__`, (c) when any non-final path element traversal does not have a `__getitem__` method (resulting in an `AttributeError`), or (d) when any path element is prefixed with the set of characters `@@` (indicating that the characters following the `@@` token should be treated as a *view name*).
7. When traversal ends for any of the reasons in the previous step, the last resource found during traversal is deemed to be the *context*. If the path has been exhausted when traversal ends, the *view name* is deemed to be the empty string (`' '`). However, if the path was *not* exhausted before traversal terminated, the first remaining path segment is treated as the view name.
8. Any subsequent path elements after the *view name* is found are deemed the *subpath*. The subpath is always a sequence of path segments that come from `PATH_INFO` that are “left over” after traversal has completed.

Once the *context* resource, the *view name*, and associated attributes such as the *subpath* are located, the job of *traversal* is finished. It passes back the information it obtained to its caller, the *Pyramid Router*, which subsequently invokes *view lookup* with the context and view name information.

The traversal algorithm exposes two special cases:

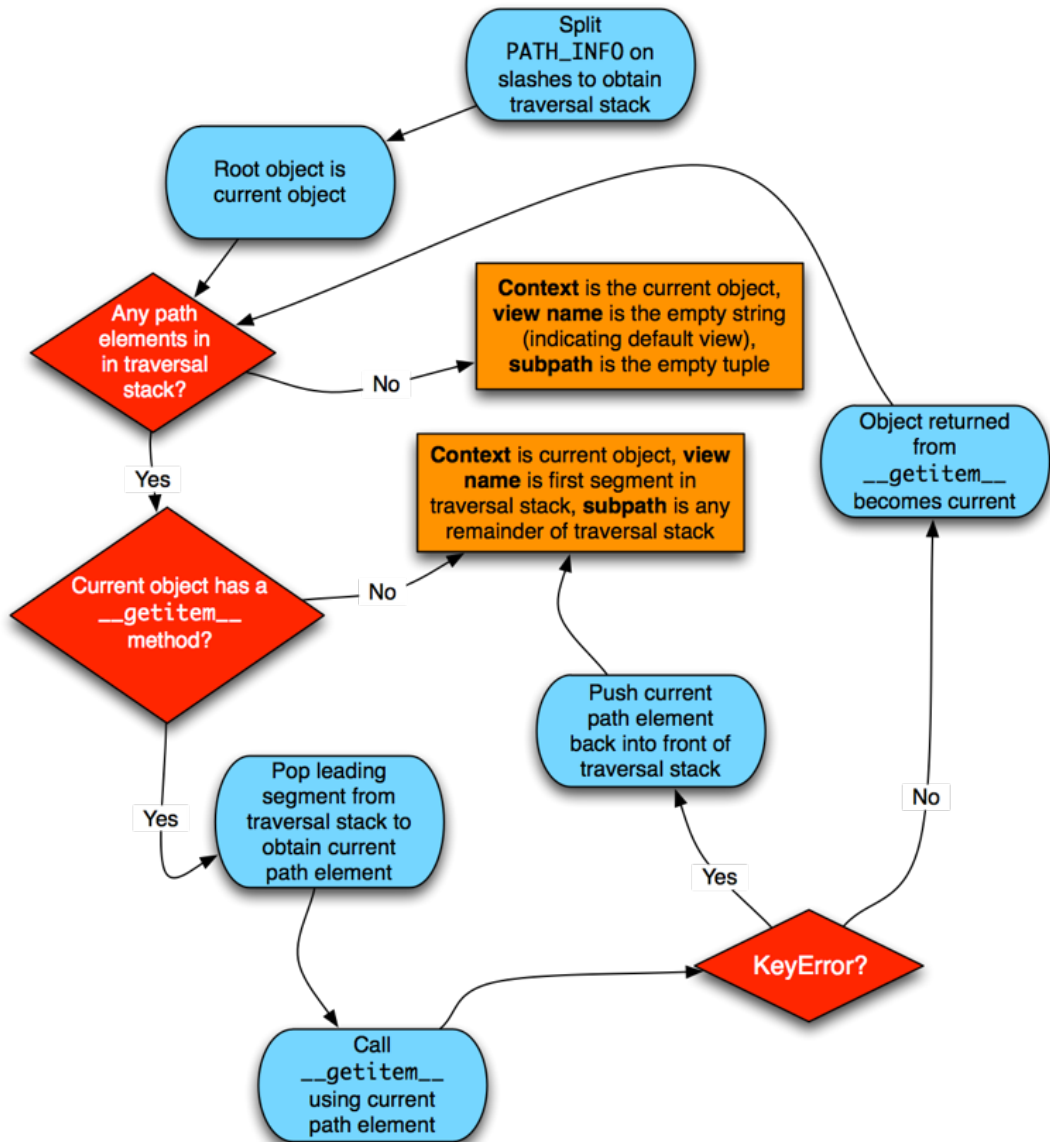
- You will often end up with a *view name* that is the empty string as the result of a particular traversal. This indicates that the view lookup machinery should lookup the *default view*. The default view is a view that is registered with no name or a view which is registered with a name that equals the empty string.
- If any path segment element begins with the special characters @@ (think of them as goggles), the value of that segment minus the goggle characters is considered the *view name* immediately and traversal stops there. This allows you to address views that may have the same names as resource names in the tree unambiguously.

Finally, traversal is responsible for locating a *virtual root*. A virtual root is used during “virtual hosting”. See the *Virtual Hosting* chapter for information. We won’t speak more about it in this chapter.



Pyramid™

Model Graph Traversal



Traversal Algorithm Examples

No one can be expected to understand the traversal algorithm by analogy and description alone, so let's examine some traversal scenarios that use concrete URLs and resource tree compositions.

Let's pretend the user asks for `http://example.com/foo/bar/baz/biz/buz.txt`. The request's `PATH_INFO` in that case is `/foo/bar/baz/biz/buz.txt`. Let's further pretend that when this request comes in, we're traversing the following resource tree:

```
/--  
 |  
 |-- foo  
    |  
    ----bar
```

Here's what happens:

- *traversal* traverses the root, and attempts to find “foo”, which it finds.
- *traversal* traverses “foo”, and attempts to find “bar”, which it finds.
- *traversal* traverses “bar”, and attempts to find “baz”, which it does not find (the “bar” resource raises a `KeyError` when asked for “baz”).

The fact that it does not find “baz” at this point does not signify an error condition. It signifies the following:

- The *context* is the “bar” resource (the context is the last resource found during traversal).
- The *view name* is `baz`.
- The *subpath* is `('biz', 'buz.txt')`.

At this point, traversal has ended, and *view lookup* begins.

Because it's the “context” resource, the view lookup machinery examines “bar” to find out what “type” it is. Let's say it finds that the context is a `Bar` type (because “bar” happens to be an instance of the class `Bar`). Using the *view name* (`baz`) and the type, view lookup asks the *application registry* this question:

- Please find me a *view callable* registered using a *view configuration* with the name “baz” that can be used for the class `Bar`.

Let's say that view lookup finds no matching view type. In this circumstance, the Pyramid *router* returns the result of the *Not Found View* and the request ends.

However, for this tree:

```

/--
|
|-- foo
|
|----bar
|
|----baz
|
|----biz

```

The user asks for `http://example.com/foo/bar/baz/biz/buz.txt`

- *traversal* traverses “foo”, and attempts to find “bar”, which it finds.
- *traversal* traverses “bar”, and attempts to find “baz”, which it finds.
- *traversal* traverses “baz”, and attempts to find “biz”, which it finds.
- *traversal* traverses “biz”, and attempts to find “buz.txt”, which it does not find.

The fact that it does not find a resource related to “buz.txt” at this point does not signify an error condition. It signifies the following:

- The *context* is the “biz” resource (the context is the last resource found during traversal).
- The *view name* is “buz.txt”.
- The *subpath* is an empty sequence (`()`).

At this point, traversal has ended, and *view lookup* begins.

Because it’s the “context” resource, the view lookup machinery examines the “biz” resource to find out what “type” it is. Let’s say it finds that the resource is a `Biz` type (because “biz” is an instance of the Python class `Biz`). Using the *view name* (`buz.txt`) and the type, view lookup asks the *application registry* this question:

- Please find me a *view callable* registered with a *view configuration* with the name `buz.txt` that can be used for class `Biz`.

Let’s say that question is answered by the application registry. In such a situation, the application registry returns a *view callable*. The view callable is then called with the current *WebOb request* as the sole argument, `request`. It is expected to return a response.

The Example View Callables Accept Only a Request; How Do I Access the Context Resource?

Most of the examples in this documentation assume that a view callable is typically passed only a *request* object. Sometimes your view callables need access to the *context* resource, especially when you use *traversal*. You might use a supported alternative view callable argument list in your view callables such as the `(context, request)` calling convention described in *Alternate View Callable Argument/Calling Conventions*. But you don't need to if you don't want to. In view callables that accept only a request, the *context* resource found by traversal is available as the `context` attribute of the request object, e.g., `request.context`. The *view name* is available as the `view_name` attribute of the request object, e.g., `request.view_name`. Other Pyramid-specific request attributes are also available as described in *Special Attributes Added to the Request by Pyramid*.

Using Resource Interfaces in View Configuration

Instead of registering your views with a `context` that names a Python resource *class*, you can optionally register a view callable with a `context` which is an *interface*. An interface can be attached arbitrarily to any resource object. View lookup treats context interfaces specially, and therefore the identity of a resource can be divorced from that of the class which implements it. As a result, associating a view with an interface can provide more flexibility for sharing a single view between two or more different implementations of a resource type. For example, if two resource objects of different Python class types share the same interface, you can use the same view configuration to specify both of them as a `context`.

In order to make use of interfaces in your application during view dispatch, you must create an interface and mark up your resource classes or instances with interface declarations that refer to this interface.

To attach an interface to a resource *class*, you define the interface and use the `zope.interface.implementer()` class decorator to associate the interface with the class.

```
1 from zope.interface import Interface
2 from zope.interface import implementer
3
4 class IHello(Interface):
5     """ A marker interface """
6
7 @implementer(IHello)
8 class Hello(object):
9     pass
```

To attach an interface to a resource *instance*, you define the interface and use the `zope.interface.alsoProvides()` function to associate the interface with the instance. This function mutates the instance in such a way that the interface is attached to it.

```

1 from zope.interface import Interface
2 from zope.interface import alsoProvides
3
4 class IHello(Interface):
5     """ A marker interface """
6
7 class Hello(object):
8     pass
9
10 def make_hello():
11     hello = Hello()
12     alsoProvides(hello, IHello)
13     return hello

```

Regardless of how you associate an interface—with either a resource instance or a resource class—the resulting code to associate that interface with a view callable is the same. Assuming the above code that defines an `IHello` interface lives in the root of your application, and its module is named “resources.py”, the interface declaration below will associate the `mypackage.views.hello_world` view with resources that implement, or provide, this interface.

```

1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('mypackage.views.hello_world', name='hello.html',
4               context='mypackage.resources.IHello')

```

Any time a resource that is determined to be the *context* provides this interface, and a view named `hello.html` is looked up against it as per the URL, the `mypackage.views.hello_world` view callable will be invoked.

Note, in cases where a view is registered against a resource class, and a view is also registered against an interface that the resource class implements, an ambiguity arises. Views registered for the resource class take precedence over any views registered for any interface the resource class implements. Thus, if one view configuration names a *context* of both the class type of a resource, and another view configuration names a *context* of interface implemented by the resource’s class, and both view configurations are otherwise identical, the view registered for the context’s class will “win”.

For more information about defining resources with interfaces for use within view configuration, see *Resources Which Implement Interfaces*.

References

A tutorial showing how *traversal* can be used within a Pyramid application exists in *ZODB + Traversal Wiki Tutorial*.

See the *View Configuration* chapter for detailed information about *view lookup*.

The `pyramid.traversal` module contains API functions that deal with traversal, such as traversal invocation from within application code.

The `pyramid.request.Request.resource_url()` method generates a URL when given a resource retrieved from a resource tree.

Security

Pyramid provides an optional, declarative, security system. Security in Pyramid is separated into authentication and authorization. The two systems communicate via *principal* identifiers. Authentication is merely the mechanism by which credentials provided in the *request* are resolved to one or more *principal* identifiers. These identifiers represent the users and groups that are in effect during the request. Authorization then determines access based on the *principal* identifiers, the requested *permission*, and a *context*.

The Pyramid authorization system can prevent a *view* from being invoked based on an *authorization policy*. Before a view is invoked, the authorization system can use the credentials in the *request* along with the *context* resource to determine if access will be allowed. Here's how it works at a high level:

- A user may or may not have previously visited the application and supplied authentication credentials, including a *userid*. If so, the application may have called `pyramid.security.remember()` to remember these.
- A *request* is generated when a user visits the application.
- Based on the request, a *context* resource is located through *resource location*. A context is located differently depending on whether the application uses *traversal* or *URL dispatch*, but a context is ultimately found in either case. See the *URL Dispatch* chapter for more information.
- A *view callable* is located by *view lookup* using the context as well as other attributes of the request.
- If an *authentication policy* is in effect, it is passed the request. It will return some number of *principal* identifiers. To do this, the policy would need to determine the authenticated *userid* present in the request.
- If an *authorization policy* is in effect and the *view configuration* associated with the view callable that was found has a *permission* associated with it, the authorization policy is passed the *context*, some number of *principal* identifiers returned by the authentication policy, and the *permission* associated with the view; it will allow or deny access.
- If the authorization policy allows access, the view callable is invoked.
- If the authorization policy denies access, the view callable is not invoked. Instead the *forbidden view* is invoked.

Authorization is enabled by modifying your application to include an *authentication policy* and *authorization policy*. Pyramid comes with a variety of implementations of these policies. To provide maximal flexibility, Pyramid also allows you to create custom authentication policies and authorization policies.

Enabling an Authorization Policy

Pyramid does not enable any authorization policy by default. All views are accessible by completely anonymous users. In order to begin protecting views from execution based on security settings, you need to enable an authorization policy.


Enabling an Authorization Policy Imperatively

Use the `set_authorization_policy()` method of the `Configurator` to enable an authorization policy.

You must also enable an *authentication policy* in order to enable the authorization policy. This is because authorization, in general, depends upon authentication. Use the `set_authentication_policy()` method during application setup to specify the authentication policy.

For example:

```
1 from pyramid.config import Configurator
2 from pyramid.authentication import AuthTktAuthenticationPolicy
3 from pyramid.authorization import ACLAuthorizationPolicy
4 authn_policy = AuthTktAuthenticationPolicy('seekrit', hashalg='sha512')
5 authz_policy = ACLAuthorizationPolicy()
6 config = Configurator()
7 config.set_authentication_policy(authn_policy)
8 config.set_authorization_policy(authz_policy)
```

 The `authentication_policy` and `authorization_policy` arguments may also be passed to their respective methods mentioned above as *dotted Python name* values, each representing the dotted name path to a suitable implementation global defined at Python module scope.

The above configuration enables a policy which compares the value of an “auth ticket” cookie passed in the request’s environment which contains a reference to a single *userid*, and matches that *userid*’s *principals* against the principals present in any *ACL* found in the resource tree when attempting to call some *view*.

While it is possible to mix and match different authentication and authorization policies, it is an error to configure a Pyramid application with an authentication policy but without the authorization policy or vice versa. If you do this, you’ll receive an error at application startup time.

See also:

See also the `pyramid.authorization` and `pyramid.authentication` modules for alternative implementations of authorization and authentication policies.

Protecting Views with Permissions

To protect a *view callable* from invocation based on a user’s security settings when a particular type of resource becomes the *context*, you must pass a *permission* to *view configuration*. Permissions are usually just strings, and they have no required composition: you can name permissions whatever you like.

For example, the following view declaration protects the view named `add_entry.html` when the context resource is of type `Blog` with the `add` permission using the `pyramid.config.Configurator.add_view()` API:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('mypackage.views.blog_entry_add_view',
4                 name='add_entry.html',
5                 context='mypackage.resources.Blog',
6                 permission='add')
```

The equivalent view registration including the `add` permission name may be performed via the `@view_config` decorator:

```
1 from pyramid.view import view_config
2 from resources import Blog
3
4 @view_config(context=Blog, name='add_entry.html', permission='add')
5 def blog_entry_add_view(request):
6     """ Add blog entry code goes here """
7     pass
```

As a result of any of these various view configuration statements, if an authorization policy is in place when the view callable is found during normal application operations, the requesting user will need to possess the `add` permission against the *context* resource in order to be able to invoke the `blog_entry_add_view` view. If they do not, the *Forbidden* view will be invoked.

Setting a Default Permission

If a permission is not supplied to a view configuration, the registered view will always be executable by entirely anonymous users: any authorization policy in effect is ignored.

In support of making it easier to configure applications which are “secure by default”, Pyramid allows you to configure a *default* permission. If supplied, the default permission is used as the permission string to all view registrations which don’t otherwise name a `permission` argument.

The `pyramid.config.Configurator.set_default_permission()` method supports configuring a default permission for an application.

When a default permission is registered:

- If a view configuration names an explicit permission, the default permission is ignored for that view registration, and the view-configuration-named permission is used.
- If a view configuration names the permission `pyramid.security.NO_PERMISSION_REQUIRED`, the default permission is ignored, and the view is registered *without* a permission (making it available to all callers regardless of their credentials).



When you register a default permission, *all* views (even *exception view* views) are protected by a permission. For all views which are truly meant to be anonymously accessible, you will need to associate the view's configuration with the `pyramid.security.NO_PERMISSION_REQUIRED` permission.

Assigning ACLs to Your Resource Objects

When the default Pyramid *authorization policy* determines whether a user possesses a particular permission with respect to a resource, it examines the *ACL* associated with the resource. An ACL is associated with a resource by adding an `__acl__` attribute to the resource object. This attribute can be defined on the resource *instance* if you need instance-level security, or it can be defined on the resource *class* if you just need type-level security.

For example, an ACL might be attached to the resource for a blog via its class:

```
1 from pyramid.security import Allow
2 from pyramid.security import Everyone
3
4 class Blog(object):
5     __acl__ = [
6         (Allow, Everyone, 'view'),
7         (Allow, 'group:editors', 'add'),
8         (Allow, 'group:editors', 'edit'),
9     ]
```

Or, if your resources are persistent, an ACL might be specified via the `__acl__` attribute of an *instance* of a resource:

```
1 from pyramid.security import Allow
2 from pyramid.security import Everyone
3
4 class Blog(object):
5     pass
6
7 blog = Blog()
8
9 blog.__acl__ = [
10     (Allow, Everyone, 'view'),
11     (Allow, 'group:editors', 'add'),
12     (Allow, 'group:editors', 'edit'),
13 ]
```

Whether an ACL is attached to a resource's class or an instance of the resource itself, the effect is the same. It is useful to decorate individual resource instances with an ACL (as opposed to just decorating their class) in applications such as content management systems where fine-grained access is required on an object-by-object basis.

Dynamic ACLs are also possible by turning the ACL into a callable on the resource. This may allow the ACL to dynamically generate rules based on properties of the instance.

```
1 from pyramid.security import Allow
2 from pyramid.security import Everyone
3
4 class Blog(object):
5     def __acl__(self):
6         return [
7             (Allow, Everyone, 'view'),
8             (Allow, self.owner, 'edit'),
9             (Allow, 'group:editors', 'edit'),
10         ]
11
12     def __init__(self, owner):
13         self.owner = owner
```



Writing `__acl__` as properties is discouraged because an `AttributeError` occurring in `gget` or `fset` will be silently dismissed (this is consistent with Python `getattr` and `hasattr` behaviors). For dynamic ACLs, simply use callables, as documented above.

Elements of an ACL

Here's an example ACL:

```
1 from pyramid.security import Allow
2 from pyramid.security import Everyone
3
4 __acl__ = [
5     (Allow, Everyone, 'view'),
6     (Allow, 'group:editors', 'add'),
7     (Allow, 'group:editors', 'edit'),
8 ]
```

The example ACL indicates that the `pyramid.security.Everyone` principal—a special system-defined principal indicating, literally, everyone—is allowed to view the blog, and the `group:editors` principal is allowed to add to and edit the blog.

Each element of an ACL is an *ACE*, or access control entry. For example, in the above code block, there are three ACEs: `(Allow, Everyone, 'view')`, `(Allow, 'group:editors', 'add')`, and `(Allow, 'group:editors', 'edit')`.

The first element of any ACE is either `pyramid.security.Allow`, or `pyramid.security.Deny`, representing the action to take when the ACE matches. The second element is a *principal*. The third argument is a permission or sequence of permission names.

A principal is usually a user id, however it also may be a group id if your authentication system provides group information and the effective *authentication policy* policy is written to respect group information. See *Extending Default Authentication Policies*.

Each ACE in an ACL is processed by an authorization policy *in the order dictated by the ACL*. So if you have an ACL like this:

```
1 from pyramid.security import Allow
2 from pyramid.security import Deny
3 from pyramid.security import Everyone
4
5 __acl__ = [
6     (Allow, Everyone, 'view'),
7     (Deny, Everyone, 'view'),
8 ]
```

The default authorization policy will *allow* everyone the view permission, even though later in the ACL you have an ACE that denies everyone the view permission. On the other hand, if you have an ACL like this:

```
1 from pyramid.security import Everyone
2 from pyramid.security import Allow
3 from pyramid.security import Deny
4
5 __acl__ = [
6     (Deny, Everyone, 'view'),
7     (Allow, Everyone, 'view'),
8 ]
```

The authorization policy will deny everyone the view permission, even though later in the ACL, there is an ACE that allows everyone.

The third argument in an ACE can also be a sequence of permission names instead of a single permission name. So instead of creating multiple ACEs representing a number of different permission grants to a single group: `editors` group, we can collapse this into a single ACE, as below.

```
1 from pyramid.security import Allow
2 from pyramid.security import Everyone
3
4 __acl__ = [
5     (Allow, Everyone, 'view'),
6     (Allow, 'group:editors', ('add', 'edit')),
7 ]
```

Special Principal Names

Special principal names exist in the `pyramid.security` module. They can be imported for use in your own code to populate ACLs, e.g., `pyramid.security.Everyone`.

pyramid.security.Everyone

Literally, everyone, no matter what. This object is actually a string under the hood (`system.Everyone`). Every user *is* the principal named “Everyone” during every request, even if a security policy is not in use.

pyramid.security.Authenticated

Any user with credentials as determined by the current security policy. You might think of it as any user that is “logged in”. This object is actually a string under the hood (`system.Authenticated`).

Special Permissions

Special permission names exist in the `pyramid.security` module. These can be imported for use in ACLs. `pyramid.security.ALL_PERMISSIONS`

An object representing, literally, *all* permissions. Useful in an ACL like so: `(Allow, 'fred', ALL_PERMISSIONS)`. The `ALL_PERMISSIONS` object is actually a stand-in object that has a `__contains__` method that always returns `True`, which, for all known authorization policies, has the effect of indicating that a given principal has any permission asked for by the system.

Special ACEs

A convenience *ACE* is defined representing a deny to everyone of all permissions in `pyramid.security.DENY_ALL`. This ACE is often used as the *last* ACE of an ACL to explicitly cause inheriting authorization policies to “stop looking up the traversal tree” (effectively breaking any inheritance). For example, an ACL which allows *only* `fred` the view permission for a particular resource, despite what inherited ACLs may say when the default authorization policy is in effect, might look like so:

```
1 from pyramid.security import Allow
2 from pyramid.security import DENY_ALL
3
4 __acl__ = [ (Allow, 'fred', 'view'), DENY_ALL ]
```

Under the hood, the `pyramid.security.DENY_ALL` ACE equals the following:

```
1 from pyramid.security import ALL_PERMISSIONS
2 __acl__ = [ (Deny, Everyone, ALL_PERMISSIONS) ]
```

ACL Inheritance and Location-Awareness

While the default *authorization policy* is in place, if a resource object does not have an ACL when it is the context, its *parent* is consulted for an ACL. If that object does not have an ACL, *its* parent is consulted for an ACL, ad infinitum, until we’ve reached the root and there are no more parents left.

In order to allow the security machinery to perform ACL inheritance, resource objects must provide *location-awareness*. Providing *location-awareness* means two things: the root object in the resource tree must have a `__name__` attribute and a `__parent__` attribute.

```
1 class Blog(object):  
2     __name__ = ''  
3     __parent__ = None
```

An object with a `__parent__` attribute and a `__name__` attribute is said to be *location-aware*. Location-aware objects define a `__parent__` attribute which points at their parent object. The root object's `__parent__` is `None`.

See also:

See also *pyramid.location* for documentations of functions which use location-awareness.

See also:

See also *Location-Aware Resources*.

Changing the Forbidden View

When Pyramid denies a view invocation due to an authorization denial, the special `forbidden` view is invoked. Out of the box, this forbidden view is very plain. See *Changing the Forbidden View* within *Using Hooks* for instructions on how to create a custom forbidden view and arrange for it to be called when view authorization is denied.

Debugging View Authorization Failures

If your application in your judgment is allowing or denying view access inappropriately, start your application under a shell using the `PYRAMID_DEBUG_AUTHORIZATION` environment variable set to 1. For example:

```
$ PYRAMID_DEBUG_AUTHORIZATION=1 $VENV/bin/pserve myproject.ini
```

When any authorization takes place during a top-level view rendering, a message will be logged to the console (to `stderr`) about what ACE in which ACL permitted or denied the authorization based on authentication information.

This behavior can also be turned on in the application `.ini` file by setting the `pyramid.debug_authorization` key to `true` within the application's configuration section, e.g.:

```
1 [app:main]
2 use = egg:MyProject
3 pyramid.debug_authorization = true
```

With this debug flag turned on, the response sent to the browser will also contain security debugging information in its body.

Debugging Imperative Authorization Failures

The `pyramid.request.Request.has_permission()` API is used to check security within view functions imperatively. It returns instances of objects that are effectively booleans. But these objects are not raw `True` or `False` objects, and have information attached to them about why the permission was allowed or denied. The object will be one of `pyramid.security.ACLAllowed`, `pyramid.security.ACLDenied`, `pyramid.security.Allowed`, or `pyramid.security.Denied`, as documented in `pyramid.security`. At the very minimum, these objects will have a `msg` attribute, which is a string indicating why the permission was denied or allowed. Introspecting this information in the debugger or via print statements when a call to `has_permission()` fails is often useful.

Extending Default Authentication Policies

Pyramid ships with some built in authentication policies for use in your applications. See `pyramid.authentication` for the available policies. They differ on their mechanisms for tracking authentication credentials between requests, however they all interface with your application in mostly the same way.

Above you learned about *Assigning ACLs to Your Resource Objects*. Each *principal* used in the *ACL* is matched against the list returned from `pyramid.interfaces.IAuthenticationPolicy.effective_principals()`. Similarly, `pyramid.request.Request.authenticated_userid()` maps to `pyramid.interfaces.IAuthenticationPolicy.authenticated_userid()`.

You may control these values by subclassing the default authentication policies. For example, below we subclass the `pyramid.authentication.AuthTktAuthenticationPolicy` and define extra functionality to query our database before confirming that the *userid* is valid in order to avoid blindly trusting the value in the cookie (what if the cookie is still valid, but the user has deleted their account?). We then use that *userid* to augment the `effective_principals` with information about groups and other state for that user.


```
1 from pyramid.authentication import AuthTktAuthenticationPolicy
2
3 class MyAuthenticationPolicy(AuthTktAuthenticationPolicy):
4     def authenticated_userid(self, request):
5         userid = self.unauthenticated_userid(request)
6         if userid:
7             if request.verify_userid_is_still_valid(userid):
8                 return userid
9
10    def effective_principals(self, request):
11        principals = [Everyone]
12        userid = self.authenticated_userid(request)
13        if userid:
14            principals += [Authenticated, str(userid)]
15        return principals
```

In most instances `authenticated_userid` and `effective_principals` are application-specific, whereas `unauthenticated_userid`, `remember`, and `forget` are generic and focused on transport and serialization of data between consecutive requests.

Creating Your Own Authentication Policy

Pyramid ships with a number of useful out-of-the-box security policies (see `pyramid.authentication`). However, creating your own authentication policy is often necessary when you want to control the “horizontal and vertical” of how your users authenticate. Doing so is a matter of creating an instance of something that implements the following interface:

```
1 class IAuthenticationPolicy(object):
2     """ An object representing a Pyramid authentication policy. """
3
4     def authenticated_userid(self, request):
5         """ Return the authenticated :term:`userid` or ``None`` if
6         no authenticated userid can be found. This method of the
7         policy should ensure that a record exists in whatever
8         persistent store is used related to the user (the user
9         should not have been deleted); if a record associated with
10        the current id does not exist in a persistent store, it
11        should return ``None``.
12
13        """
14
15    def unauthenticated_userid(self, request):
```

```

16     """ Return the *unauthenticated* userid. This method
17     performs the same duty as ``authenticated_userid`` but is
18     permitted to return the userid based only on data present
19     in the request; it needn't (and shouldn't) check any
20     persistent store to ensure that the user record related to
21     the request userid exists.
22
23     This method is intended primarily a helper to assist the
24     ``authenticated_userid`` method in pulling credentials out
25     of the request data, abstracting away the specific headers,
26     query strings, etc that are used to authenticate the request.
27
28     """
29
30     def effective_principals(self, request):
31         """ Return a sequence representing the effective principals
32         typically including the :term:'userid' and any groups belonged
33         to by the current user, always including 'system' groups such
34         as ``pyramid.security.Everyone`` and
35         ``pyramid.security.Authenticated``.
36
37         """
38
39     def remember(self, request, userid, **kw):
40         """ Return a set of headers suitable for 'remembering' the
41         :term:'userid' named ``userid`` when set in a response. An
42         individual authentication policy and its consumers can
43         decide on the composition and meaning of **kw.
44
45         """
46
47     def forget(self, request):
48         """ Return a set of headers suitable for 'forgetting' the
49         current user on subsequent requests.
50
51         """

```

After you do so, you can pass an instance of such a class into the `set_authentication_policy` method at configuration time to use it.

Creating Your Own Authorization Policy

An authorization policy is a policy that allows or denies access after a user has been authenticated. Most Pyramid applications will use the default `pyramid.authorization.ACLAuthorizationPolicy`.

However, in some cases, it's useful to be able to use a different authorization policy than the default *ACLAuthorizationPolicy*. For example, it might be desirable to construct an alternate authorization policy which allows the application to use an authorization mechanism that does not involve *ACL* objects.

Pyramid ships with only a single default authorization policy, so you'll need to create your own if you'd like to use a different one. Creating and using your own authorization policy is a matter of creating an instance of an object that implements the following interface:

```
1 class IAuthorizationPolicy(object):
2     """ An object representing a Pyramid authorization policy. """
3     def permits(self, context, principals, permission):
4         """ Return ``True`` if any of the ``principals`` is allowed the
5             ``permission`` in the current ``context``, else return ``False``
6         """
7
8     def principals_allowed_by_permission(self, context, permission):
9         """ Return a set of principal identifiers allowed by the
10            ``permission`` in ``context``. This behavior is optional; if you
11            choose to not implement it you should define this method as
12            something which raises a ``NotImplementedError``. This method
13            will only be called when the
14            ``pyramid.security.principals_allowed_by_permission`` API is
15            used. """
```

After you do so, you can pass an instance of such a class into the *set_authorization_policy* method at configuration time to use it.

Admonishment Against Secret-Sharing

A “secret” is required by various components of Pyramid. For example, the *authentication policy* below uses a secret value *seekrit*:

```
authn_policy = AuthTktAuthenticationPolicy('seekrit', hashalg='sha512')
```

A *session factory* also requires a secret:

```
my_session_factory = SignedCookieSessionFactory('itsaseekreet')
```

It is tempting to use the same secret for multiple Pyramid subsystems. For example, you might be tempted to use the value `seekrit` as the secret for both the authentication policy and the session factory defined above. This is a bad idea, because in both cases, these secrets are used to sign the payload of the data.

If you use the same secret for two different parts of your application for signing purposes, it may allow an attacker to get his chosen plaintext signed, which would allow the attacker to control the content of the payload. Re-using a secret across two different subsystems might drop the security of signing to zero. Keys should not be re-used across different contexts where an attacker has the possibility of providing a chosen plaintext.

Combining Traversal and URL Dispatch

When you write most Pyramid applications, you'll be using one or the other of two available *resource location* subsystems: traversal or URL dispatch. However, to solve a limited set of problems, it's useful to use *both* traversal and URL dispatch together within the same application. Pyramid makes this possible via *hybrid* applications.



Reasoning about the behavior of a “hybrid” URL dispatch + traversal application can be challenging. To successfully reason about using URL dispatch and traversal together, you need to understand URL pattern matching, root factories, and the *traversal* algorithm, and the potential interactions between them. Therefore, we don't recommend creating an application that relies on hybrid behavior unless you must.

A Review of Non-Hybrid Applications

When used according to the tutorials in its documentation, Pyramid is a “dual-mode” framework: the tutorials explain how to create an application in terms of using either *URL dispatch* or *traversal*. This chapter details how you might combine these two dispatch mechanisms, but we'll review how they work in isolation before trying to combine them.

URL Dispatch Only

An application that uses *URL dispatch* exclusively to map URLs to code will often have statements like this within its application startup configuration:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_route('foobar', '{foo}/{bar}')
4 config.add_route('bazbuz', '{baz}/{buz}')
5
6 config.add_view('myproject.views.foobar', route_name='foobar')
7 config.add_view('myproject.views.bazbuz', route_name='bazbuz')
```

Each *route* corresponds to one or more view callables. Each view callable is associated with a route by passing a `route_name` parameter that matches its name during a call to `add_view()`. When a route is matched during a request, *view lookup* is used to match the request to its associated view callable. The presence of calls to `add_route()` signify that an application is using URL dispatch.

Traversal Only

An application that uses only traversal will have view configuration declarations that look like this:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('mypackage.views.foobar', name='foobar')
4 config.add_view('mypackage.views.bazbuz', name='bazbuz')
```

When the above configuration is applied to an application, the `mypackage.views.foobar` view callable above will be called when the URL `/foobar` is visited. Likewise, the view `mypackage.views.bazbuz` will be called when the URL `/bazbuz` is visited.

Typically, an application that uses traversal exclusively won't perform any calls to `pyramid.config.Configurator.add_route()` in its startup code.

Hybrid Applications

Either traversal or URL dispatch alone can be used to create a Pyramid application. However, it is also possible to combine the concepts of traversal and URL dispatch when building an application, the result of which is a hybrid application. In a hybrid application, traversal is performed *after* a particular route has matched.

A hybrid application is a lot more like a “pure” traversal-based application than it is like a “pure” URL-dispatch based application. But unlike in a “pure” traversal-based application, in a hybrid application

traversal is performed during a request after a route has already matched. This means that the URL pattern that represents the `pattern` argument of a route must match the `PATH_INFO` of a request, and after the route pattern has matched, most of the “normal” rules of traversal with respect to *resource location* and *view lookup* apply.

There are only four real differences between a purely traversal-based application and a hybrid application:

- In a purely traversal-based application, no routes are defined. In a hybrid application, at least one route will be defined.
- In a purely traversal-based application, the root object used is global, implied by the *root factory* provided at startup time. In a hybrid application, the *root* object at which traversal begins may be varied on a per-route basis.
- In a purely traversal-based application, the `PATH_INFO` of the underlying *WSGI* environment is used wholesale as a traversal path. In a hybrid application, the traversal path is not the entire `PATH_INFO` string, but a portion of the URL determined by a matching pattern in the matched route configuration’s pattern.
- In a purely traversal-based application, view configurations which do not mention a `route_name` argument are considered during *view lookup*. In a hybrid application, when a route is matched, only view configurations which mention that route’s name as a `route_name` are considered during *view lookup*.

More generally, a hybrid application *is* a traversal-based application except:

- the traversal *root* is chosen based on the route configuration of the route that matched, instead of from the `root_factory` supplied during application startup configuration.
- the traversal *path* is chosen based on the route configuration of the route that matched, rather than from the `PATH_INFO` of a request.
- the set of views that may be chosen during *view lookup* when a route matches are limited to those which specifically name a `route_name` in their configuration that is the same as the matched route’s name.

To create a hybrid mode application, use a *route configuration* that implies a particular *root factory* and which also includes a `pattern` argument that contains a special dynamic part: either `*traverse` or `*subpath`.

The Root Object for a Route Match

A hybrid application implies that traversal is performed during a request after a route has matched. Traversal, by definition, must always begin at a root object. Therefore it's important to know *which* root object will be traversed after a route has matched.

Figuring out which *root* object results from a particular route match is straightforward. When a route is matched:

- If the route's configuration has a *factory* argument which points to a *root factory* callable, that callable will be called to generate a *root* object.
- If the route's configuration does not have a *factory* argument, the *global root factory* will be called to generate a *root* object. The global root factory is the callable implied by the *root_factory* argument passed to the *Configurator* at application startup time.
- If a *root_factory* argument is not provided to the *Configurator* at startup time, a *default* root factory is used. The default root factory is used to generate a root object.



Root factories related to a route were explained previously within *Route Factories*. Both the global root factory and default root factory were explained previously within *The Resource Tree*.

Using `*traverse` in a Route Pattern

A hybrid application most often implies the inclusion of a route configuration that contains the special token `*traverse` at the end of a route's pattern:

```
1 config.add_route('home', '{foo}/{bar}/*traverse')
```

A `*traverse` token at the end of the pattern in a route's configuration implies a “remainder” *capture* value. When it is used, it will match the remainder of the path segments of the URL. This remainder becomes the path used to perform traversal.



The `*remainder` route pattern syntax is explained in more detail within *Route Pattern Syntax*.

A hybrid mode application relies more heavily on *traversal* to do *resource location* and *view lookup* than most examples indicate within *URL Dispatch*.

Because the pattern of the above route ends with `*traverse`, when this route configuration is matched during a request, Pyramid will attempt to use *traversal* against the *root* object implied by the *root factory* that is implied by the route’s configuration. Since no `root_factory` argument is explicitly specified for this route, this will either be the *global* root factory for the application, or the *default* root factory. Once *traversal* has found a *context* resource, *view lookup* will be invoked in almost exactly the same way it would have been invoked in a “pure” traversal-based application.

Let’s assume there is no *global root factory* configured in this application. The *default root factory* cannot be traversed; it has no useful `__getitem__` method. So we’ll need to associate this route configuration with a custom root factory in order to create a useful hybrid application. To that end, let’s imagine that we’ve created a root factory that looks like so in a module named `routes.py`:

```

1 class Resource(object):
2     def __init__(self, subobjects):
3         self.subobjects = subobjects
4
5     def __getitem__(self, name):
6         return self.subobjects[name]
7
8 root = Resource(
9     {'a': Resource({'b': Resource({'c': Resource({})})})})
10 )
11
12 def root_factory(request):
13     return root

```

Above we’ve defined a (bogus) resource tree that can be traversed, and a `root_factory` function that can be used as part of a particular route configuration statement:

```

1 config.add_route('home', '{foo}/{bar}/*traverse',
2                 factory='mypackage.routes.root_factory')

```

The `factory` above points at the function we’ve defined. It will return an instance of the `Resource` class as a root object whenever this route is matched. Instances of the `Resource` class can be used for tree traversal because they have a `__getitem__` method that does something nominally useful. Since traversal uses `__getitem__` to walk the resources of a resource tree, using traversal against the root resource implied by our route statement is a reasonable thing to do.



We could have also used our `root_factory` function as the `root_factory` argument of the *Configurator* constructor, instead of associating it with a particular route inside the route's configuration. Every hybrid route configuration that is matched, but which does *not* name a `factory` attribute, will use the global `root_factory` function to generate a root object.

When the route configuration named `home` above is matched during a request, the `matchdict` generated will be based on its pattern: `{foo}/{bar}/*traverse`. The “capture value” implied by the `*traverse` element in the pattern will be used to traverse the resource tree in order to find a context resource, starting from the root object returned from the root factory. In the above example, the *root* object found will be the instance named `root` in `routes.py`.

If the URL that matched a route with the pattern `{foo}/{bar}/*traverse` is `http://example.com/one/two/a/b/c`, the traversal path used against the root object will be `a/b/c`. As a result, Pyramid will attempt to traverse through the edges `'a'`, `'b'`, and `'c'`, beginning at the root object.

In our above example, this particular set of traversal steps will mean that the *context* resource of the view would be the `Resource` object we've named `'c'` in our bogus resource tree, and the *view name* resulting from traversal will be the empty string. If you need a refresher about why this outcome is presumed, see *The Traversal Algorithm*.

At this point, a suitable view callable will be found and invoked using *view lookup* as described in *View Configuration*, but with a caveat: in order for view lookup to work, we need to define a view configuration that will match when *view lookup* is invoked after a route matches:

```
1 config.add_route('home', '{foo}/{bar}/*traverse',
2                     factory='mypackage.routes.root_factory')
3 config.add_view('mypackage.views.myview', route_name='home')
```

Note that the above call to `add_view()` includes a `route_name` argument. View configurations that include a `route_name` argument are meant to associate a particular view declaration with a route, using the route's name, in order to indicate that the view should *only be invoked when the route matches*.

Calls to `add_view()` may pass a `route_name` attribute, which refers to the value of an existing route's name argument. In the above example, the route name is `home`, referring to the name of the route defined above it.

The above `mypackage.views.myview` view callable will be invoked when the following conditions are met:

- The route named “home” is matched.

- The *view name* resulting from traversal is the empty string.
- The *context* resource is any object.

It is also possible to declare alternative views that may be invoked when a hybrid route is matched:

```
1 config.add_route('home', '{foo}/{bar}/*traverse',  
2                 factory='mypackage.routes.root_factory')  
3 config.add_view('mypackage.views.myview', route_name='home')  
4 config.add_view('mypackage.views.another_view', route_name='home',  
5                 name='another')
```

The `add_view` call for `mypackage.views.another_view` above names a different view and, more importantly, a different *view name*. The above `mypackage.views.another_view` view will be invoked when the following conditions are met:

- The route named “home” is matched.
- The *view name* resulting from traversal is `another`.
- The *context* resource is any object.

For instance, if the URL `http://example.com/one/two/a/another` is provided to an application that uses the previously mentioned resource tree, the `mypackage.views.another_view` view callable will be called instead of the `mypackage.views.myview` view callable because the *view name* will be `another` instead of the empty string.

More complicated matching can be composed. All arguments to *route* configuration statements and *view* configuration statements are supported in hybrid applications (such as *predicate* arguments).

Using the `traverse` Argument in a Route Definition

Rather than using the `*traverse` remainder marker in a pattern, you can use the `traverse` argument to the `add_route()` method.

When you use the `*traverse` remainder marker, the traversal path is limited to being the remainder segments of a request URL when a route matches. However, when you use the `traverse` argument or attribute, you have more control over how to compose a traversal path.

Here’s a use of the `traverse` pattern in a call to `add_route()`:

```
1 config.add_route('abc', '/articles/{article}/edit',  
2                   traverse='/{article}')
```

The syntax of the `traverse` argument is the same as it is for `pattern`.

If, as above, the pattern provided is `/articles/{article}/edit`, and the `traverse` argument provided is `/ {article}`, when a request comes in that causes the route to match in such a way that the `article` match value is `1` (when the request URI is `/articles/1/edit`), the traversal path will be generated as `/1`. This means that the root object's `__getitem__` will be called with the name `1` during the traversal phase. If the `1` object exists, it will become the *context* of the request. The *Traversal* chapter has more information about traversal.

If the traversal path contains segment marker names which are not present in the pattern argument, a runtime error will occur. The `traverse` pattern should not contain segment markers that do not exist in the path.

Note that the `traverse` argument is ignored when attached to a route that has a `*traverse` remainder marker in its pattern.

Traversal will begin at the root object implied by this route (either the global root, or the object returned by the `factory` associated with this route).

Making Global Views Match

By default, only view configurations that mention a `route_name` will be found during view lookup when a route that has a `*traverse` in its pattern matches. You can allow views without a `route_name` attribute to match a route by adding the `use_global_views` flag to the route definition. For example, the `myproject.views.bazbuz` view below will be found if the route named `abc` below is matched and the `PATH_INFO` is `/abc/bazbuz`, even though the view configuration statement does not have the `route_name="abc"` attribute.

```
1 config.add_route('abc', '/abc/*traverse', use_global_views=True)  
2 config.add_view('myproject.views.bazbuz', name='bazbuz')
```

Using `*subpath` in a Route Pattern

There are certain extremely rare cases when you'd like to influence the traversal *subpath* when a route matches without actually performing traversal. For instance, the `pyramid.wsgi.wsgiapp2()` decorator and the `pyramid.static.static_view` helper attempt to compute `PATH_INFO` from the request's subpath when its `use_subpath` argument is `True`, so it's useful to be able to influence this value.

When `*subpath` exists in a pattern, no path is actually traversed, but the traversal algorithm will return a *subpath* list implied by the capture value of `*subpath`. You'll see this pattern most commonly in route declarations that look like this:

```
1 from pyramid.static import static_view
2
3 www = static_view('mypackage:static', use_subpath=True)
4
5 config.add_route('static', '/static/*subpath')
6 config.add_view(www, route_name='static')
```

`mypackage.views.www` is an instance of `pyramid.static.static_view`. This effectively tells the static helper to traverse everything in the subpath as a filename.

Generating Hybrid URLs

New in version 1.5.

The `pyramid.request.Request.resource_url()` method and the `pyramid.request.Request.resource_path()` method both accept optional keyword arguments that make it easier to generate route-prefixed URLs that contain paths to traversal resources: `route_name`, `route_kw`, and `route_remainder_name`.

Any route that has a pattern that contains a `*remainder` pattern (any stararg remainder pattern, such as `*traverse`, `*subpath`, or `*fred`) can be used as the target name for `request.resource_url(..., route_name=)` and `request.resource_path(..., route_name=)`.

For example, let's imagine you have a route defined in your Pyramid application like so:

```
config.add_route('mysection', '/mysection*traverse')
```

If you'd like to generate the URL `http://example.com/mysection/a/`, you can use the following incantation, assuming that the variable `a` below points to a resource that is a child of the root with a `__name__` of `a`:

```
request.resource_url(a, route_name='mysection')
```

You can generate only the path portion `/mysection/a/` assuming the same:

```
request.resource_path(a, route_name='mysection')
```

The path is virtual host aware, so if the `X-Vhm-Root` environment variable is present in the request, and it's set to `/a`, the above call to `request.resource_url` would generate `http://example.com/mysection/`, and the above call to `request.resource_path` would generate `/mysection/`. See *Virtual Root Support* for more information.

If the route you're trying to use needs simple dynamic part values to be filled in to successfully generate the URL, you can pass these as the `route_kw` argument to `resource_url` and `resource_path`. For example, assuming that the route definition is like so:

```
config.add_route('mysection', '/{id}/mysection*traverse')
```

You can pass `route_kw` in to fill in `{id}` above:

```
request.resource_url(a, route_name='mysection', route_kw={'id':'1'})
```

If you pass `route_kw` but do not pass `route_name`, `route_kw` will be ignored.

By default this feature works by calling `route_url` under the hood, and passing the value of the resource path to that function as `traverse`. If your route has a different `*stararg` remainder name (such as `*subpath`), you can tell `resource_url` or `resource_path` to use that instead of `traverse` by passing `route_remainder_name`. For example, if you have the following route:

```
config.add_route('mysection', '/mysection*subpath')
```

You can fill in the `*subpath` value using `resource_url` by doing:

```
request.resource_path(a, route_name='mysection',  
                     route_remainder_name='subpath')
```

If you pass `route_remainder_name` but do not pass `route_name`, `route_remainder_name` will be ignored.

If you try to use `resource_path` or `resource_url` when the `route_name` argument points at a route that does not have a remainder stararg, an error will not be raised, but the generated URL will not contain any remainder information either.

All other values that are normally passable to `resource_path` and `resource_url` (such as `query`, `anchor`, `host`, `port`, and positional elements) work as you might expect in this configuration.

Note that this feature is incompatible with the `__resource_url__` feature (see *Overriding Resource URL Generation*) implemented on resource objects. Any `__resource_url__` supplied by your resource will be ignored when you pass `route_name`.

Invoking a Subrequest

New in version 1.4.

Pyramid allows you to invoke a subrequest at any point during the processing of a request. Invoking a subrequest allows you to obtain a *response* object from a view callable within your Pyramid application while you're executing a different view callable within the same application.

Here's an example application which uses a subrequest:

```

1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.request import Request
4
5 def view_one(request):
6     subreq = Request.blank('/view_two')
7     response = request.invoke_subrequest(subreq)
8     return response
9
10 def view_two(request):
11     request.response.body = 'This came from view_two'
12     return request.response
13
14 if __name__ == '__main__':
15     config = Configurator()
16     config.add_route('one', '/view_one')
17     config.add_route('two', '/view_two')
18     config.add_view(view_one, route_name='one')
19     config.add_view(view_two, route_name='two')
20     app = config.make_wsgi_app()
21     server = make_server('0.0.0.0', 8080, app)
22     server.serve_forever()

```

When `/view_one` is visited in a browser, the text printed in the browser pane will be `This came from view_two`. The `view_one` view used the `pyramid.request.Request.invoke_subrequest()` API to obtain a response from another view (`view_two`) within the same application when it executed. It did so by constructing a new request that had a URL that it knew would match the `view_two` view registration, and passed that new request along to `pyramid.request.Request.invoke_subrequest()`. The `view_two` view callable was invoked, and it returned a response. The `view_one` view callable then simply returned the response it obtained from the `view_two` view callable.

Note that it doesn't matter if the view callable invoked via a subrequest actually returns a *literal* Response object. Any view callable that uses a renderer or which returns an object that can be interpreted by a response adapter when found and invoked via `pyramid.request.Request.invoke_subrequest()` will return a Response object:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.request import Request
4
5 def view_one(request):
6     subreq = Request.blank('/view_two')
7     response = request.invoke_subrequest(subreq)
8     return response
9
10 def view_two(request):
11     return 'This came from view_two'
12
13 if __name__ == '__main__':
14     config = Configurator()
15     config.add_route('one', '/view_one')
16     config.add_route('two', '/view_two')
17     config.add_view(view_one, route_name='one')
18     config.add_view(view_two, route_name='two', renderer='string')
19     app = config.make_wsgi_app()
20     server = make_server('0.0.0.0', 8080, app)
21     server.serve_forever()
```

Even though the `view_two` view callable returned a string, it was invoked in such a way that the `string` renderer associated with the view registration that was found turned it into a “real” response object for consumption by `view_one`.

Being able to unconditionally obtain a response object by invoking a view callable indirectly is the main advantage to using `pyramid.request.Request.invoke_subrequest()` instead of simply importing a view callable and executing it directly. Note that there's not much advantage to invoking a view

using a subrequest if you *can* invoke a view callable directly. Subrequests are slower and are less convenient if you actually do want just the literal information returned by a function that happens to be a view callable.

Note that, by default, if a view callable invoked by a subrequest raises an exception, the exception will be raised to the caller of `invoke_subrequest()` even if you have a *exception view* configured:

```

1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.request import Request
4
5 def view_one(request):
6     subreq = Request.blank('/view_two')
7     response = request.invoke_subrequest(subreq)
8     return response
9
10 def view_two(request):
11     raise ValueError('foo')
12
13 def excview(request):
14     request.response.body = b'An exception was raised'
15     request.response.status_int = 500
16     return request.response
17
18 if __name__ == '__main__':
19     config = Configurator()
20     config.add_route('one', '/view_one')
21     config.add_route('two', '/view_two')
22     config.add_view(view_one, route_name='one')
23     config.add_view(view_two, route_name='two', renderer='string')
24     config.add_view(excview, context=Exception)
25     app = config.make_wsgi_app()
26     server = make_server('0.0.0.0', 8080, app)
27     server.serve_forever()

```

When we run the above code and visit `/view_one` in a browser, the `excview` *exception view* will *not* be executed. Instead, the call to `invoke_subrequest()` will cause a `ValueError` exception to be raised and a response will never be generated. We can change this behavior; how to do so is described below in our discussion of the `use_tweens` argument.

Subrequests with Tweens

The `pyramid.request.Request.invoke_subrequest()` API accepts two arguments: a required positional argument `request`, and an optional keyword argument `use_tweens` which defaults to `False`.

The `request` object passed to the API must be an object that implements the Pyramid request interface (such as a `pyramid.request.Request` instance). If `use_tweens` is `True`, the request will be sent to the *tween* in the tween stack closest to the request ingress. If `use_tweens` is `False`, the request will be sent to the main router handler, and no tweens will be invoked.

In the example above, the call to `invoke_subrequest()` will always raise an exception. This is because it's using the default value for `use_tweens`, which is `False`. Alternatively, you can pass `use_tweens=True` to ensure that it will convert an exception to a `Response` if an *exception view* is configured, instead of raising the exception. This is because exception views are called by the exception view *tween* as described in *Custom Exception Views* when any view raises an exception.

We can cause the subrequest to be run through the tween stack by passing `use_tweens=True` to the call to `invoke_subrequest()`, like this:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.request import Request
4
5 def view_one(request):
6     subreq = Request.blank('/view_two')
7     response = request.invoke_subrequest(subreq, use_tweens=True)
8     return response
9
10 def view_two(request):
11     raise ValueError('foo')
12
13 def excview(request):
14     request.response.body = b'An exception was raised'
15     request.response.status_int = 500
16     return request.response
17
18 if __name__ == '__main__':
19     config = Configurator()
20     config.add_route('one', '/view_one')
21     config.add_route('two', '/view_two')
22     config.add_view(view_one, route_name='one')
23     config.add_view(view_two, route_name='two', renderer='string')
24     config.add_view(excview, context=Exception)
25     app = config.make_wsgi_app()
26     server = make_server('0.0.0.0', 8080, app)
27     server.serve_forever()
```

In the above case, the call to `request.invoke_subrequest(subreq)` will not raise an exception. Instead, it will retrieve a “500” response from the attempted invocation of `view_two`, because the tween which invokes an exception view to generate a response is run, and therefore `excview` is executed.

This is one of the major differences between specifying the `use_tweens=True` and `use_tweens=False` arguments to `invoke_subrequest()`. `use_tweens=True` may also imply invoking a transaction commit or abort for the logic executed in the subrequest if you've got `pyramid_tm` in the tween list, injecting debug HTML if you've got `pyramid_debugtoolbar` in the tween list, and other tween-related side effects as defined by your particular tween list.

The `invoke_subrequest()` function also unconditionally does the following:

- It manages the threadlocal stack so that `get_current_request()` and `get_current_registry()` work during a request (they will return the subrequest instead of the original request).
- It adds a `registry` attribute and an `invoke_subrequest` attribute (a callable) to the request object to which it is handed.
- It sets request extensions (such as those added via `add_request_method()` or `set_request_property()`) on the subrequest object passed as `request`.
- It causes a `NewRequest` event to be sent at the beginning of request processing.
- It causes a `ContextFound` event to be sent when a context resource is found.
- It ensures that the user implied by the request passed in has the necessary authorization to invoke the view callable before calling it.
- It calls any *response callback* functions defined within the subrequest's lifetime if a response is obtained from the Pyramid application.
- It causes a `NewResponse` event to be sent if a response is obtained.
- It calls any *finished callback* functions defined within the subrequest's lifetime.

The invocation of a subrequest has more or less exactly the same effect as the invocation of a request received by the Pyramid router from a web client when `use_tweens=True`. When `use_tweens=False`, the tweens are skipped but all the other steps take place.

It's a poor idea to use the original `request` object as an argument to `invoke_subrequest()`. You should construct a new request instead as demonstrated in the above example, using `pyramid.request.Request.blank()`. Once you've constructed a request object, you'll need to massage it to match the view callable that you'd like to be executed during the subrequest. This can be done by adjusting the subrequest's URL, its headers, its request method, and other attributes. The documentation for `pyramid.request.Request` exposes the methods you should call and attributes you should set on the request that you create, then massage it into something that will actually match the view you'd like to call via a subrequest.

We've demonstrated use of a subrequest from within a view callable, but you can use the `invoke_subrequest()` API from within a tween or an event handler as well. Even though you can do it, it's usually a poor idea to invoke `invoke_subrequest()` from within a tween, because tweens already, by definition, have access to a function that will cause a subrequest (they are passed a *handle* function). It's fine to invoke `invoke_subrequest()` from within an event handler, however.

Invoking an Exception View

New in version 1.7.

Pyramid apps may define *exception views* which can handle any raised exceptions that escape from your code while processing a request. By default an unhandled exception will be caught by the `EXCVIEW tween`, which will then lookup an exception view that can handle the exception type, generating an appropriate error response.

In Pyramid 1.7 the `pyramid.request.Request.invoke_exception_view()` was introduced, allowing a user to invoke an exception view while manually handling an exception. This can be useful in a few different circumstances:

- Manually handling an exception losing the current call stack or flow.
- Handling exceptions outside of the context of the `EXCVIEW tween`. The tween only covers certain parts of the request processing pipeline (See *Request Processing*). There are also some corner cases where an exception can be raised that will still bubble up to middleware, and possibly to the web server in which case a generic 500 Internal Server Error will be returned to the client.

Below is an example usage of `pyramid.request.Request.invoke_exception_view()`:

```
1 def foo(request):
2     try:
3         some_func_that_errors()
4         return response
5     except Exception:
6         response = request.invoke_exception_view()
7         if response is not None:
8             return response
9     else:
10        # there is no exception view for this exception, simply
11        # re-raise and let someone else handle it
12        raise
```

Please note that in most cases you do not need to write code like this, and you may rely on the `EXCVIEW tween` to handle this for you.

Using Hooks

“Hooks” can be used to influence the behavior of the Pyramid framework in various ways.

Changing the Not Found View

When Pyramid can't map a URL to view code, it invokes a *Not Found View*, which is a *view callable*. The default Not Found View can be overridden through application configuration.

If your application uses *imperative configuration*, you can replace the Not Found View by using the `pyramid.config.Configurator.add_notfound_view()` method:

```
1 def notfound(request):
2     return Response('Not Found', status='404 Not Found')
3
4 def main(globals, **settings):
5     config = Configurator()
6     config.add_notfound_view(notfound)
```

The *Not Found View* callable is a view callable like any other.

If your application instead uses `pyramid.view.view_config` decorators and a *scan*, you can replace the Not Found View by using the `pyramid.view.notfound_view_config` decorator:

```
1 from pyramid.view import notfound_view_config
2
3 @notfound_view_config()
4 def notfound(request):
5     return Response('Not Found', status='404 Not Found')
6
7 def main(globals, **settings):
8     config = Configurator()
9     config.scan()
```

This does exactly what the imperative example above showed.

Your application can define *multiple* Not Found Views if necessary. Both `pyramid.config.Configurator.add_notfound_view()` and `pyramid.view.notfound_view_config` take most of the same arguments as `pyramid.config.Configurator.add_view` and `pyramid.view.view_config`, respectively. This means that Not Found Views can carry predicates limiting their applicability. For example:

```
1 from pyramid.view import notfound_view_config
2
3 @notfound_view_config(request_method='GET')
4 def notfound_get(request):
5     return Response('Not Found during GET', status='404 Not Found')
```

```
6
7 @notfound_view_config(request_method='POST')
8 def notfound_post(request):
9     return Response('Not Found during POST', status='404 Not Found')
10
11 def main(globals, **settings):
12     config = Configurator()
13     config.scan()
```


The `notfound_get` view will be called when a view could not be found and the request method was GET. The `notfound_post` view will be called when a view could not be found and the request method was POST.


Like any other view, the Not Found View must accept at least a `request` parameter, or both `context` and `request`. The `request` is the current *request* representing the denied action. The `context` (if used in the call signature) will be the instance of the *HTTPNotFound* exception that caused the view to be called.

Both `pyramid.config.Configurator.add_notfound_view()` and `pyramid.view.notfound_view_config` can be used to automatically redirect requests to slash-appended routes. See *Redirecting to Slash-Appended Routes* for examples.

Here's some sample code that implements a minimal *Not Found View* callable:

```
1 from pyramid.httpexceptions import HTTPNotFound
2
3 def notfound(request):
4     return HTTPNotFound()
```

 When a Not Found View callable is invoked, it is passed a *request*. The `exception` attribute of the request will be an instance of the *HTTPNotFound* exception that caused the Not Found View to be called. The value of `request.exception.message` will be a value explaining why the Not Found exception was raised. This message has different values depending on whether the `pyramid.debug_notfound` environment setting is true or false.

 When a Not Found View callable accepts an argument list as described in *Alternate View Callable Argument/Calling Conventions*, the `context` passed as the first argument to the view callable will be the *HTTPNotFound* exception instance. If available, the resource context will still be available as `request.context`.



The *Not Found View* callables are only invoked when a *HTTPNotFound* exception is raised. If the exception is returned from a view then it will be treated as a regular response object and it will not trigger the custom view.

Changing the Forbidden View

When Pyramid can't authorize execution of a view based on the *authorization policy* in use, it invokes a *forbidden view*. The default forbidden response has a 403 status code and is very plain, but the view which generates it can be overridden as necessary.

The *forbidden view* callable is a view callable like any other. The *view configuration* which causes it to be a “forbidden” view consists of using the *pyramid.config.Configurator.add_forbidden_view()* API or the *pyramid.view.forbidden_view_config* decorator.

For example, you can add a forbidden view by using the *pyramid.config.Configurator.add_forbidden_view()* method to register a forbidden view:

```
1 def forbidden(request):
2     return Response('forbidden')
3
4 def main(globals, **settings):
5     config = Configurator()
6     config.add_forbidden_view(forbidden)
```


If instead you prefer to use decorators and a *scan*, you can use the *pyramid.view.forbidden_view_config* decorator to mark a view callable as a forbidden view:

```
1 from pyramid.view import forbidden_view_config
2
3 @forbidden_view_config()
4 def forbidden(request):
5     return Response('forbidden')
6
7 def main(globals, **settings):
8     config = Configurator()
9     config.scan()
```

Like any other view, the forbidden view must accept at least a `request` parameter, or both `context` and `request`. If a forbidden view callable accepts both `context` and `request`, the `HTTPException` is passed as `context`. The `context` as found by the router when the view was denied (which you normally would expect) is available as `request.context`. The `request` is the current *request* representing the denied action.

Here's some sample code that implements a minimal forbidden view:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 def forbidden_view(request):
5     return Response('forbidden')
```

 When a forbidden view callable is invoked, it is passed a *request*. The `exception` attribute of the request will be an instance of the `HTTPForbidden` exception that caused the forbidden view to be called. The value of `request.exception.message` will be a value explaining why the forbidden exception was raised, and `request.exception.result` will be extended information about the forbidden exception. These messages have different values depending on whether the `pyramid.debug_authorization` environment setting is true or false.



The *forbidden view* callables are only invoked when a `HTTPForbidden` exception is raised. If the exception is returned from a view then it will be treated as a regular response object and it will not trigger the custom view.

Changing the Request Factory

Whenever Pyramid handles a request from a *WSGI* server, it creates a *request* object based on the *WSGI* environment it has been passed. By default, an instance of the `pyramid.request.Request` class is created to represent the request object.

The class (a.k.a., “factory”) that Pyramid uses to create a request object instance can be changed by passing a `request_factory` argument to the constructor of the *configurator*. This argument can be either a callable or a *dotted Python name* representing a callable.

```

1 from pyramid.request import Request
2
3 class MyRequest(Request):
4     pass
5
6 config = Configurator(request_factory=MyRequest)

```

If you're doing imperative configuration, and you'd rather do it after you've already constructed a *configurator*, it can also be registered via the `pyramid.config.Configurator.set_request_factory()` method:

```

1 from pyramid.config import Configurator
2 from pyramid.request import Request
3
4 class MyRequest(Request):
5     pass
6
7 config = Configurator()
8 config.set_request_factory(MyRequest)

```

Adding Methods or Properties to a Request Object

New in version 1.4.

Since each Pyramid application can only have one *request* factory, *changing the request factory* is not that extensible, especially if you want to build composable features (e.g., Pyramid add-ons and plugins).

A lazy property can be registered to the request object via the `pyramid.config.Configurator.add_request_method()` API. This allows you to specify a callable that will be available on the request object, but will not actually execute the function until accessed.



This will silently override methods and properties from *request factory* that have the same name.

```

1 from pyramid.config import Configurator
2
3 def total(request, *args):
4     return sum(args)
5

```



```
6 def prop(request):
7     print("getting the property")
8     return "the property"
9
10 config = Configurator()
11 config.add_request_method(total)
12 config.add_request_method(prop, reify=True)
```

In the above example, `total` is added as a method. However, `prop` is added as a property and its result is cached per-request by setting `reify=True`. This way, we eliminate the overhead of running the function multiple times.

```
>>> request.total(1, 2, 3)
6
>>> request.prop
getting the property
'the property'
>>> request.prop
'the property'
```

To not cache the result of `request.prop`, set `property=True` instead of `reify=True`.

Here is an example of passing a class to `Configurator.add_request_method`:

```
1 from pyramid.config import Configurator
2 from pyramid.decorator import reify
3
4 class ExtraStuff(object):
5
6     def __init__(self, request):
7         self.request = request
8
9     def total(self, *args):
10         return sum(args)
11
12     # use @property if you don't want to cache the result
13     @reify
14     def prop(self):
15         print("getting the property")
16         return "the property"
17
18 config = Configurator()
19 config.add_request_method(ExtraStuff, 'extra', reify=True)
```

We attach and cache an object named `extra` to the `request` object.

```
>>> request.extra.total(1, 2, 3)
6
>>> request.extra.prop
getting the property
'the property'
>>> request.extra.prop
'the property'
```

Changing the Response Factory

New in version 1.6.

Whenever Pyramid returns a response from a view, it creates a *response* object. By default, an instance of the `pyramid.response.Response` class is created to represent the response object.

The factory that Pyramid uses to create a response object instance can be changed by passing a `pyramid.interfaces.IResponseFactory` argument to the constructor of the *configurator*. This argument can be either a callable or a *dotted Python name* representing a callable.

The factory takes a single positional argument, which is a *Request* object. The argument may be `None`.

```
1 from pyramid.response import Response
2
3 class MyResponse(Response):
4     pass
5
6 config = Configurator(response_factory=lambda r: MyResponse())
```

If you're doing imperative configuration and you'd rather do it after you've already constructed a *configurator*, it can also be registered via the `pyramid.config.Configurator.set_response_factory()` method:

```
1 from pyramid.config import Configurator
2 from pyramid.response import Response
3
4 class MyResponse(Response):
5     pass
6
7 config = Configurator()
8 config.set_response_factory(lambda r: MyResponse())
```

Using the Before Render Event

Subscribers to the `pyramid.events.BeforeRender` event may introspect and modify the set of *renderer globals* before they are passed to a *renderer*. This event object itself has a dictionary-like interface that can be used for this purpose. For example:

```
1 from pyramid.events import subscriber
2 from pyramid.events import BeforeRender
3
4 @subscriber(BeforeRender)
5 def add_global(event):
6     event['mykey'] = 'foo'
```

An object of this type is sent as an event just before a *renderer* is invoked.

If a subscriber attempts to add a key that already exists in the *renderer globals* dictionary, a `KeyError` is raised. This limitation is enforced because event subscribers do not possess any relative ordering. The set of keys added to the *renderer globals* dictionary by all `pyramid.events.BeforeRender` subscribers and *renderer globals* factories must be unique.

The dictionary returned from the view is accessible through the `rendering_val` attribute of a *BeforeRender* event.

Suppose you return `{'mykey': 'somevalue', 'mykey2': 'somevalue2'}` from your view callable, like so:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='some_renderer')
4 def myview(request):
5     return {'mykey': 'somevalue', 'mykey2': 'somevalue2'}
```

`rendering_val` can be used to access these values from the *BeforeRender* object:

```
1 from pyramid.events import subscriber
2 from pyramid.events import BeforeRender
3
4 @subscriber(BeforeRender)
5 def read_return(event):
6     # {'mykey': 'somevalue'} is returned from the view
7     print(event.rendering_val['mykey'])
```

See the API documentation for the *BeforeRender* event interface at `pyramid.interfaces.IBeforeRender`.

Using Response Callbacks

Unlike many other web frameworks, Pyramid does not eagerly create a global response object. Adding a *response callback* allows an application to register an action to be performed against whatever response object is returned by a view, usually in order to mutate the response.

The `pyramid.request.Request.add_response_callback()` method is used to register a response callback.

A response callback is a callable which accepts two positional parameters: `request` and `response`. For example:

```
1 def cache_callback(request, response):
2     """Set the cache_control max_age for the response"""
3     if request.exception is not None:
4         response.cache_control.max_age = 360
5     request.add_response_callback(cache_callback)
```

No response callback is called if an unhandled exception happens in application code, or if the response object returned by a *view callable* is invalid. Response callbacks *are*, however, invoked when a *exception view* is rendered successfully. In such a case, the `request.exception` attribute of the request when it enters a response callback will be an exception object instead of its default value of `None`.

Response callbacks are called in the order they're added (first-to-most-recently-added). All response callbacks are called *before* the *NewResponse* event is sent. Errors raised by response callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

A response callback has a lifetime of a *single* request. If you want a response callback to happen as the result of *every* request, you must re-register the callback into every new request (perhaps within a subscriber of a *NewRequest* event).

Using Finished Callbacks

A *finished callback* is a function that will be called unconditionally by the Pyramid *router* at the very end of request processing. A finished callback can be used to perform an action at the end of a request unconditionally.

The `pyramid.request.Request.add_finished_callback()` method is used to register a finished callback.

A finished callback is a callable which accepts a single positional parameter: `request`. For example:

```
1 import logging
2
3 log = logging.getLogger(__name__)
4
5 def log_callback(request):
6     """Log information at the end of request"""
7     log.debug('Request is finished.')
8     request.add_finished_callback(log_callback)
```

Finished callbacks are called in the order they’re added (first-to-most-recently-added). Finished callbacks (unlike a *response callback*) are *always* called, even if an exception happens in application code that prevents a response from being generated.

The set of finished callbacks associated with a request are called *very late* in the processing of that request; they are essentially the very last thing called by the *router* before a request “ends”. They are called after response processing has already occurred in a top-level `finally:` block within the router request processing code. As a result, mutations performed to the `request` provided to a finished callback will have no meaningful effect, because response processing will have already occurred, and the request’s scope will expire almost immediately after all finished callbacks have been processed.

Errors raised by finished callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

A finished callback has a lifetime of a *single* request. If you want a finished callback to happen as the result of *every* request, you must re-register the callback into every new request (perhaps within a subscriber of a *NewRequest* event).

Changing the Traverser

The default *traversal* algorithm that Pyramid uses is explained in *The Traversal Algorithm*. Though it is rarely necessary, this default algorithm can be swapped out selectively for a different traversal pattern via configuration.

```
1 from pyramid.config import Configurator
2 from myapp.traversal import Traverser
3 config = Configurator()
4 config.add_traverser(Traverser)
```

In the example above, `myapp.traversal.Traverser` is assumed to be a class that implements the following interface:

```

1 class Traverser(object):
2     def __init__(self, root):
3         """ Accept the root object returned from the root factory """
4
5     def __call__(self, request):
6         """ Return a dictionary with (at least) the keys ``root``,
7             ``context``, ``view_name``, ``subpath``, ``traversed``,
8             ``virtual_root``, and ``virtual_root_path``. These values are
9             typically the result of a resource tree traversal. ``root``
10            is the physical root object, ``context`` will be a resource
11            object, ``view_name`` will be the view name used (a Unicode
12            name), ``subpath`` will be a sequence of Unicode names that
13            followed the view name but were not traversed, ``traversed``
14            will be a sequence of Unicode names that were traversed
15            (including the virtual root path, if any) ``virtual_root``
16            will be a resource object representing the virtual root (or the
17            physical root if traversal was not performed), and
18            ``virtual_root_path`` will be a sequence representing the
19            virtual root path (a sequence of Unicode names) or None if
20            traversal was not performed.
21
22            Extra keys for special purpose functionality can be added as
23            necessary.
24
25            All values returned in the dictionary will be made available
26            as attributes of the ``request`` object.
27            """

```

More than one traversal algorithm can be active at the same time. For instance, if your *root factory* returns more than one type of object conditionally, you could claim that an alternative traverser adapter is “for” only one particular class or interface. When the root factory returned an object that implemented that class or interface, a custom traverser would be used. Otherwise the default traverser would be used. For example:

```

1 from myapp.traversal import Traverser
2 from myapp.resources import MyRoot
3 from pyramid.config import Configurator
4 config = Configurator()
5 config.add_traverser(Traverser, MyRoot)

```

If the above stanza was added to a Pyramid `__init__.py` file’s main function, Pyramid would use the `myapp.traversal.Traverser` only when the application *root factory* returned an instance of the `myapp.resources.MyRoot` object. Otherwise it would use the default Pyramid traverser to do traversal.

Changing How `pyramid.request.Request.resource_url()` Generates a URL

When you add a traverser as described in *Changing the Traverser*, it's often convenient to continue to use the `pyramid.request.Request.resource_url()` API. However, since the way traversal is done will have been modified, the URLs it generates by default may be incorrect when used against resources derived from your custom traverser.

If you've added a traverser, you can change how `resource_url()` generates a URL for a specific type of resource by adding a call to `pyramid.config.Configurator.add_resource_url_adapter()`.

For example:

```
1 from myapp.traversal import ResourceURLAdapter
2 from myapp.resources import MyRoot
3
4 config.add_resource_url_adapter(ResourceURLAdapter, MyRoot)
```

In the above example, the `myapp.traversal.ResourceURLAdapter` class will be used to provide services to `resource_url()` any time the *resource* passed to `resource_url` is of the class `myapp.resources.MyRoot`. The *resource_iface* argument `MyRoot` represents the type of interface that must be possessed by the resource for this resource url factory to be found. If the *resource_iface* argument is omitted, this resource URL adapter will be used for *all* resources.

The API that must be implemented by a class that provides *IResourceURL* is as follows:

```
1 class MyResourceURL(object):
2     """ An adapter which provides the virtual and physical paths of a
3         resource
4     """
5     def __init__(self, resource, request):
6         """ Accept the resource and request and set self.physical_path and
7             self.virtual_path """
8         self.virtual_path = some_function_of(resource, request)
9         self.virtual_path_tuple = some_function_of(resource, request)
10        self.physical_path = some_other_function_of(resource, request)
11        self.physical_path_tuple = some_function_of(resource, request)
```

The default context URL generator is available for perusal as the class `pyramid.traversal.ResourceURL` in the traversal module of the *Pylons* GitHub Pyramid repository.

See `pyramid.config.Configurator.add_resource_url_adapter()` for more information.

Changing How Pyramid Treats View Responses

New in version 1.1.

It is possible to control how Pyramid treats the result of calling a view callable on a per-type basis by using a hook involving `pyramid.config.Configurator.add_response_adapter()` or the `response_adapter` decorator.

Pyramid, in various places, adapts the result of calling a view callable to the `IResponse` interface to ensure that the object returned by the view callable is a “true” response object. The vast majority of time, the result of this adaptation is the result object itself, as view callables written by “civilians” who read the narrative documentation contained in this manual will always return something that implements the `IResponse` interface. Most typically, this will be an instance of the `pyramid.response.Response` class or a subclass. If a civilian returns a non-Response object from a view callable that isn’t configured to use a `renderer`, they will typically expect the router to raise an error. However, you can hook Pyramid in such a way that users can return arbitrary values from a view callable by providing an adapter which converts the arbitrary return value into something that implements `IResponse`.

For example, if you’d like to allow view callables to return bare string objects (without requiring a `renderer` to convert a string to a response object), you can register an adapter which converts the string to a Response:

```
1 from pyramid.response import Response
2
3 def string_response_adapter(s):
4     response = Response(s)
5     return response
6
7 # config is an instance of pyramid.config.Configurator
8
9 config.add_response_adapter(string_response_adapter, str)
```

Likewise, if you want to be able to return a simplified kind of response object from view callables, you can use the `IResponse` hook to register an adapter to the more complex `IResponse` interface:

```
1 from pyramid.response import Response
2
3 class SimpleResponse(object):
4     def __init__(self, body):
5         self.body = body
6
7 def simple_response_adapter(simple_response):
8     response = Response(simple_response.body)
```



```
9     return response
10
11 # config is an instance of pyramid.config.Configurator
12
13 config.add_response_adapter(simple_response_adapter, SimpleResponse)
```

If you want to implement your own `Response` object instead of using the `pyramid.response.Response` object in any capacity at all, you'll have to make sure that the object implements every attribute and method outlined in `pyramid.interfaces.IResponse` and you'll have to ensure that it uses `zope.interface.implementer(IResponse)` as a class decorator.

```
1 from pyramid.interfaces import IResponse
2 from zope.interface import implementer
3
4 @implementer(IResponse)
5 class MyResponse(object):
6     # ... an implementation of every method and attribute
7     # documented in IResponse should follow ...
```

When an alternate response object implementation is returned by a view callable, if that object asserts that it implements `IResponse` (via `zope.interface.implementer(IResponse)`), an adapter needn't be registered for the object; Pyramid will use it directly.

An `IResponse` adapter for `webob.Response` (as opposed to `pyramid.response.Response`) is registered by Pyramid by default at startup time, as by their nature, instances of this class (and instances of subclasses of the class) will natively provide `IResponse`. The adapter registered for `webob.Response` simply returns the response object.

Instead of using `pyramid.config.Configurator.add_response_adapter()`, you can use the `pyramid.response.response_adapter` decorator:

```
1 from pyramid.response import Response
2 from pyramid.response import response_adapter
3
4 @response_adapter(str)
5 def string_response_adapter(s):
6     response = Response(s)
7     return response
```

The above example, when scanned, has the same effect as:

```
config.add_response_adapter(string_response_adapter, str)
```

The *response_adapter* decorator will have no effect until activated by a *scan*.

Using a View Mapper

The default calling conventions for view callables are documented in the *Views* chapter. You can change the way users define view callables by employing a *view mapper*.

A view mapper is an object that accepts a set of keyword arguments and which returns a callable. The returned callable is called with the *view callable* object. The returned callable should itself return another callable which can be called with the “internal calling protocol” (*context*, *request*).

You can use a view mapper in a number of ways:

- by setting a `__view_mapper__` attribute (which is the view mapper object) on the view callable itself
- by passing the mapper object to `pyramid.config.Configurator.add_view()` (or its declarative and decorator equivalents) as the mapper argument
- by registering a *default* view mapper

Here’s an example of a view mapper that emulates (somewhat) a Pylons “controller”. The mapper is initialized with some keyword arguments. Its `__call__` method accepts the view object (which will be a class). It uses the `attr` keyword argument it is passed to determine which attribute should be used as an action method. The wrapper method it returns accepts (*context*, *request*) and returns the result of calling the action method with keyword arguments implied by the *matchdict* after popping the *action* out of it. This somewhat emulates the Pylons style of calling action methods with routing parameters pulled out of the route matching dict as keyword arguments.

```
1 # framework
2
3 class PylonsControllerViewMapper(object):
4     def __init__(self, **kw):
5         self.kw = kw
6
7     def __call__(self, view):
8         attr = self.kw['attr']
9         def wrapper(context, request):
10             matchdict = request.matchdict.copy()
```

```
11         matchdict.pop('action', None)
12         inst = view(request)
13         meth = getattr(inst, attr)
14         return meth(**matchdict)
15     return wrapper
16
17 class BaseController(object):
18     __view_mapper__ = PylonsControllerViewMapper
```

A user might make use of these framework components like so:

```
1  # user application
2
3  from pyramid.response import Response
4  from pyramid.config import Configurator
5  import pyramid_handlers
6  from wsgiref.simple_server import make_server
7
8  class MyController(BaseController):
9      def index(self, id):
10         return Response(id)
11
12  if __name__ == '__main__':
13      config = Configurator()
14      config.include(pyramid_handlers)
15      config.add_handler('one', '/{id}', MyController, action='index')
16      config.add_handler('two', '/{action}/{id}', MyController)
17      server.make_server('0.0.0.0', 8080, config.make_wsgi_app())
18      server.serve_forever()
```

The `pyramid.config.Configurator.set_view_mapper()` method can be used to set a *default* view mapper (overriding the superdefault view mapper used by Pyramid itself).

A *single* view registration can use a view mapper by passing the mapper as the `mapper` argument to `add_view()`.

Registering Configuration Decorators

Decorators such as `view_config` don't change the behavior of the functions or classes they're decorating. Instead when a *scan* is performed, a modified version of the function or class is registered with Pyramid.

You may wish to have your own decorators that offer such behaviour. This is possible by using the *Venusian* package in the same way that it is used by Pyramid.

By way of example, let's suppose you want to write a decorator that registers the function it wraps with a *Zope Component Architecture* “utility” within the *application registry* provided by Pyramid. The application registry and the utility inside the registry is likely only to be available once your application's configuration is at least partially completed. A normal decorator would fail as it would be executed before the configuration had even begun.

However, using *Venusian*, the decorator could be written as follows:

```

1 import venusian
2 from mypackage.interfaces import IMyUtility
3
4 class registerFunction(object):
5
6     def __init__(self, path):
7         self.path = path
8
9     def register(self, scanner, name, wrapped):
10         registry = scanner.config.registry
11         registry.getUtility(IMyUtility).register(
12             self.path, wrapped)
13
14     def __call__(self, wrapped):
15         venusian.attach(wrapped, self.register)
16         return wrapped

```

This decorator could then be used to register functions throughout your code:

```

1 @registerFunction('/some/path')
2 def my_function():
3     do_stuff()

```

However, the utility would only be looked up when a *scan* was performed, enabling you to set up the utility in advance:

```

1 from zope.interface import implementer
2
3 from wsgiref.simple_server import make_server
4 from pyramid.config import Configurator
5 from mypackage.interfaces import IMyUtility
6
7 @implementer(IMyUtility)

```

```
8 class UtilityImplementation:
9
10     def __init__(self):
11         self.registrations = {}
12
13     def register(self, path, callable_):
14         self.registrations[path] = callable_
15
16 if __name__ == '__main__':
17     config = Configurator()
18     config.registry.registerUtility(UtilityImplementation())
19     config.scan()
20     app = config.make_wsgi_app()
21     server = make_server('0.0.0.0', 8080, app)
22     server.serve_forever()
```

For full details, please read the Venusian documentation.

Registering Tweens

New in version 1.2: Tweens

A *tween* (a contraction of the word “between”) is a bit of code that sits between the Pyramid router’s main request handling function and the upstream WSGI component that uses Pyramid as its “app”. This is a feature that may be used by Pyramid framework extensions to provide, for example, Pyramid-specific view timing support bookkeeping code that examines exceptions before they are returned to the upstream WSGI application. Tweens behave a bit like *WSGI middleware*, but they have the benefit of running in a context in which they have access to the Pyramid *request*, *response*, and *application registry*, as well as the Pyramid rendering machinery.

Creating a Tween

To create a tween, you must write a “tween factory”. A tween factory must be a globally importable callable which accepts two arguments: *handler* and *registry*. *handler* will be either the main Pyramid request handling function or another tween. *registry* will be the Pyramid *application registry* represented by this Configurator. A tween factory must return the tween (a callable object) when it is called.

A tween is called with a single argument, *request*, which is the *request* created by Pyramid’s router when it receives a WSGI request. A tween should return a *response*, usually the one generated by the downstream Pyramid application.

You can write the tween factory as a simple closure-returning function:

```
1 def simple_tween_factory(handler, registry):
2     # one-time configuration code goes here
3
4     def simple_tween(request):
5         # code to be executed for each request before
6         # the actual application code goes here
7
8         response = handler(request)
9
10        # code to be executed for each request after
11        # the actual application code goes here
12
13        return response
14
15    return simple_tween
```

Alternatively, the tween factory can be a class with the `__call__` magic method:

```
1 class simple_tween_factory(object):
2     def __init__(self, handler, registry):
3         self.handler = handler
4         self.registry = registry
5
6         # one-time configuration code goes here
7
8     def __call__(self, request):
9         # code to be executed for each request before
10        # the actual application code goes here
11
12        response = self.handler(request)
13
14        # code to be executed for each request after
15        # the actual application code goes here
16
17        return response
```

You should avoid mutating any state on the tween instance. The tween is invoked once per request and any shared mutable state needs to be carefully handled to avoid any race conditions.

The closure style performs slightly better and enables you to conditionally omit the tween from the request processing pipeline (see the following timing tween example), whereas the class style makes it easier to have shared mutable state and allows subclassing.

Here's a complete example of a tween that logs the time spent processing each request:

```
1 # in a module named myapp.tweens
2
3 import time
4 from pyramid.settings import asbool
5 import logging
6
7 log = logging.getLogger(__name__)
8
9 def timing_tween_factory(handler, registry):
10     if asbool(registry.settings.get('do_timing')):
11         # if timing support is enabled, return a wrapper
12         def timing_tween(request):
13             start = time.time()
14             try:
15                 response = handler(request)
16             finally:
17                 end = time.time()
18                 log.debug('The request took %s seconds' %
19                           (end - start))
20             return response
21         return timing_tween
22     # if timing support is not enabled, return the original
23     # handler
24     return handler
```

In the above example, the tween factory defines a `timing_tween` tween and returns it if `asbool(registry.settings.get('do_timing'))` is true. It otherwise simply returns the handler which it was given. The `registry.settings` attribute is a handle to the deployment settings provided by the user (usually in an `.ini` file). In this case, if the user has defined a `do_timing` setting and that setting is `True`, the user has said they want to do timing, so the tween factory returns the timing tween; it otherwise just returns the handler it has been provided, preventing any timing.

The example timing tween simply records the start time, calls the downstream handler, logs the number of seconds consumed by the downstream handler, and returns the response.

Registering an Implicit Tween Factory

Once you've created a tween factory, you can register it into the implicit tween chain using the `pyramid.config.Configurator.add_tween()` method using its *dotted Python name*.

Here's an example of registering a tween factory as an "implicit" tween in a Pyramid application:

```

1 from pyramid.config import Configurator
2 config = Configurator()
3 config.add_tween('myapp.twens.timing_tween_factory')

```

Note that you must use a *dotted Python name* as the first argument to `pyramid.config.Configurator.add_tween()`; this must point at a tween factory. You cannot pass the tween factory object itself to the method: it must be *dotted Python name* that points to a globally importable object. In the above example, we assume that a `timing_tween_factory` tween factory was defined in a module named `myapp.twens`, so the tween factory is importable as `myapp.twens.timing_tween_factory`.

When you use `pyramid.config.Configurator.add_tween()`, you’re instructing the system to use your tween factory at startup time unless the user has provided an explicit tween list in their configuration. This is what’s meant by an “implicit” tween. A user can always elect to supply an explicit tween list, reordering or disincluding implicitly added tweens. See *Explicit Tween Ordering* for more information about explicit tween ordering.

If more than one call to `pyramid.config.Configurator.add_tween()` is made within a single application configuration, the tweens will be chained together at application startup time. The *first* tween factory added via `add_tween` will be called with the Pyramid exception view tween factory as its handler argument, then the tween factory added directly after that one will be called with the result of the first tween factory as its handler argument, and so on, ad infinitum until all tween factories have been called. The Pyramid router will use the outermost tween produced by this chain (the tween generated by the very last tween factory added) as its request handler function. For example:

```

1 from pyramid.config import Configurator
2
3 config = Configurator()
4 config.add_tween('myapp.tween_factory1')
5 config.add_tween('myapp.tween_factory2')

```

The above example will generate an implicit tween chain that looks like this:

```

INGRESS (implicit)
myapp.tween_factory2
myapp.tween_factory1
pyramid.twens.excview_tween_factory (implicit)
MAIN (implicit)

```


Suggesting Implicit Tween Ordering

By default, as described above, the ordering of the chain is controlled entirely by the relative ordering of calls to `pyramid.config.Configurator.add_tween()`. However, the caller of `add_tween` can provide an optional hint that can influence the implicit tween chain ordering by supplying `under` or `over` (or both) arguments to `add_tween()`. These hints are only used when an explicit tween ordering is not used. See *Explicit Tween Ordering* for a description of how to set an explicit tween ordering.

Allowable values for `under` or `over` (or both) are:

- None (the default),
- a *dotted Python name* to a tween factory: a string representing the predicted dotted name of a tween factory added in a call to `add_tween` in the same configuration session,
- one of the constants `pyramid.tweens.MAIN`, `pyramid.tweens.INGRESS`, or `pyramid.tweens.EXCVIEW`, or
- an iterable of any combination of the above. This allows the user to specify fallbacks if the desired tween is not included, as well as compatibility with multiple other tweens.

Effectively, `over` means “closer to the request ingress than” and `under` means “closer to the main Pyramid application than”. You can think of an onion with outer layers over the inner layers, the application being under all the layers at the center.

For example, the following call to `add_tween()` will attempt to place the tween factory represented by `myapp.tween_factory` directly “above” (in ptweens order) the main Pyramid request handler.

```
1 import pyramid.tweens
2
3 config.add_tween('myapp.tween_factory', over=pyramid.tweens.MAIN)
```

The above example will generate an implicit tween chain that looks like this:

```
INGRESS (implicit)
pyramid.tweens.excview_tween_factory (implicit)
myapp.tween_factory
MAIN (implicit)
```

Likewise, calling the following call to `add_tween()` will attempt to place this tween factory “above” the main handler but “below” a separately added tween factory:

```

1 import pyramid.tweens
2
3 config.add_tween('myapp.tween_factory1',
4                 over=pyramid.tweens.MAIN)
5 config.add_tween('myapp.tween_factory2',
6                 over=pyramid.tweens.MAIN,
7                 under='myapp.tween_factory1')

```

The above example will generate an implicit tween chain that looks like this:

```

INGRESS (implicit)
pyramid.tweens.excview_tween_factory (implicit)
myapp.tween_factory1
myapp.tween_factory2
MAIN (implicit)

```

Specifying neither `over` nor `under` is equivalent to specifying `under=INGRESS`.

If all options for `under` (or `over`) cannot be found in the current configuration, it is an error. If some options are specified purely for compatibility with other tweens, just add a fallback of `MAIN` or `INGRESS`. For example, `under=('someothertween', 'someothertween2', INGRESS)`. This constraint will require the tween to be located under the `someothertween` tween, the `someothertween2` tween, and `INGRESS`. If any of these is not in the current configuration, this constraint will only organize itself based on the tweens that are present.

Explicit Tween Ordering

Implicit tween ordering is obviously only best-effort. Pyramid will attempt to provide an implicit order of tweens as best it can using hints provided by calls to `add_tween()`. But because it's only best-effort, if very precise tween ordering is required, the only surefire way to get it is to use an explicit tween order. The deploying user can override the implicit tween inclusion and ordering implied by calls to `add_tween()` entirely by using the `pyramid.tweens` settings value. When used, this settings value must be a list of Python dotted names which will override the ordering (and inclusion) of tween factories in the implicit tween chain. For example:


```

1 [app:main]
2 use = egg:MyApp
3 pyramid.reload_templates = true
4 pyramid.debug_authorization = false
5 pyramid.debug_notfound = false

```

```
6 pyramid.debug_routematch = false
7 pyramid.debug_templates = true
8 pyramid.tweens = myapp.my_cool_tween_factory
9                 pyramid.tweens.excview_tween_factory
```

In the above configuration, calls made during configuration to `pyramid.config.Configurator.add_tween()` are ignored, and the user is telling the system to use the tween factories he has listed in the `pyramid.tweens` configuration setting (each is a *dotted Python name* which points to a tween factory) instead of any tween factories added via `pyramid.config.Configurator.add_tween()`. The *first* tween factory in the `pyramid.tweens` list will be used as the producer of the effective Pyramid request handling function; it will wrap the tween factory declared directly “below” it, ad infinitum. The “main” Pyramid request handler is implicit, and always “at the bottom”.

 Pyramid’s own *exception view* handling logic is implemented as a tween factory function: `pyramid.tweens.excview_tween_factory()`. If Pyramid exception view handling is desired, and tween factories are specified via the `pyramid.tweens` configuration setting, the `pyramid.tweens.excview_tween_factory()` function must be added to the `pyramid.tweens` configuration setting list explicitly. If it is not present, Pyramid will not perform exception view handling.

Tween Conflicts and Ordering Cycles

Pyramid will prevent the same tween factory from being added to the tween chain more than once using configuration conflict detection. If you wish to add the same tween factory more than once in a configuration, you should either: (a) use a tween factory that is a separate globally importable instance object from the factory that it conflicts with; (b) use a function or class as a tween factory with the same logic as the other tween factory it conflicts with, but with a different `__name__` attribute; or (c) call `pyramid.config.Configurator.commit()` between calls to `pyramid.config.Configurator.add_tween()`.

If a cycle is detected in implicit tween ordering when `over` and `under` are used in any call to `add_tween`, an exception will be raised at startup time.

Displaying Tween Ordering

The `ptweens` command-line utility can be used to report the current implicit and explicit tween chains used by an application. See *Displaying “Tweens”*.

Adding a Third Party View, Route, or Subscriber Predicate

New in version 1.4.

View and Route Predicates

View and route predicates used during configuration allow you to narrow the set of circumstances under which a view or route will match. For example, the `request_method` view predicate can be used to ensure a view callable is only invoked when the request's method is POST:

```
@view_config(request_method='POST')
def someview(request):
    ...
```

Likewise, a similar predicate can be used as a *route* predicate:

```
config.add_route('name', '/foo', request_method='POST')
```

Many other built-in predicates exists (`request_param`, and others). You can add third-party predicates to the list of available predicates by using one of `pyramid.config.Configurator.add_view_predicate()` or `pyramid.config.Configurator.add_route_predicate()`. The former adds a view predicate, the latter a route predicate.

When using one of those APIs, you pass a *name* and a *factory* to add a predicate during Pyramid's configuration stage. For example:

```
config.add_view_predicate('content_type', ContentTypePredicate)
```

The above example adds a new predicate named `content_type` to the list of available predicates for views. This will allow the following view configuration statement to work:

```
1 @view_config(content_type='File')
2 def aview(request): ...
```

The first argument to `pyramid.config.Configurator.add_view_predicate()`, the *name*, is a string representing the name that is expected to be passed to `view_config` (or its imperative analogue `add_view`).

The second argument is a view or route predicate factory, or a *dotted Python name* which refers to a view or route predicate factory. A view or route predicate factory is most often a class with a constructor (`__init__`), a text method, a `phash` method, and a `__call__` method. For example:

```
1 class ContentTypePredicate(object):
2     def __init__(self, val, config):
3         self.val = val
4
5     def text(self):
6         return 'content_type = %s' % (self.val,)
7
8     phash = text
9
10    def __call__(self, context, request):
11        return request.content_type == self.val
```

The constructor of a predicate factory takes two arguments: `val` and `config`. The `val` argument will be the argument passed to `view_config` (or `add_view`). In the example above, it will be the string `File`. The second argument, `config`, will be the `Configurator` instance at the time of configuration.

The `text` method must return a string. It should be useful to describe the behavior of the predicate in error messages.

The `phash` method must return a string or a sequence of strings. It's most often the same as `text`, as long as `text` uniquely describes the predicate's name and the value passed to the constructor. If `text` is more general, or doesn't describe things that way, `phash` should return a string with the name and the value serialized. The result of `phash` is not seen in output anywhere, it just informs the uniqueness constraints for view configuration.

The `__call__` method differs depending on whether the predicate is used as a *view predicate* or a *route predicate*:

- When used as a route predicate, the `__call__` signature is `(info, request)`. The `info` object is a dictionary containing two keys: `match` and `route`. `info['match']` is the match-dict containing the patterns matched in the route pattern. `info['route']` is the `pyramid.interfaces.IRoute` object for the current route.
- When used as a view predicate, the `__call__` signature is `(context, request)`. The `context` is the result of *traversal* performed using either the route's *root factory* or the app's *default root factory*.

In both cases the `__call__` method is expected to return `True` or `False`.

It is possible to use the same predicate factory as both a view predicate and as a route predicate, but they'll need to handle the `info` or `context` argument specially (many predicates do not need this argument) and you'll need to call `add_view_predicate` and `add_route_predicate` separately with the same factory.

Subscriber Predicates

Subscriber predicates work almost exactly like view and route predicates. They narrow the set of circumstances in which a subscriber will be called. There are several minor differences between a subscriber predicate and a view or route predicate:

- There are no default subscriber predicates. You must register one to use one.
- The `__call__` method of a subscriber predicate accepts a single event object instead of a context and a request.
- Not every subscriber predicate can be used with every event type. Some subscriber predicates will assume a certain event type.

Here's an example of a subscriber predicate that can be used in conjunction with a subscriber that subscribes to the `pyramid.events.NewRequest` event type.

```
1 class RequestPathStartsWith(object):
2     def __init__(self, val, config):
3         self.val = val
4
5     def text(self):
6         return 'path_startswith = %s' % (self.val,)
7
8     phash = text
9
10    def __call__(self, event):
11        return event.request.path.startswith(self.val)
```

Once you've created a subscriber predicate, it may be registered via `pyramid.config.Configurator.add_subscriber_predicate()`. For example:

```
config.add_subscriber_predicate(
    'request_path_startswith', RequestPathStartsWith)
```

Once a subscriber predicate is registered, you can use it in a call to `pyramid.config.Configurator.add_subscriber()` or to `pyramid.events.subscriber`. Here's an example of using the previously registered `request_path_startswith` predicate in a call to `add_subscriber()`:

```
1 # define a subscriber in your code
2
3 def yosubscriber(event):
4     event.request.yo = 'YO!'
5
6 # and at configuration time
7
8 config.add_subscriber(yosubscriber, NewRequest,
9                       request_path_startswith='/add_yo')
```

Here's the same subscriber/predicate/event-type combination used via *subscriber*.

```
1 from pyramid.events import subscriber
2
3 @subscriber(NewRequest, request_path_startswith='/add_yo')
4 def yosubscriber(event):
5     event.request.yo = 'YO!'
```

In either of the above configurations, the `yosubscriber` callable will only be called if the request path starts with `/add_yo`. Otherwise the event subscriber will not be called.

Note that the `request_path_startswith` subscriber you defined can be used with events that have a `request` attribute, but not ones that do not. So, for example, the predicate can be used with subscribers registered for `pyramid.events.NewRequest` and `pyramid.events.ContextFound` events, but it cannot be used with subscribers registered for `pyramid.events.ApplicationCreated` because the latter type of event has no `request` attribute. The point being, unlike route and view predicates, not every type of subscriber predicate will necessarily be applicable for use in every subscriber registration. It is not the responsibility of the predicate author to make every predicate make sense for every event type; it is the responsibility of the predicate consumer to use predicates that make sense for a particular event type registration.

View Derivers

New in version 1.7.

Every URL processed by Pyramid is matched against a custom view pipeline. See *Request Processing* for how this works. The view pipeline itself is built from the user-supplied *view callable*, which is then composed with *view derivers*. A view deriver is a composable element of the view pipeline which is used to wrap a view with added functionality. View derivers are very similar to the `decorator` argument to `pyramid.config.Configurator.add_view()`, except that they have the option to execute for every view in the application.

It is helpful to think of a *view deriver* as middleware for views. Unlike tweens or WSGI middleware which are scoped to the application itself, a view deriver is invoked once per view in the application, and can use configuration options from the view to customize its behavior.

Built-in View Derivers

There are several built-in view derivers that Pyramid will automatically apply to any view. Below they are defined in order from furthest to closest to the user-defined *view callable*:

`secured_view`

Enforce the permission defined on the view. This element is a no-op if no permission is defined. Note there will always be a permission defined if a default permission was assigned via `pyramid.config.Configurator.set_default_permission()`.

This element will also output useful debugging information when `pyramid.debug_authorization` is enabled.

`csrf_view`

Used to check the CSRF token provided in the request. This element is a no-op if `require_csrf` view option is not `True`. Note there will always be a `require_csrf` option if a default value was assigned via `pyramid.config.Configurator.set_default_csrf_options()`.

`owrapped_view`

Invokes the wrapped view defined by the `wrapper` option.

`http_cached_view`

Applies cache control headers to the response defined by the `http_cache` option. This element is a no-op if the `pyramid.prevent_http_cache` setting is enabled or the `http_cache` option is `None`.

`decorated_view`

Wraps the view with the decorators from the `decorator` option.

`rendered_view`

Adapts the result of the *view callable* into a *response* object. Below this point the result may be any Python object.

`mapped_view`

Applies the *view mapper* defined by the `mapper` option or the application's default view mapper to the *view callable*. This is always the closest deriver to the user-defined view and standardizes the view pipeline interface to accept `(context, request)` from all previous view derivers.



Any view drivers defined under the `rendered_view` are not guaranteed to receive a valid response object. Rather they will receive the result from the *view mapper* which is likely the original response returned from the view. This is possibly a dictionary for a renderer but it may be any Python object that may be adapted into a response.

Custom View Drivers

It is possible to define custom view drivers which will affect all views in an application. There are many uses for this, but most will likely be centered around monitoring and security. In order to register a custom *view driver*, you should create a callable that conforms to the `pyramid.interfaces.IViewDriver` interface, and then register it with your application using `pyramid.config.Configurator.add_view_driver()`. For example, below is a callable that can provide timing information for the view pipeline:

```
1 import time
2
3 def timing_view(view, info):
4     if info.options.get('timed'):
5         def wrapper_view(context, request):
6             start = time.time()
7             response = view(context, request)
8             end = time.time()
9             response.headers['X-View-Performance'] = '%.3f' % (end - start,
10 →)
11             return response
12         return wrapper_view
13     return view
14
15 timing_view.options = ('timed',)
16
17 config.add_view_driver(timing_view)
```

The setting of `timed` on the `timing_view` signifies to Pyramid that `timed` is a valid `view_config` keyword argument now. The `timing_view` custom view driver as registered above will only be active for any view defined with a `timed=True` value passed as one of its `view_config` keywords.

For example, this view configuration will *not* be a timed view:

```
1 @view_config(route_name='home')
2 def home(request):
3     return Response('Home')
```

But this view *will* have timing information added to the response headers:

```
1 @view_config(route_name='home', timed=True)
2 def home(request):
3     return Response('Home')
```

View drivers are unique in that they have access to most of the options passed to `pyramid.config.Configurator.add_view()` in order to decide what to do, and they have a chance to affect every view in the application.

Ordering View Drivers

By default, every new view driver is added between the `decorated_view` and `rendered_view` built-in drivers. It is possible to customize this ordering using the `over` and `under` options. Each option can use the names of other view drivers in order to specify an ordering. There should rarely be a reason to worry about the ordering of the drivers except when the driver depends on other operations in the view pipeline.

Both `over` and `under` may also be iterables of constraints. For either option, if one or more constraints was defined, at least one must be satisfied, else a `pyramid.exceptions.ConfigurationError` will be raised. This may be used to define fallback constraints if another driver is missing.

Two sentinel values exist, `pyramid.viewdrivers.INGRESS` and `pyramid.viewdrivers.VIEW`, which may be used when specifying constraints at the edges of the view pipeline. For example, to add a driver at the start of the pipeline you may use `under=INGRESS`.

It is not possible to add a view driver under the `mapped_view` as the *view mapper* is intimately tied to the signature of the user-defined *view callable*. If you simply need to know what the original view callable was, it can be found as `info.original_view` on the provided `pyramid.interfaces.IViewDriverInfo` object passed to every view driver.



The default constraints for any view driver are `over='rendered_view'` and `under='decorated_view'`. When escaping these constraints you must take care to avoid cyclic dependencies between drivers. For example, if you want to add a new view driver before `secured_view` then simply specifying `over='secured_view'` is not enough, because the default is also under `decorated_view` there will be an unsatisfiable cycle. You must specify a valid `under` constraint as well, such as `under=INGRESS` to fall between `INGRESS` and `secured_view` at the beginning of the view pipeline.

Pyramid Configuration Introspection

New in version 1.3.

When Pyramid starts up, each call to a *configuration directive* causes one or more *introspectable* objects to be registered with an *introspector*. The introspector can be queried by application code to obtain information about the configuration of the running application. This feature is useful for debug toolbars, command-line scripts which show some aspect of configuration, and for runtime reporting of startup-time configuration settings.

Using the Introspector

Here's an example of using Pyramid's introspector from within a view callable:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='bar')
5 def show_current_route_pattern(request):
6     introspector = request.registry.introspector
7     route_name = request.matched_route.name
8     route_intr = introspector.get('routes', route_name)
9     return Response(str(route_intr['pattern']))
```

This view will return a response that contains the “pattern” argument provided to the `add_route` method of the route which matched when the view was called. It uses the `pyramid.interfaces.IIntrospector.get()` method to return an introspectable in the category `routes` with a *discriminator* equal to the matched route name. It then uses the returned introspectable to obtain a “pattern” value.

The introspectable returned by the query methods of the introspector has methods and attributes described by `pyramid.interfaces.IIntrospectable`. In particular, the `get()`, `get_category()`, `categories()`, `categorized()`, and `related()` methods of an introspector can be used to query for introspectables.

Introspectable Objects

Introspectable objects are returned from query methods of an introspector. Each introspectable object implements the attributes and methods documented at `pyramid.interfaces.IIntrospectable`.

The important attributes shared by all introspectables are the following:

`title`

A human-readable text title describing the introspectable

`category_name`

A text category name describing the introspection category to which this introspectable belongs. It is often a plural if there are expected to be more than one introspectable registered within the category.

`discriminator`

A hashable object representing the unique value of this introspectable within its category.

`discriminator_hash`

The integer hash of the discriminator (useful in HTML links).

`type_name`

The text name of a subtype within this introspectable's category. If there is only one type name in this introspectable's category, this value will often be a singular version of the category name but it can be an arbitrary value.

`action_info`

An object describing the directive call site which caused this introspectable to be registered. It contains attributes described in *pyramid.interfaces.IActionInfo*.

Besides having the attributes described above, an introspectable is a dictionary-like object. An introspectable can be queried for data values via its `__getitem__`, `get`, `keys`, `values`, or `items` methods. For example:

```
1 route_intr = introspector.get('routes', 'edit_user')
2 pattern = route_intr['pattern']
```

Pyramid Introspection Categories

The list of concrete introspection categories provided by built-in Pyramid configuration directives follows. Add-on packages may supply other introspectables in categories not described here.

subscribers

Each introspectable in the `subscribers` category represents a call to `pyramid.config.Configurator.add_subscriber()` (or the decorator equivalent). Each will have the following data.

subscriber

The subscriber callable object (the resolution of the `subscriber` argument passed to `add_subscriber`).

interfaces

A sequence of interfaces (or classes) that are subscribed to (the resolution of the `ifaces` argument passed to `add_subscriber`).

derived_subscriber

A wrapper around the subscriber used internally by the system so it can call it with more than one argument if your original subscriber accepts only one.

predicates

The predicate objects created as the result of passing predicate arguments to `add_subscriber`.

derived_predicates

Wrappers around the predicate objects created as the result of passing predicate arguments to `add_subscriber` (to be used when predicates take only one value but must be passed more than one).

response adapters

Each introspectable in the `response adapters` category represents a call to `pyramid.config.Configurator.add_response_adapter()` (or a decorator equivalent). Each will have the following data.

adapter

The adapter object (the resolved adapter argument to `add_response_adapter`).

type

The resolved `type_or_iface` argument passed to `add_response_adapter`.

root factories

Each introspectable in the `root factories` category represents a call to `pyramid.config.Configurator.set_root_factory()` (or the `Configurator` constructor equivalent) or a factory argument passed to `pyramid.config.Configurator.add_route()`. Each will have the following data.

factory

The factory object (the resolved factory argument to `set_root_factory`).

route_name

The name of the route which will use this factory. If this is the *default* root factory (if it's registered during a call to `set_root_factory`), this value will be `None`.

session factory

Only one introspectable will exist in the `session factory` category. It represents a call to `pyramid.config.Configurator.set_session_factory()` (or the `Configurator` constructor equivalent). It will have the following data.

factory

The factory object (the resolved factory argument to `set_session_factory`).

request factory

Only one introspectable will exist in the `request factory` category. It represents a call to `pyramid.config.Configurator.set_request_factory()` (or the `Configurator` constructor equivalent). It will have the following data.

factory

The factory object (the resolved factory argument to `set_request_factory`).

locale negotiator

Only one introspectable will exist in the `locale negotiator` category. It represents a call to `pyramid.config.Configurator.set_locale_negotiator()` (or the `Configurator` constructor equivalent). It will have the following data.

negotiator

The factory object (the resolved negotiator argument to `set_locale_negotiator`).

renderer factories

Each introspectable in the `renderer factories` category represents a call to `pyramid.config.Configurator.add_renderer()` (or the `Configurator` constructor equivalent). Each will have the following data.

name

The name of the renderer (the value of the `name` argument to `add_renderer`).

factory

The factory object (the resolved factory argument to `add_renderer`).

routes

Each introspectable in the `routes` category represents a call to `pyramid.config.Configurator.add_route()`. Each will have the following data.

name

The name argument passed to `add_route`.

pattern

The pattern argument passed to `add_route`.

factory

The (resolved) factory argument passed to `add_route`.

xhr

The `xhr` argument passed to `add_route`.

request_method

The `request_method` argument passed to `add_route`.

request_methods

A sequence of request method names implied by the `request_method` argument passed to `add_route` or the value `None` if a `request_method` argument was not supplied.

path_info

The `path_info` argument passed to `add_route`.

request_param

The `request_param` argument passed to `add_route`.

header

The `header` argument passed to `add_route`.

accept

The `accept` argument passed to `add_route`.

traverse

The `traverse` argument passed to `add_route`.

custom_predicates

The `custom_predicates` argument passed to `add_route`.

pregenerator

The `pregenerator` argument passed to `add_route`.

static

The static argument passed to `add_route`.

`use_global_views`

The `use_global_views` argument passed to `add_route`.

object

The `pyramid.interfaces.IRoute` object that is used to perform matching and generation for this route.

authentication policy

There will be one and only one introspectable in the authentication policy category. It represents a call to the `pyramid.config.Configurator.set_authentication_policy()` method (or its Configurator constructor equivalent). It will have the following data.

policy

The policy object (the resolved policy argument to `set_authentication_policy`).

authorization policy

There will be one and only one introspectable in the authorization policy category. It represents a call to the `pyramid.config.Configurator.set_authorization_policy()` method (or its Configurator constructor equivalent). It will have the following data.

policy

The policy object (the resolved policy argument to `set_authorization_policy`).

default permission

There will be one and only one introspectable in the default permission category. It represents a call to the `pyramid.config.Configurator.set_default_permission()` method (or its Configurator constructor equivalent). It will have the following data.

value

The permission name passed to `set_default_permission`.

`default csrf options`

There will be one and only one introspectable in the `default csrf options` category. It represents a call to the `pyramid.config.Configurator.set_default_csrf_options()` method. It will have the following data.

`require_csrf`

The default value for `require_csrf` if left unspecified on calls to `pyramid.config.Configurator.add_view()`.

`token`

The name of the token searched in `request.POST` to find a valid CSRF token.

`header`

The name of the request header searched to find a valid CSRF token.

`safe_methods`

The list of HTTP methods considered safe and exempt from CSRF checks.

`views`

Each introspectable in the `views` category represents a call to `pyramid.config.Configurator.add_view()`. Each will have the following data.

`name`

The `name` argument passed to `add_view`.

`context`

The (resolved) `context` argument passed to `add_view`.

`containment`

The (resolved) `containment` argument passed to `add_view`.

`request_param`

The `request_param` argument passed to `add_view`.

`request_methods`

A sequence of request method names implied by the `request_method` argument passed to `add_view` or the value `None` if a `request_method` argument was not supplied.

`route_name`

The `route_name` argument passed to `add_view`.

`attr`

The `attr` argument passed to `add_view`.

`xhr`

The `xhr` argument passed to `add_view`.

`accept`

The `accept` argument passed to `add_view`.

`header`

The `header` argument passed to `add_view`.

`path_info`

The `path_info` argument passed to `add_view`.

`match_param`

The `match_param` argument passed to `add_view`.

`csrf_token`

The `csrf_token` argument passed to `add_view`.

`callable`

The (resolved) view argument passed to `add_view`. Represents the “raw” view callable.

`derived_callable`

The view callable derived from the `view` argument passed to `add_view`. Represents the view callable which Pyramid itself calls (wrapped in security and other wrappers).

`mapper`

The (resolved) `mapper` argument passed to `add_view`.

`decorator`

The (resolved) `decorator` argument passed to `add_view`.

`permissions`

Each introspectable in the `permissions` category represents a call to `pyramid.config.Configurator.add_view()` that has an explicit `permission` argument or a call to `pyramid.config.Configurator.set_default_permission()`. Each will have the following data.

`value`

The permission name passed to `add_view` or `set_default_permission`.

`templates`

Each introspectable in the `templates` category represents a call to `pyramid.config.Configurator.add_view()` that has a `renderer` argument which points to a template. Each will have the following data.

`name`

The renderer's name (a string).

`type`

The renderer's type (a string).

`renderer`

The `pyramid.interfaces.IRendererInfo` object which represents this template's renderer.

view mappers

Each introspectable in the `view mappers` category represents a call to `pyramid.config.Configurator.add_view()` that has an explicit `mapper` argument or a call to `pyramid.config.Configurator.set_view_mapper()`. Each will have the following data.

mapper

The (resolved) `mapper` argument passed to `add_view` or `set_view_mapper`.

asset overrides

Each introspectable in the `asset overrides` category represents a call to `pyramid.config.Configurator.override_asset()`. Each will have the following data.

to_override

The `to_override` argument (an asset spec) passed to `override_asset`.

override_with

The `override_with` argument (an asset spec) passed to `override_asset`.

translation directories

Each introspectable in the `translation directories` category represents an individual element in a `specs` argument passed to `pyramid.config.Configurator.add_translation_dirs()`. Each will have the following data.

directory

The absolute path of the translation directory.

spec

The asset specification passed to `add_translation_dirs`.

tweens

Each introspectable in the `tweens` category represents a call to `pyramid.config.Configurator.add_tween()`. Each will have the following data.

name

The dotted name to the tween factory as a string (passed as the `tween_factory` argument to `add_tween`).

factory

The (resolved) tween factory object.

type

`implicit` or `explicit` as a string.

under

The `under` argument passed to `add_tween` (a string).

over

The `over` argument passed to `add_tween` (a string).

static views

Each introspectable in the `static views` category represents a call to `pyramid.config.Configurator.add_static_view()`. Each will have the following data.

name

The `name` argument provided to `add_static_view`.

spec

A normalized version of the `spec` argument provided to `add_static_view`.

traversers

Each introspectable in the `traversers` category represents a call to `pyramid.config.Configurator.add_traverser()`. Each will have the following data.

iface

The (resolved) interface or class object that represents the return value of a root factory for which this traverser will be used.

adapter

The (resolved) traverser class.

resource url adapters

Each introspectable in the `resource url adapters` category represents a call to `pyramid.config.Configurator.add_resource_url_adapter()`. Each will have the following data.

adapter

The (resolved) resource URL adapter class.

resource_iface

The (resolved) interface or class object that represents the resource interface for which this URL adapter is registered.

request_iface

The (resolved) interface or class object that represents the request interface for which this URL adapter is registered.

Introspection in the Toolbar

The Pyramid debug toolbar (part of the `pyramid_debugtoolbar` package) provides a canned view of all registered introspectables and their relationships. It is currently under the “Global” tab in the main navigation, and it looks something like this:

Pyramid Debug Toolbar

localhost:6543/_debug_toolbar/34333932303138343438#

History Global Settings

Introspection Routes Settings Tweens Versions

Introspection

Permissions

permission <code>__no_permission_required__</code>	
value	<code>'__no_permission_required__'</code>

Source

Line 9 of file `/Users/stevepiercy/projects/hack-on-pyramid/scaffolds/scaffolds/__init__.py`:
`config.add_static_view('static', 'static', cache_max_age=3600)`

References

view object `<pyramid.static.static_view object at 0x105bed110>`

Renderer factories

renderer factory object <code><pyramid.renderers.JSON object at 0x1018aee50></code>	
factory	<code><pyramid.renderers.JSON object at 0x1018aee50></code>
name	<code>'json'</code>

Source

Line 17 of file `/Users/stevepiercy/projects/hack-on-pyramid/pyramid/config/rendering.py`:
`self.add_renderer(name, renderer)`

Disabling Introspection

You can disable Pyramid introspection by passing the flag `introspection=False` to the *Configurator* constructor in your application setup:

```
from pyramid.config import Configurator
config = Configurator(..., introspection=False)
```

When `introspection` is `False`, all introspectables generated by configuration directives are thrown away.

Extending an Existing Pyramid Application

If a Pyramid developer has obeyed certain constraints while building an application, a third party should be able to change the application's behavior without needing to modify its source code. The behavior of a Pyramid application that obeys certain constraints can be *overridden* or *extended* without modification.

We'll define some jargon here for the benefit of identifying the parties involved in such an effort.

Developer The original application developer.

Integrator Another developer who wishes to reuse the application written by the original application developer in an unanticipated context. They may also wish to modify the original application without changing the original application's source code.

The Difference Between “Extensible” and “Pluggable” Applications

Other web frameworks, such as *Django*, advertise that they allow developers to create “pluggable applications”. They claim that if you create an application in a certain way, it will be integratable in a sensible, structured way into another arbitrarily-written application or project created by a third-party developer.

Pyramid, as a platform, does not claim to provide such a feature. The platform provides no guarantee that you can create an application and package it up such that an arbitrary integrator can use it as a subcomponent in a larger Pyramid application or project. Pyramid does not mandate the constraints necessary for such a pattern to work satisfactorily. Because Pyramid is not very “opinionated”, developers are able to use wildly different patterns and technologies to build an application. A given Pyramid application may happen to be reusable by a particular third party integrator because the integrator and the original developer may share similar base technology choices (such as the use of a particular relational database or ORM). But the same application may not be reusable by a different developer, because they have made different technology choices which are incompatible with the original developer's.

As a result, the concept of a “pluggable application” is left to layers built above Pyramid, such as a “CMS” layer or “application server” layer. Such layers are apt to provide the necessary “opinions” (such as mandating a storage layer, a templating system, and a structured, well-documented pattern of registering that certain URLs map to certain bits of code) which makes the concept of a “pluggable application” possible. “Pluggable applications”, thus, should not plug into Pyramid itself but should instead plug into a system written atop Pyramid.

Although it does not provide for “pluggable applications”, Pyramid *does* provide a rich set of mechanisms which allows for the extension of a single existing application. Such features can be used by frameworks built using Pyramid as a base. All Pyramid applications may not be *pluggable*, but all Pyramid applications are *extensible*.

Rules for Building an Extensible Application

There is only one rule you need to obey if you want to build a maximally extensible Pyramid application: as a developer, you should factor any overridable *imperative configuration* you’ve created into functions which can be used via `pyramid.config.Configurator.include()`, rather than inlined as calls to methods of a *Configurator* within the main function in your application’s `__init__.py`. For example, rather than:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     config = Configurator()
5     config.add_view('myapp.views.view1', name='view1')
6     config.add_view('myapp.views.view2', name='view2')
```

You should move the calls to `add_view` outside of the (non-reusable) `if __name__ == '__main__'` block, and into a reusable function:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     config = Configurator()
5     config.include(add_views)
6
7 def add_views(config):
8     config.add_view('myapp.views.view1', name='view1')
9     config.add_view('myapp.views.view2', name='view2')
```

Doing this allows an integrator to maximally reuse the configuration statements that relate to your application by allowing them to selectively include or exclude the configuration functions you’ve created from an “override package”.

Alternatively you can use *ZCML* for the purpose of making configuration extensible and overridable. *ZCML* declarations that belong to an application can be overridden and extended by integrators as necessary in a similar fashion. If you use only *ZCML* to configure your application, it will automatically be maximally extensible without any manual effort. See `pyramid_zcml` for information about using *ZCML*.

Fundamental Plugpoints

The fundamental “plug points” of an application developed using Pyramid are *routes*, *views*, and *assets*. Routes are declarations made using the `pyramid.config.Configurator.add_route()`

method. Views are declarations made using the `pyramid.config.Configurator.add_view()` method. Assets are files that are accessed by Pyramid using the `pkg_resources` API such as static files and templates via a *asset specification*. Other directives and configurator methods also deal in routes, views, and assets. For example, the `add_handler` directive of the `pyramid_handlers` package adds a single route and some number of views.

Extending an Existing Application

The steps for extending an existing application depend largely on whether the application does or does not use configuration decorators or imperative code.

If the Application Has Configuration Decorations

You've inherited a Pyramid application which you'd like to extend or override that uses `pyramid.view.view_config` decorators or other *configuration decoration* decorators.

If you just want to *extend* the application, you can run a *scan* against the application's package, then add additional configuration that registers more views or routes.

```
1 if __name__ == '__main__':
2     config.scan('someotherpackage')
3     config.add_view('mypackage.views.myview', name='myview')
```

If you want to *override* configuration in the application, you *may* need to run `pyramid.config.Configurator.commit()` after performing the scan of the original package, then add additional configuration that registers more views or routes which perform overrides.

```
1 if __name__ == '__main__':
2     config.scan('someotherpackage')
3     config.commit()
4     config.add_view('mypackage.views.myview', name='myview')
```

Once this is done, you should be able to extend or override the application like any other (see *Extending the Application*).

You can alternatively just prevent a *scan* from happening by omitting any call to the `pyramid.config.Configurator.scan()` method. This will cause the decorators attached to objects in the target application to do nothing. At this point, you will need to convert all the configuration done in decorators into equivalent imperative configuration or ZCML, and add that configuration or ZCML to a separate Python package as described in *Extending the Application*.

Extending the Application

To extend or override the behavior of an existing application, you will need to create a new package which includes the configuration of the old package, and you'll perhaps need to create implementations of the types of things you'd like to override (such as views), to which they are referred within the original package.

The general pattern for extending an existing application looks something like this:

- Create a new Python package. The easiest way to do this is to create a new Pyramid application using the scaffold mechanism. See *Creating the Project* for more information.
- In the new package, create Python files containing views and other overridden elements, such as templates and static assets as necessary.
- Install the new package into the same Python environment as the original application (e.g., `$VENV/bin/pip install -e .` or `$VENV/bin/pip install .`).
- Change the main function in the new package's `__init__.py` to include the original Pyramid application's configuration functions via `pyramid.config.Configurator.include()` statements or a *scan*.
- Wire the new views and assets created in the new package up using imperative registrations within the main function of the `__init__.py` file of the new application. This wiring should happen *after* including the configuration functions of the old application. These registrations will extend or override any registrations performed by the original application. See *Overriding Views*, *Overriding Routes*, and *Overriding Assets*.

Overriding Views

The *view configuration* declarations that you make which *override* application behavior will usually have the same *view predicate* attributes as the original that you wish to override. These `<view>` declarations will point at “new” view code in the override package that you’ve created. The new view code itself will usually be copy-and-paste copies of view callables from the original application with slight tweaks.

For example, if the original application has the following `configure_views` configuration method:

```
1 def configure_views(config):  
2     config.add_view('theoriginalapp.views.theview', name='theview')
```

You can override the first view configuration statement made by `configure_views` within the override package, after loading the original configuration function:

```
1 from pyramid.config import Configurator
2 from originalapp import configure_views
3
4 if __name__ == '__main__':
5     config = Configurator()
6     config.include(configure_views)
7     config.add_view('theoverrideapp.views.theview', name='theview')
```

In this case, the `theoriginalapp.views.theview` view will never be executed. Instead, a new view, `theoverrideapp.views.theview` will be executed when request circumstances dictate.

A similar pattern can be used to *extend* the application with `add_view` declarations. Just register a new view against some other set of predicates to make sure the URLs it implies are available on some other page rendering.

Overriding Routes

Route setup is currently typically performed in a sequence of ordered calls to `add_route()`. Because these calls are ordered relative to each other, and because this ordering is typically important, you should retain their relative ordering when performing an override. Typically this means *copying* all the `add_route` statements into the override package's file and changing them as necessary. Then exclude any `add_route` statements from the original application.

Overriding Assets

Assets are files on the filesystem that are accessible within a Python *package*. An entire chapter is devoted to assets: *Static Assets*. Within this chapter is a section named *Overriding Assets*. This section of that chapter describes in detail how to override package assets with other assets by using the `pyramid.config.Configurator.override_asset()` method. Add such `override_asset` calls to your override package's `__init__.py` to perform overrides.

Advanced Configuration

To support application extensibility, the Pyramid *Configurator* by default detects configuration conflicts and allows you to include configuration imperatively from other packages or modules. It also by default performs configuration in two separate phases. This allows you to ignore relative configuration statement ordering in some circumstances.

Conflict Detection

Here's a familiar example of one of the simplest Pyramid applications, configured imperatively:

```

1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 if __name__ == '__main__':
9     config = Configurator()
10    config.add_view(hello_world)
11    app = config.make_wsgi_app()
12    server = make_server('0.0.0.0', 8080, app)
13    server.serve_forever()

```

When you start this application, all will be OK. However, what happens if we try to add another view to the configuration with the same set of *predicate* arguments as one we've already added?

```

1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 def goodbye_world(request):
9     return Response('Goodbye world!')
10
11 if __name__ == '__main__':
12     config = Configurator()
13
14     config.add_view(hello_world, name='hello')
15
16     # conflicting view configuration
17     config.add_view(goodbye_world, name='hello')
18
19     app = config.make_wsgi_app()
20     server = make_server('0.0.0.0', 8080, app)
21     server.serve_forever()

```

The application now has two conflicting view configuration statements. When we try to start it again, it won't start. Instead we'll receive a traceback that ends something like this:

```
1  Traceback (most recent call last):
2    File "app.py", line 12, in <module>
3      app = config.make_wsgi_app()
4    File "pyramid/config.py", line 839, in make_wsgi_app
5      self.commit()
6    File "pyramid/pyramid/config.py", line 473, in commit
7      self._ctx.execute_actions()
8      ... more code ...
9  pyramid.exceptions.ConfigurationConflictError:
10     Conflicting configuration actions
11  For: ('view', None, '', None, <InterfaceClass pyramid.interfaces.IView>,
12       None, None, None, None, None, False, None, None, None)
13  Line 14 of file app.py in <module>: 'config.add_view(hello_world) '
14  Line 17 of file app.py in <module>: 'config.add_view(goodbye_world) '
```

This traceback is trying to tell us:

- We’ve got conflicting information for a set of view configuration statements (The `For:` line).
- There are two statements which conflict, shown beneath the `For:` line: `config.add_view(hello_world, 'hello')` on line 14 of `app.py`, and `config.add_view(goodbye_world, 'hello')` on line 17 of `app.py`.

These two configuration statements are in conflict because we’ve tried to tell the system that the set of *predicate* values for both view configurations are exactly the same. Both the `hello_world` and `goodbye_world` views are configured to respond under the same set of circumstances. This circumstance, the *view name* represented by the `name=predicate`, is `hello`.

This presents an ambiguity that Pyramid cannot resolve. Rather than allowing the circumstance to go unreported, by default Pyramid raises a `ConfigurationConflictError` error and prevents the application from running.

Conflict detection happens for any kind of configuration: imperative configuration or configuration that results from the execution of a *scan*.

Manually Resolving Conflicts

There are a number of ways to manually resolve conflicts: by changing registrations to not conflict, by strategically using `pyramid.config.Configurator.commit()`, or by using an “autocommitting” configurator.

The Right Thing

The most correct way to resolve conflicts is to “do the needful”: change your configuration code to not have conflicting configuration statements. The details of how this is done depends entirely on the configuration statements made by your application. Use the detail provided in the `ConfigurationConflictError` to track down the offending conflicts and modify your configuration code accordingly.

If you’re getting a conflict while trying to extend an existing application, and that application has a function which performs configuration like this one:

```
1 def add_routes(config):
2     config.add_route(...)
```

Don’t call this function directly with `config` as an argument. Instead, use `pyramid.config.Configurator.include()`:

```
1 config.include(add_routes)
```

Using `include()` instead of calling the function directly provides a modicum of automated conflict resolution, with the configuration statements you define in the calling code overriding those of the included function.

See also:

See also *Automatic Conflict Resolution* and *Including Configuration from External Sources*.

Using `config.commit()`

You can manually commit a configuration by using the `commit()` method between configuration calls. For example, we prevent conflicts from occurring in the application we examined previously as the result of adding a `commit`. Here’s the application that generates conflicts:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
```



```
7
8 def goodbye_world(request):
9     return Response('Goodbye world!')
10
11 if __name__ == '__main__':
12     config = Configurator()
13
14     config.add_view(hello_world, name='hello')
15
16     # conflicting view configuration
17     config.add_view(goodbye_world, name='hello')
18
19     app = config.make_wsgi_app()
20     server = make_server('0.0.0.0', 8080, app)
21     server.serve_forever()
```

We can prevent the two `add_view` calls from conflicting by issuing a call to `commit()` between them:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 def goodbye_world(request):
9     return Response('Goodbye world!')
10
11 if __name__ == '__main__':
12     config = Configurator()
13
14     config.add_view(hello_world, name='hello')
15
16     config.commit() # commit any pending configuration actions
17
18     # no-longer-conflicting view configuration
19     config.add_view(goodbye_world, name='hello')
20
21     app = config.make_wsgi_app()
22     server = make_server('0.0.0.0', 8080, app)
23     server.serve_forever()
```

In the above example we've issued a call to `commit()` between the two `add_view` calls. `commit()` will execute any pending configuration statements.

Calling `commit()` is safe at any time. It executes all pending configuration actions and leaves the configuration action list “clean”.

Note that `commit()` has no effect when you’re using an *autocommitting* configurator (see *Using an Autocommitting Configurator*).

Using an Autocommitting Configurator

You can also use a heavy hammer to circumvent conflict detection by using a configurator constructor parameter: `autocommit=True`. For example:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     config = Configurator(autocommit=True)
```

When the `autocommit` parameter passed to the `Configurator` is `True`, conflict detection (and *Two-Phase Configuration*) is disabled. Configuration statements will be executed immediately, and succeeding statements will override preceding ones.

`commit()` has no effect when `autocommit` is `True`.

If you use a `Configurator` in code that performs unit testing, it’s usually a good idea to use an auto-committing `Configurator`, because you are usually unconcerned about conflict detection or two-phase configuration in test code.

Automatic Conflict Resolution

If your code uses the `include()` method to include external configuration, some conflicts are automatically resolved. Configuration statements that are made as the result of an “include” will be overridden by configuration statements that happen within the caller of the “include” method.

Automatic conflict resolution supports this goal. If a user wants to reuse a Pyramid application, and they want to customize the configuration of this application without hacking its code “from outside”, they can “include” a configuration function from the package and override only some of its configuration statements within the code that does the include. No conflicts will be generated by configuration statements within the code that does the including, even if configuration statements in the included code would conflict if it was moved “up” to the calling code.

Methods Which Provide Conflict Detection

These are the methods of the configurator which provide conflict detection:

```
add_view(),    add_route(),    add_renderer(),    add_request_method(),
set_request_factory(), set_session_factory(), set_request_property(),
set_root_factory(), set_view_mapper(), set_authentication_policy(),
set_authorization_policy(),                                set_locale_negotiator(),
set_default_permission(), add_traverser(), add_resource_url_adapter(), and
add_response_adapter().
```

`add_static_view()` also indirectly provides conflict detection, because it's implemented in terms of the conflict-aware `add_route` and `add_view` methods.

Including Configuration from External Sources

Some application programmers will factor their configuration code in such a way that it is easy to reuse and override configuration statements. For example, such a developer might factor out a function used to add routes to their application:

```
1 def add_routes(config):
2     config.add_route(...)
```

Rather than calling this function directly with `config` as an argument, instead use `pyramid.config.Configurator.include()`:

```
1 config.include(add_routes)
```

Using `include` rather than calling the function directly will allow *Automatic Conflict Resolution* to work.

`include()` can also accept a *module* as an argument:

```
1 import myapp
2
3 config.include(myapp)
```

For this to work properly, the `myapp` module must contain a callable with the special name `includeme`, which should perform configuration (like the `add_routes` callable we showed above as an example).

`include()` can also accept a *dotted Python name* to a function or a module.



See The `<include>` Tag for a declarative alternative to the `include()` method.

Two-Phase Configuration

When a non-autocommitting *Configurator* is used to do configuration (the default), configuration execution happens in two phases. In the first phase, “eager” configuration actions (actions that must happen before all others, such as registering a renderer) are executed, and *discriminators* are computed for each of the actions that depend on the result of the eager actions. In the second phase, the discriminators of all actions are compared to do conflict detection.

Due to this, for configuration methods that have no internal ordering constraints, execution order of configuration method calls is not important. For example, the relative ordering of `add_view()` and `add_renderer()` is unimportant when a non-autocommitting configurator is used. This code snippet:

```
1 config.add_view('some.view', renderer='path_to_custom/renderer.rn')
2 config.add_renderer('.rn', SomeCustomRendererFactory)
```

Has the same result as:

```
1 config.add_renderer('.rn', SomeCustomRendererFactory)
2 config.add_view('some.view', renderer='path_to_custom/renderer.rn')
```

Even though the view statement depends on the registration of a custom renderer, due to two-phase configuration, the order in which the configuration statements are issued is not important. `add_view` will be able to find the `.rn` renderer even if `add_renderer` is called after `add_view`.

The same is untrue when you use an *autocommitting* configurator (see *Using an Autocommitting Configurator*). When an autocommitting configurator is used, two-phase configuration is disabled, and configuration statements must be ordered in dependency order.

Some configuration methods, such as `add_route()` have internal ordering constraints: the routes they imply require relative ordering. Such ordering constraints are not absolved by two-phase configuration. Routes are still added in configuration execution order.

More Information

For more information, see the article *A Whirlwind Tour of Advanced Configuration Tactics in the Pyramid Community Cookbook*.

Extending Pyramid Configuration

Pyramid allows you to extend its Configurator with custom directives. Custom directives can use other directives, they can add a custom *action*, they can participate in *conflict resolution*, and they can provide some number of *introspectable* objects.

Adding Methods to the Configurator via `add_directive`

Framework extension writers can add arbitrary methods to a *Configurator* by using the `pyramid.config.Configurator.add_directive()` method of the configurator. Using `add_directive()` makes it possible to extend a Pyramid configurator in arbitrary ways, and allows it to perform application-specific tasks more succinctly.

The `add_directive()` method accepts two positional arguments: a method name and a callable object. The callable object is usually a function that takes the configurator instance as its first argument and accepts other arbitrary positional and keyword arguments. For example:

```
1 from pyramid.events import NewRequest
2 from pyramid.config import Configurator
3
4 def add_newrequest_subscriber(config, subscriber):
5     config.add_subscriber(subscriber, NewRequest)
6
7 if __name__ == '__main__':
8     config = Configurator()
9     config.add_directive('add_newrequest_subscriber',
10                          add_newrequest_subscriber)
```

Once `add_directive()` is called, a user can then call the added directive by its given name as if it were a built-in method of the *Configurator*:

```
1 def mysubscriber(event):
2     print(event.request)
3
4 config.add_newrequest_subscriber(mysubscriber)
```

A call to `add_directive()` is often “hidden” within an `includeme` function within a “frameworky” package meant to be included as per *Including Configuration from External Sources* via `include()`. For example, if you put this code in a package named `pyramid_subscriberhelpers`:

```
1 def includeme(config):
2     config.add_directive('add_newrequest_subscriber',
3                           add_newrequest_subscriber)
```

The user of the add-on package `pyramid_subscriberhelpers` would then be able to install it and subsequently do:

```
1 def mysubscriber(event):
2     print(event.request)
3
4 from pyramid.config import Configurator
5 config = Configurator()
6 config.include('pyramid_subscriberhelpers')
7 config.add_newrequest_subscriber(mysubscriber)
```

Using `config.action` in a Directive

If a custom directive can't do its work exclusively in terms of existing configurator methods (such as `pyramid.config.Configurator.add_subscriber()` as above), the directive may need to make use of the `pyramid.config.Configurator.action()` method. This method adds an entry to the list of “actions” that Pyramid will attempt to process when `pyramid.config.Configurator.commit()` is called. An action is simply a dictionary that includes a *discriminator*, possibly a callback function, and possibly other metadata used by Pyramid's action system.

Here's an example directive which uses the “action” method:

```
1 def add_jammyjam(config, jammyjam):
2     def register():
3         config.registry.jammyjam = jammyjam
4         config.action('jammyjam', register)
5
6 if __name__ == '__main__':
7     config = Configurator()
8     config.add_directive('add_jammyjam', add_jammyjam)
```

Fancy, but what does it do? The action method accepts a number of arguments. In the above directive named `add_jammyjam`, we call `action()` with two arguments: the string `jammyjam` is passed as the first argument named *discriminator*, and the closure function named `register` is passed as the second argument named *callable*.

When the `action()` method is called, it appends an action to the list of pending configuration actions. All pending actions with the same discriminator value are potentially in conflict with one another (see *Conflict Detection*). When the `commit()` method of the Configurator is called (either explicitly or as the result of calling `make_wsgi_app()`), conflicting actions are potentially automatically resolved as per *Automatic Conflict Resolution*. If a conflict cannot be automatically resolved, a `pyramid.exceptions.ConfigurationConflictError` is raised and application startup is prevented.

In our above example, therefore, if a consumer of our `add_jammyjam` directive did this:

```
config.add_jammyjam('first')
config.add_jammyjam('second')
```

When the action list was committed resulting from the set of calls above, our user's application would not start, because the discriminators of the actions generated by the two calls are in direct conflict. Automatic conflict resolution cannot resolve the conflict (because no `config.include` is involved), and the user provided no intermediate `pyramid.config.Configurator.commit()` call between the calls to `add_jammyjam` to ensure that the successive calls did not conflict with each other.

This demonstrates the purpose of the discriminator argument to the action method: it's used to indicate a uniqueness constraint for an action. Two actions with the same discriminator will conflict unless the conflict is automatically or manually resolved. A discriminator can be any hashable object, but it is generally a string or a tuple. *You use a discriminator to declaratively ensure that the user doesn't provide ambiguous configuration statements.*

But let's imagine that a consumer of `add_jammyjam` used it in such a way that no configuration conflicts are generated.

```
config.add_jammyjam('first')
```

What happens now? When the `add_jammyjam` method is called, an action is appended to the pending actions list. When the pending configuration actions are processed during `commit()`, and no conflicts occur, the *callable* provided as the second argument to the `action()` method within `add_jammyjam` is called with no arguments. The callable in `add_jammyjam` is the `register` closure function. It simply sets the value `config.registry.jammyjam` to whatever the user passed in as the `jammyjam` argument to the `add_jammyjam` function. Therefore, the result of the user's call to our directive will set the `jammyjam` attribute of the registry to the string `first`. *A callable is used by a directive to defer the result of a user's call to the directive until conflict detection has had a chance to do its job.*

Other arguments exist to the `action()` method, including `args`, `kw`, `order`, and `introspectables`.

`args` and `kw` exist as values, which if passed will be used as arguments to the callable function when it is called back. For example, our directive might use them like so:

```

1 def add_jammyjam(config, jammyjam):
2     def register(*arg, **kw):
3         config.registry.jammyjam_args = arg
4         config.registry.jammyjam_kw = kw
5         config.registry.jammyjam = jammyjam
6         config.action('jammyjam', register, args=('one',), kw={'two':'two'})

```

In the above example, when this directive is used to generate an action, and that action is committed, `config.registry.jammyjam_args` will be set to `('one',)` and `config.registry.jammyjam_kw` will be set to `{ 'two': 'two' }`. `args` and `kw` are honestly not very useful when your callable is a closure function, because you already usually have access to every local in the directive without needing them to be passed back. They can be useful, however, if you don't use a closure as a callable.

`order` is a crude order control mechanism. `order` defaults to the integer 0; it can be set to any other integer. All actions that share an order will be called before other actions that share a higher order. This makes it possible to write a directive with callable logic that relies on the execution of the callable of another directive being done first. For example, Pyramid's `pyramid.config.Configurator.add_view()` directive registers an action with a higher order than the `pyramid.config.Configurator.add_route()` method. Due to this, the `add_view` method's callable can assume that, if a `route_name` was passed to it, that a route by this name was already registered by `add_route`, and if such a route has not already been registered, it's a configuration error (a view that names a nonexistent route via its `route_name` parameter will never be called).

Changed in version 1.6: As of Pyramid 1.6 it is possible for one action to invoke another. See *Ordering Actions* for more information.

Finally, `introspectables` is a sequence of *introspectable* objects. You can pass a sequence of introspectables to the `action()` method, which allows you to augment Pyramid's configuration introspection system.

Ordering Actions

In Pyramid every *action* has an inherent ordering relative to other actions. The logic within actions is deferred until a call to `pyramid.config.Configurator.commit()` (which is automatically invoked by `pyramid.config.Configurator.make_wsgi_app()`). This means you may call `config.add_view(route_name='foo')` **before** `config.add_route('foo', '/foo')` because nothing actually happens until commit-time. During a commit cycle, conflicts are resolved, and actions are ordered and executed.

By default, almost every action in Pyramid has an order of `pyramid.config.PHASE3_CONFIG`. Every action within the same order-level will be executed in the order it was called. This means that if an

action must be reliably executed before or after another action, the `order` must be defined explicitly to make this work. For example, views are dependent on routes being defined. Thus the action created by `pyramid.config.Configurator.add_route()` has an order of `pyramid.config.PHASE2_CONFIG`.

Pre-defined Phases

`pyramid.config.PHASE0_CONFIG`

- This phase is reserved for developers who want to execute actions prior to Pyramid's core directives.

`pyramid.config.PHASE1_CONFIG`

- `pyramid.config.Configurator.add_renderer()`
- `pyramid.config.Configurator.add_route_predicate()`
- `pyramid.config.Configurator.add_subscriber_predicate()`
- `pyramid.config.Configurator.add_view_predicate()`
- `pyramid.config.Configurator.add_view_deriver()`
- `pyramid.config.Configurator.set_authorization_policy()`
- `pyramid.config.Configurator.set_default_csrf_options()`
- `pyramid.config.Configurator.set_default_permission()`
- `pyramid.config.Configurator.set_view_mapper()`

`pyramid.config.PHASE2_CONFIG`

- `pyramid.config.Configurator.add_route()`
- `pyramid.config.Configurator.set_authentication_policy()`

`pyramid.config.PHASE3_CONFIG`

- The default for all builtin or custom directives unless otherwise specified.

Calling Actions from Actions

New in version 1.6.

Pyramid's configurator allows actions to be added during a commit-cycle as long as they are added to the current or a later `order` phase. This means that your custom action can defer decisions until commit-time and then do things like invoke `pyramid.config.Configurator.add_route()`. It can also provide better conflict detection if your addon needs to call more than one other action.

For example, let's make an addon that invokes `add_route` and `add_view`, but we want it to conflict with any other call to our addon:

```
1 from pyramid.config import PHASE0_CONFIG
2
3 def includeme(config):
4     config.add_directive('add_auto_route', add_auto_route)
5
6 def add_auto_route(config, name, view):
7     def register():
8         config.add_view(route_name=name, view=view)
9         config.add_route(name, '/' + name)
10    config.action(('auto route', name), register, order=PHASE0_CONFIG)
```

Now someone else can use your addon and be informed if there is a conflict between this route and another, or two calls to `add_auto_route`. Notice how we had to invoke our action **before** `add_view` or `add_route`. If we tried to invoke this afterward, the subsequent calls to `add_view` and `add_route` would cause conflicts because that phase had already been executed, and the configurator cannot go back in time to add more views during that commit-cycle.

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     config = Configurator()
5     config.include('auto_route_addon')
6     config.add_auto_route('foo', my_view)
7
8 def my_view(request):
9     return request.response
```

Adding Configuration Introspection

New in version 1.3.

Pyramid provides a configuration introspection system that can be used by debugging tools to provide visibility into the configuration of a running application.

All built-in Pyramid directives (such as `pyramid.config.Configurator.add_view()` and `pyramid.config.Configurator.add_route()`) register a set of introspectables when called. For example, when you register a view via `add_view`, the directive registers at least one introspectable: an introspectable about the view registration itself, providing human-consumable values for the arguments passed into it. You can later use the introspection query system to determine whether a particular view uses a renderer, or whether a particular view is limited to a particular request method, or against which routes a particular view is registered. The Pyramid “debug toolbar” makes use of the introspection system in various ways to display information to Pyramid developers.

Introspection values are set when a sequence of *introspectable* objects is passed to the `action()` method. Here’s an example of a directive which uses introspectables:

```
1 def add_jammyjam(config, value):
2     def register():
3         config.registry.jammyjam = value
4         intr = config.introspectable(category_name='jammyjams',
5                                     discriminator='jammyjam',
6                                     title='a jammyjam',
7                                     type_name=None)
8         intr['value'] = value
9         config.action('jammyjam', register, introspectables=(intr,))
10
11 if __name__ == '__main__':
12     config = Configurator()
13     config.add_directive('add_jammyjam', add_jammyjam)
```

If you notice, the above directive uses the `introspectable` attribute of a `Configurator` (`pyramid.config.Configurator.introspectable`) to create an introspectable object. The introspectable object’s constructor requires at least four arguments: the `category_name`, the `discriminator`, the `title`, and the `type_name`.

The `category_name` is a string representing the logical category for this introspectable. Usually the `category_name` is a pluralization of the type of object being added via the action.

The `discriminator` is a value unique **within the category** (unlike the action discriminator, which must be unique within the entire set of actions). It is typically a string or tuple representing the values unique to this introspectable within the category. It is used to generate links and as part of a relationship-forming target for other introspectables.

The `title` is a human-consumable string that can be used by introspection system frontends to show a friendly summary of this introspectable.

The `type_name` is a value that can be used to subtype this introspectable within its category for sorting and presentation purposes. It can be any value.

An introspectable is also dictionary-like. It can contain any set of key/value pairs, typically related to the arguments passed to its related directive. While the `category_name`, `discriminator`, `title`, and `type_name` are *metadata* about the introspectable, the values provided as key/value pairs are the actual data provided by the introspectable. In the above example, we set the `value` key to the value of the `value` argument passed to the directive.

Our directive above mutates the introspectable, and passes it in to the `action` method as the first element of a tuple as the value of the `introspectable` keyword argument. This associates this introspectable with the action. Introspection tools will then display this introspectable in their index.

Introspectable Relationships

Two introspectables may have relationships between each other.

```

1 def add_jammyjam(config, value, template):
2     def register():
3         config.registry.jammyjam = (value, template)
4         intr = config.introspectable(category_name='jammyjams',
5                                     discriminator='jammyjam',
6                                     title='a jammyjam',
7                                     type_name=None)
8
9         intr['value'] = value
10        tpl_intr = config.introspectable(category_name='jammyjam templates',
11                                        discriminator=template,
12                                        title=template,
13                                        type_name=None)
14
15        tpl_intr['value'] = template
16        intr.relate('jammyjam templates', template)
17        config.action('jammyjam', register, introspectables=(intr, tpl_intr))
18
19 if __name__ == '__main__':
20     config = Configurator()
21     config.add_directive('add_jammyjam', add_jammyjam)

```

In the above example, the `add_jammyjam` directive registers two introspectables: the first is related to the `value` passed to the directive, and the second is related to the `template` passed to the directive. If you believe a concept within a directive is important enough to have its own introspectable, you can cause the same directive to register more than one introspectable, registering one introspectable for the “main idea” and another for a related concept.

The call to `intr.relate` above (`pyramid.interfaces.IIntrospectable.relate()`) is passed two arguments: a category name and a directive. The example above effectively indicates that the directive wishes to form a relationship between the `intr` introspectable and the `tmpl_intr` introspectable; the arguments passed to `relate` are the category name and discriminator of the `tmpl_intr` introspectable.

Relationships need not be made between two introspectables created by the same directive. Instead a relationship can be formed between an introspectable created in one directive and another introspectable created in another by calling `relate` on either side with the other directive's category name and discriminator. An error will be raised at configuration commit time if you attempt to relate an introspectable with another nonexistent introspectable, however.

Introspectable relationships will show up in frontend system renderings of introspection values. For example, if a view registration names a route name, the introspectable related to the view callable will show a reference to the route to which it relates and vice versa.

Creating Pyramid Scaffolds

You can extend Pyramid by creating a *scaffold* template. A scaffold template is useful if you'd like to distribute a customizable configuration of Pyramid to other users. Once you've created a scaffold, and someone has installed the distribution that houses the scaffold, they can use the `pcreate` script to create a custom version of your scaffold's template. Pyramid itself uses scaffolds to allow people to bootstrap new projects. For example, `pcreate -s alchemy MyStuff` causes Pyramid to render the `alchemy` scaffold template to the `MyStuff` directory.

Basics

A scaffold template is just a bunch of source files and directories on disk. A small definition class points at this directory. It is in turn pointed at by a *setuptools* "entry point" which registers the scaffold so it can be found by the `pcreate` command.

To create a scaffold template, create a Python *distribution* to house the scaffold which includes a `setup.py` that relies on the *setuptools* package. See *Packaging and Distributing Projects* for more information about how to do this. For example, we'll pretend the distribution you create is named `CoolExtension`, and it has a package directory within it named `coolextension`.

Once you've created the distribution, put a "scaffolds" directory within your distribution's package directory, and create a file within that directory named `__init__.py` with something like the following:

```

1 # CoolExtension/coolextension/scaffolds/__init__.py
2
3 from pyramid.scaffolds import PyramidTemplate
4
5 class CoolExtensionTemplate(PyramidTemplate):
6     _template_dir = 'coolextension_scaffold'
7     summary = 'My cool extension'

```

Once this is done, within the `scaffolds` directory, create a template directory. Our example used a template directory named `coolextension_scaffold`.

As you create files and directories within the template directory, note that:

- Files which have a name which are suffixed with the value `_tmpl` will be rendered, and replacing any instance of the literal string `{{var}}` with the string value of the variable named `var` provided to the scaffold.
- Files and directories with filenames that contain the string `+var+` will have that string replaced with the value of the `var` variable provided to the scaffold.
- Files that start with a dot (e.g., `.env`) are ignored and will not be copied over to the destination directory. If you want to include a file with a leading dot, then you must replace the dot with `+dot+` (e.g., `+dot+env`).

Otherwise, files and directories which live in the template directory will be copied directly without modification to the `pcreate` output location.

The variables provided by the default `PyramidTemplate` include `project` (the project name provided by the user as an argument to `pcreate`), `package` (a lowercasing and normalizing of the project name provided by the user), `random_string` (a long random string), and `package_logger` (the name of the package’s logger).

See Pyramid’s “scaffolds” package (<https://github.com/Pylons/pyramid/tree/master/pyramid/scaffolds>) for concrete examples of scaffold directories (`zodb`, `alchemy`, and `starter`, for example).

After you’ve created the template directory, add the following to the `entry_points` value of your distribution’s `setup.py`:

```

[pyramid.scaffold]
coolextension=coolextension.scaffolds:CoolExtensionTemplate

```

For example:

```
def setup(
    ...
    entry_points = """\
        [pyramid.scaffold]
        cooextension=cooextension.scaffolds:CoolExtensionTemplate
    """
)
```

Run your distribution's `setup.py develop` or `setup.py install` command. After that, you should be able to see your scaffolding template listed when you run `pcreate -l`. It will be named `cooextension` because that's the name we gave it in the entry point setup. Running `pcreate -s cooextension MyStuff` will then render your scaffold to an output directory named `MyStuff`.

See the module documentation for `pyramid.scaffolds` for information about the API of the `pyramid.scaffolds.Template` class and related classes. You can override methods of this class to get special behavior.

Supporting Older Pyramid Versions

Because different versions of Pyramid handled scaffolding differently, if you want to have extension scaffolds that can work across Pyramid 1.0.X, 1.1.X, 1.2.X and 1.3.X, you'll need to use something like this bit of horror while defining your scaffold template:

```
1 try: # pyramid 1.0.X
2     # "pyramid.paster.paste_script..." doesn't exist past 1.0.X
3     from pyramid.paster import paste_script_template_renderer
4     from pyramid.paster import PyramidTemplate
5 except ImportError:
6     try: # pyramid 1.1.X, 1.2.X
7         # trying to import "paste_script_template_renderer" fails on 1.3.X
8         from pyramid.scaffolds import paste_script_template_renderer
9         from pyramid.scaffolds import PyramidTemplate
10    except ImportError: # pyramid >=1.3a2
11        paste_script_template_renderer = None
12        from pyramid.scaffolds import PyramidTemplate
13
14 class CoolExtensionTemplate(PyramidTemplate):
15     _template_dir = 'cooextension_scaffold'
16     summary = 'My cool extension'
17     template_renderer = staticmethod(paste_script_template_renderer)
```

And then in the `setup.py` of the package that contains your scaffold, define the template as a target of both `paste.paster_create_template` (for `paster create`) and `pyramid.scaffold` (for `pcreate`).

```
[paste.paster_create_template]
coolextension=coolextension.scaffolds:CoolExtensionTemplate
[pyramid.scaffold]
coolextension=coolextension.scaffolds:CoolExtensionTemplate
```

Doing this hideousness will allow your scaffold to work as a `paster create` target (under 1.0, 1.1, or 1.2) or as a `pcreate` target (under 1.3). If an invoker tries to run `paster create` against a scaffold defined this way under 1.3, an error is raised instructing them to use `pcreate` instead.

If you want to support Pyramid 1.3 only, it's much cleaner, and the API is stable:

```
1 from pyramid.scaffolds import PyramidTemplate
2
3 class CoolExtensionTemplate(PyramidTemplate):
4     _template_dir = 'coolextension_scaffold'
5     summary = 'My cool_extension'
```

You only need to specify a `paste.paster_create_template` entry point target in your `setup.py` if you want your scaffold to be consumable by users of Pyramid 1.0, 1.1, or 1.2. To support only 1.3, specifying only the `pyramid.scaffold` entry point is good enough. If you want to support both `paster create` and `pcreate` (meaning you want to support Pyramid 1.2 and some older version), you'll need to define both.

Examples

Existing third-party distributions which house scaffolding are available via *PyPI*. The `pyramid_jqm`, `pyramid_zcml`, and `pyramid_jinja2` packages house scaffolds. You can install and examine these packages to see how they work in the quest to develop your own scaffolding.

Upgrading Pyramid

When a new version of Pyramid is released, it will sometimes deprecate a feature or remove a feature that was deprecated in an older release. When features are removed from Pyramid, applications that depend on those features will begin to break. This chapter explains how to ensure your Pyramid applications keep working when you upgrade the Pyramid version you're using.

About Release Numbering

Conventionally, application version numbering in Python is described as `major.minor.micro`. If your Pyramid version is “1.2.3”, it means you’re running a version of Pyramid with the major version “1”, the minor version “2” and the micro version “3”. A “major” release is one that increments the first-dot number; 2.X.X might follow 1.X.X. A “minor” release is one that increments the second-dot number; 1.3.X might follow 1.2.X. A “micro” release is one that increments the third-dot number; 1.2.3 might follow 1.2.2. In general, micro releases are “bugfix-only”, and contain no new features, minor releases contain new features but are largely backwards compatible with older versions, and a major release indicates a large set of backwards incompatibilities.

The Pyramid core team is conservative when it comes to removing features. We don’t remove features unnecessarily, but we’re human and we make mistakes which cause some features to be evolutionary dead ends. Though we are willing to support dead-end features for some amount of time, some eventually have to be removed when the cost of supporting them outweighs the benefit of keeping them around, because each feature in Pyramid represents a certain documentation and maintenance burden.

Deprecation and removal policy

When a feature is scheduled for removal from Pyramid or any of its official add-ons, the core development team takes these steps:

- Using the feature will begin to generate a *DeprecationWarning*, indicating the version in which the feature became deprecated.
- A note is added to the documentation indicating that the feature is deprecated.
- A note is added to the *Pyramid Change History* about the deprecation.

When a deprecated feature is eventually removed:

- The feature is removed.
- A note is added to the *Pyramid Change History* about the removal.

Features are never removed in *micro* releases. They are only removed in minor and major releases. Deprecated features are kept around for at least *three* minor releases from the time the feature became deprecated. Therefore, if a feature is added in Pyramid 1.0, but it’s deprecated in Pyramid 1.1, it will be kept around through all 1.1.X releases, all 1.2.X releases and all 1.3.X releases. It will finally be removed in the first 1.4.X release.

Sometimes features are “docs-deprecated” instead of formally deprecated. This means that the feature will be kept around indefinitely, but it will be removed from the documentation or a note will be added to the documentation telling folks to use some other newer feature. This happens when the cost of keeping an old feature around is very minimal and the support and documentation burden is very low. For example, we might rename a function that is an API without changing the arguments it accepts. In this case, we’ll often rename the function, and change the docs to point at the new function name, but leave around a backwards compatibility alias to the old function name so older code doesn’t break.

“Docs deprecated” features tend to work “forever”, meaning that they won’t be removed, and they’ll never generate a deprecation warning. However, such changes are noted in the *Pyramid Change History*, so it’s possible to know that you should change older spellings to newer ones to ensure that people reading your code can find the APIs you’re using in the Pyramid docs.

Python support policy

At the time of a Pyramid version release, each supports all versions of Python through the end of their lifespans. The end-of-life for a given version of Python is when security updates are no longer released.

- Python 3.2 Lifespan ends February 2016.
- Python 3.3 Lifespan ends September 2017.
- Python 3.4 Lifespan TBD.
- Python 3.5 Lifespan TBD.
- Python 3.6 Lifespan December 2021.

To determine the Python support for a specific release of Pyramid, view its `tox.ini` file at the root of the repository’s version.

Consulting the change history

Your first line of defense against application failures caused by upgrading to a newer Pyramid release is always to read the *Pyramid Change History* to find the deprecations and removals for each release between the release you’re currently running and the one to which you wish to upgrade. The change history notes every deprecation within a *Deprecation* section and every removal within a *Backwards Incompatibilities* section for each release.

The change history often contains instructions for changing your code to avoid deprecation warnings and how to change docs-deprecated spellings to newer ones. You can follow along with each deprecation explanation in the change history, simply doing a `grep` or other code search to your application, using the change log examples to remediate each potential problem.

Testing your application under a new Pyramid release

Once you've upgraded your application to a new Pyramid release and you've remediated as much as possible by using the change history notes, you'll want to run your application's tests (see *Run the tests*) in such a way that you can see DeprecationWarnings printed to the console when the tests run.

```
$ python -Wd setup.py test -q
```

The `-Wd` argument tells Python to print deprecation warnings to the console. See the Python `-W` flag documentation for more information.

As your tests run, deprecation warnings will be printed to the console explaining the deprecation and providing instructions about how to prevent the deprecation warning from being issued. For example:

```
$ python -Wd setup.py test -q
# .. elided ...
running build_ext
/home/chris/projects/pyramid/env27/myproj/myproj/views.py:3:
DeprecationWarning: static: The "pyramid.view.static" class is deprecated
as of Pyramid 1.1; use the "pyramid.static.static_view" class instead with
the "use_subpath" argument set to True.
    from pyramid.view import static
.
-----
Ran 1 test in 0.014s

OK
```

In the above case, it's line #3 in the `myproj.views` module (`from pyramid.view import static`) that is causing the problem:

```
1 from pyramid.view import view_config
2
3 from pyramid.view import static
4 myview = static('static', 'static')
```

The deprecation warning tells me how to fix it, so I can change the code to do things the newer way:

```
1 from pyramid.view import view_config
2
3 from pyramid.static import static_view
4 myview = static_view('static', 'static', use_subpath=True)
```

When I run the tests again, the deprecation warning is no longer printed to my console:

```
$ python -Wd setup.py test -q
# .. elided ...
running build_ext
.
-----
Ran 1 test in 0.014s

OK
```

My application doesn't have any tests or has few tests

If your application has no tests, or has only moderate test coverage, running tests won't tell you very much, because the Pyramid codepaths that generate deprecation warnings won't be executed.

In this circumstance, you can start your application interactively under a server run with the `PYTHONWARNINGS` environment variable set to `default`. On UNIX, you can do that via:

```
$ PYTHONWARNINGS=default $VENV/bin/pserve development.ini
```

On Windows, you need to issue two commands:

```
c:\> set PYTHONWARNINGS=default
c:\> Scripts/pserve.exe development.ini
```

At this point, it's ensured that deprecation warnings will be printed to the console whenever a codepath is hit that generates one. You can then click around in your application interactively to try to generate them, and remediate as explained in *Testing your application under a new Pyramid release*.

See the `PYTHONWARNINGS` environment variable documentation or the Python `-W` flag documentation for more information.

Upgrading to the very latest Pyramid release

When you upgrade your application to the most recent Pyramid release, it's advisable to upgrade step-wise through each most recent minor release, beginning with the one that you know your application currently runs under, and ending on the most recent release. For example, if your application is running in production on Pyramid 1.2.1, and the most recent Pyramid 1.3 release is Pyramid 1.3.3, and the most recent Pyramid release is 1.4.4, it's advisable to do this:

- Upgrade your environment to the most recent 1.2 release. For example, the most recent 1.2 release might be 1.2.3, so upgrade to it. Then run your application’s tests under 1.2.3 as described in *Testing your application under a new Pyramid release*. Note any deprecation warnings and remediate.
- Upgrade to the most recent 1.3 release, 1.3.3. Run your application’s tests, note any deprecation warnings, and remediate.
- Upgrade to 1.4.4. Run your application’s tests, note any deprecation warnings, and remediate.

If you skip testing your application under each minor release (for example if you upgrade directly from 1.2.1 to 1.4.4), you might miss a deprecation warning and waste more time trying to figure out an error caused by a feature removal than it would take to upgrade stepwise through each minor release.

Thread Locals

A *thread local* variable is a variable that appears to be a “global” variable to an application which uses it. However, unlike a true global variable, one thread or process serving the application may receive a different value than another thread or process when that variable is “thread local”.

When a request is processed, Pyramid makes two *thread local* variables available to the application: a “registry” and a “request”.

Why and How Pyramid Uses Thread Local Variables

How are thread locals beneficial to Pyramid and application developers who use Pyramid? Well, usually they’re decidedly **not**. Using a global or a thread local variable in any application usually makes it a lot harder to understand for a casual reader. Use of a thread local or a global is usually just a way to avoid passing some value around between functions, which is itself usually a very bad idea, at least if code readability counts as an important concern.

For historical reasons, however, thread local variables are indeed consulted by various Pyramid API functions. For example, the implementation of the `pyramid.security` function named `authenticated_userid()` (deprecated as of 1.5) retrieves the thread local *application registry* as a matter of course to find an *authentication policy*. It uses the `pyramid.threadlocal.get_current_registry()` function to retrieve the application registry, from which it looks up the authentication policy; it then uses the authentication policy to retrieve the authenticated user id. This is how Pyramid allows arbitrary authentication policies to be “plugged in”.

When they need to do so, Pyramid internals use two API functions to retrieve the *request* and *application registry*: `get_current_request()` and `get_current_registry()`. The former returns the

“current” request; the latter returns the “current” registry. Both `get_current_*` functions retrieve an object from a thread-local data structure. These API functions are documented in `pyramid.threadlocal`.

These values are thread locals rather than true globals because one Python process may be handling multiple simultaneous requests or even multiple Pyramid applications. If they were true globals, Pyramid could not handle multiple simultaneous requests or allow more than one Pyramid application instance to exist in a single Python process.

Because one Pyramid application is permitted to call *another* Pyramid application from its own *view* code (perhaps as a *WSGI* app with help from the `pyramid.wsgi.wsgiapp2()` decorator), these variables are managed in a *stack* during normal system operations. The stack instance itself is a `threading.local`.

During normal operations, the thread locals stack is managed by a *Router* object. At the beginning of a request, the Router pushes the application’s registry and the request on to the stack. At the end of a request, the stack is popped. The topmost request and registry on the stack are considered “current”. Therefore, when the system is operating normally, the very definition of “current” is defined entirely by the behavior of a pyramid *Router*.

However, during unit testing, no Router code is ever invoked, and the definition of “current” is defined by the boundary between calls to the `pyramid.config.Configurator.begin()` and `pyramid.config.Configurator.end()` methods (or between calls to the `pyramid.testing.setUp()` and `pyramid.testing.tearDown()` functions). These functions push and pop the threadlocal stack when the system is under test. See *Test Set Up and Tear Down* for the definitions of these functions.

Scripts which use Pyramid machinery but never actually start a WSGI server or receive requests via HTTP, such as scripts which use the `pyramid.scripting` API, will never cause any Router code to be executed. However, the `pyramid.scripting` APIs also push some values on to the thread locals stack as a matter of course. Such scripts should expect the `get_current_request()` function to always return `None`, and should expect the `get_current_registry()` function to return exactly the same *application registry* for every request.

Why You Shouldn’t Abuse Thread Locals

You probably should almost never use the `get_current_request()` or `get_current_registry()` functions, except perhaps in tests. In particular, it’s almost always a mistake to use `get_current_request` or `get_current_registry` in application code because its usage makes it possible to write code that can be neither easily tested nor scripted. Inappropriate usage is defined as follows:

- `get_current_request` should never be called within the body of a *view callable*, or within code called by a view callable. View callables already have access to the request (it’s passed in to each as `request`).

- `get_current_request` should never be called in *resource* code. If a resource needs access to the request, it should be passed the request by a *view callable*.
- `get_current_request` function should never be called because it's "easier" or "more elegant" to think about calling it than to pass a request through a series of function calls when creating some API design. Your application should instead, almost certainly, pass around data derived from the request rather than relying on being able to call this function to obtain the request in places that actually have no business knowing about it. Parameters are *meant* to be passed around as function arguments; this is why they exist. Don't try to "save typing" or create "nicer APIs" by using this function in the place where a request is required; this will only lead to sadness later.
- Neither `get_current_request` nor `get_current_registry` should ever be called within application-specific forks of third-party library code. The library you've forked almost certainly has nothing to do with Pyramid, and making it dependent on Pyramid (rather than making your pyramid application depend upon it) means you're forming a dependency in the wrong direction.

Use of the `get_current_request()` function in application code *is* still useful in very limited circumstances. As a rule of thumb, usage of `get_current_request` is useful **within code which is meant to eventually be removed**. For instance, you may find yourself wanting to deprecate some API that expects to be passed a request object in favor of one that does not expect to be passed a request object. But you need to keep implementations of the old API working for some period of time while you deprecate the older API. So you write a "facade" implementation of the new API which calls into the code which implements the older API. Since the new API does not require the request, your facade implementation doesn't have local access to the request when it needs to pass it into the older API implementation. After some period of time, the older implementation code is disused and the hack that uses `get_current_request` is removed. This would be an appropriate place to use the `get_current_request`.

Use of the `get_current_registry()` function should be limited to testing scenarios. The registry made current by use of the `pyramid.config.Configurator.begin()` method during a test (or via `pyramid.testing.setUp()`) when you do not pass one in is available to you via this API.

Using the Zope Component Architecture in Pyramid

Under the hood, Pyramid uses a *Zope Component Architecture* component registry as its *application registry*. The Zope Component Architecture is referred to colloquially as the "ZCA."

The `zope.component` API used to access data in a traditional Zope application can be opaque. For example, here is a typical "unnamed utility" lookup using the `zope.component.getUtility()` global API as it might appear in a traditional Zope application:

```
1 from pyramid.interfaces import ISettings
2 from zope.component import getUtility
3 settings = getUtility(ISettings)
```

After this code runs, `settings` will be a Python dictionary. But it's unlikely that any “civilian” will be able to figure this out just by reading the code casually. When the `zope.component.getUtility` API is used by a developer, the conceptual load on a casual reader of code is high.

While the ZCA is an excellent tool with which to build a *framework* such as Pyramid, it is not always the best tool with which to build an *application* due to the opacity of the `zope.component` APIs. Accordingly, Pyramid tends to hide the presence of the ZCA from application developers. You needn't understand the ZCA to create a Pyramid application; its use is effectively only a framework implementation detail.

However, developers who are already used to writing *Zope* applications often still wish to use the ZCA while building a Pyramid application. Pyramid makes this possible.

Using the ZCA global API in a Pyramid application

Zope uses a single ZCA registry—the “global” ZCA registry—for all Zope applications that run in the same Python process, effectively making it impossible to run more than one Zope application in a single process.

However, for ease of deployment, it's often useful to be able to run more than a single application per process. For example, use of a *PasteDeploy* “composite” allows you to run separate individual WSGI applications in the same process, each answering requests for some URL prefix. This makes it possible to run, for example, a TurboGears application at `/turbogears` and a Pyramid application at `/pyramid`, both served up using the same WSGI server within a single Python process.

Most production Zope applications are relatively large, making it impractical due to memory constraints to run more than one Zope application per Python process. However, a Pyramid application may be very small and consume very little memory, so it's a reasonable goal to be able to run more than one Pyramid application per process.

In order to make it possible to run more than one Pyramid application in a single process, Pyramid defaults to using a separate ZCA registry *per application*.

While this services a reasonable goal, it causes some issues when trying to use patterns which you might use to build a typical *Zope* application to build a Pyramid application. Without special help, ZCA “global” APIs such as `zope.component.getUtility()` and `zope.component.getSiteManager()` will use the ZCA “global” registry. Therefore, these APIs will appear to fail when used in a Pyramid application, because they'll be consulting the ZCA global registry rather than the component registry associated with your Pyramid application.

There are three ways to fix this: by disusing the ZCA global API entirely, by using `pyramid.config.Configurator.hook_zca()` or by passing the ZCA global registry to the *Configurator* constructor at startup time. We'll describe all three methods in this section.

Disusing the global ZCA API

ZCA “global” API functions such as `zope.component.getSiteManager`, `zope.component.getUtility`, `zope.component.getAdapter()`, and `zope.component.getMultiAdapter()` aren’t strictly necessary. Every component registry has a method API that offers the same functionality; it can be used instead. For example, presuming the `registry` value below is a Zope Component Architecture component registry, the following bit of code is equivalent to `zope.component.getUtility(IFoo)`:

```
registry.getUtility(IFoo)
```

The full method API is documented in the `zope.component` package, but it largely mirrors the “global” API almost exactly.

If you are willing to disuse the “global” ZCA APIs and use the method interface of a registry instead, you need only know how to obtain the Pyramid component registry.

There are two ways of doing so:

- use the `pyramid.threadlocal.get_current_registry()` function within Pyramid view or resource code. This will always return the “current” Pyramid application registry.
- use the attribute of the `request` object named `registry` in your Pyramid view code, e.g., `request.registry`. This is the ZCA component registry related to the running Pyramid application.

See *Thread Locals* for more information about `pyramid.threadlocal.get_current_registry()`.

Enabling the ZCA global API by using `hook_zca`

Consider the following bit of idiomatic Pyramid startup code:

```
1 from pyramid.config import Configurator
2
3 def app(global_settings, **settings):
4     config = Configurator(settings=settings)
5     config.include('some.other.package')
6     return config.make_wsgi_app()
```

When the `app` function above is run, a *Configurator* is constructed. When the configurator is created, it creates a *new application registry* (a ZCA component registry). A new registry is constructed whenever the `registry` argument is omitted, when a *Configurator* constructor is called, or when a `registry` argument with a value of `None` is passed to a *Configurator* constructor.

During a request, the application registry created by the *Configurator* is “made current”. This means calls to `get_current_registry()` in the thread handling the request will return the component registry associated with the application.

As a result, application developers can use `get_current_registry` to get the registry and thus get access to utilities and such, as per *Disusing the global ZCA API*. But they still cannot use the global ZCA API. Without special treatment, the ZCA global APIs will always return the global ZCA registry (the one in `zope.component.globalregistry.base`).

To “fix” this and make the ZCA global APIs use the “current” Pyramid registry, you need to call `hook_zca()` within your setup code. For example:

```
1 from pyramid.config import Configurator
2
3 def app(global_settings, **settings):
4     config = Configurator(settings=settings)
5     config.hook_zca()
6     config.include('some.other.application')
7     return config.make_wsgi_app()
```

We’ve added a line to our original startup code, line number 5, which calls `config.hook_zca()`. The effect of this line under the hood is that an analogue of the following code is executed:

```
1 from zope.component import getSiteManager
2 from pyramid.threadlocal import get_current_registry
3 getSiteManager().sethook(get_current_registry)
```

This causes the ZCA global API to start using the Pyramid application registry in threads which are running a Pyramid request.

Calling `hook_zca` is usually sufficient to “fix” the problem of being able to use the global ZCA API within a Pyramid application. However, it also means that a Zope application that is running in the same process may start using the Pyramid global registry instead of the Zope global registry, effectively inverting the original problem. In such a case, follow the steps in the next section, *Enabling the ZCA global API by using the ZCA global registry*.

Enabling the ZCA global API by using the ZCA global registry

You can tell your Pyramid application to use the ZCA global registry at startup time instead of constructing a new one:

```
1 from zope.component import getGlobalSiteManager
2 from pyramid.config import Configurator
3
4 def app(global_settings, **settings):
5     globalreg = getGlobalSiteManager()
6     config = Configurator(registry=globalreg)
7     config.setup_registry(settings=settings)
8     config.include('some.other.application')
9     return config.make_wsgi_app()
```

Lines 5, 6, and 7 above are the interesting ones. Line 5 retrieves the global ZCA component registry. Line 6 creates a *Configurator*, passing the global ZCA registry into its constructor as the `registry` argument. Line 7 “sets up” the global registry with Pyramid-specific registrations; this is code that is normally executed when a registry is constructed rather than created, but we must call it “by hand” when we pass an explicit registry.

At this point, Pyramid will use the ZCA global registry rather than creating a new application-specific registry. Since by default the ZCA global API will use this registry, things will work as you might expect in a Zope app when you use the global ZCA API.

API Documentation

API Documentation

Comprehensive reference material for every public API exposed by Pyramid:

`pyramid.authentication`

Authentication Policies

```
class AuthTktAuthenticationPolicy (secret, callback=None, cookie_name='auth_tkt',  
                                     secure=False, include_ip=False, time-  
                                     out=None, reissue_time=None, max_age=None,  
                                     path='/', http_only=False, wild_domain=True,  
                                     debug=False, hashalg='sha512', par-  
                                     ent_domain=False, domain=None)
```

A Pyramid *authentication policy* which obtains data from a Pyramid “auth ticket” cookie.

Constructor Arguments

`secret`

The secret (a string) used for `auth_tkt` cookie signing. This value should be unique across all values provided to Pyramid for various subsystem secrets (see *Admonishment Against Secret-Sharing*). Required.

`callback`

Default: `None`. A callback passed the `userid` and the request, expected to return `None` if the `userid` doesn't exist or a sequence of principal identifiers (possibly empty) if the user does exist. If `callback` is `None`, the `userid` will be assumed to exist with no principals. Optional.

`cookie_name`

Default: `auth_tkt`. The cookie name used (string). Optional.

`secure`

Default: `False`. Only send the cookie back over a secure conn. Optional.

`include_ip`

Default: `False`. Make the requesting IP address part of the authentication data in the cookie. Optional.

For IPv6 this option is not recommended. The `mod_auth_tkt` specification does not specify how to handle IPv6 addresses, so using this option in combination with IPv6 addresses may cause an incompatible cookie. It ties the authentication ticket to that individual's IPv6 address.

`timeout`

Default: `None`. Maximum number of seconds which a newly issued ticket will be considered valid. After this amount of time, the ticket will expire (effectively logging the user out). If this value is `None`, the ticket never expires. Optional.

`reissue_time`

Default: `None`. If this parameter is set, it represents the number of seconds that must pass before an authentication token cookie is automatically reissued as the result of a request which requires authentication. The duration is measured as the number of seconds since the last `auth_tkt` cookie was issued and ‘now’. If this value is 0, a new ticket cookie will be reissued on every request which requires authentication.

A good rule of thumb: if you want auto-expired cookies based on inactivity: set the `timeout` value to 1200 (20 mins) and set the `reissue_time` value to perhaps a tenth of the `timeout` value (120 or 2 mins). It’s nonsensical to set the `timeout` value lower than the `reissue_time` value, as the ticket will never be reissued if so. However, such a configuration is not explicitly prevented.

Optional.

`max_age`

Default: `None`. The max age of the `auth_tkt` cookie, in seconds. This differs from `timeout` inasmuch as `timeout` represents the lifetime of the ticket contained in the cookie, while this value represents the lifetime of the cookie itself. When this value is set, the cookie’s `Max-Age` and `Expires` settings will be set, allowing the `auth_tkt` cookie to last between browser sessions. It is typically nonsensical to set this to a value that is lower than `timeout` or `reissue_time`, although it is not explicitly prevented. Optional.

`path`

Default: `/`. The path for which the `auth_tkt` cookie is valid. May be desirable if the application only serves part of a domain. Optional.

`http_only`

Default: `False`. Hide cookie from JavaScript by setting the `HttpOnly` flag. Not honored by all browsers. Optional.

`wild_domain`

Default: `True`. An `auth_tkt` cookie will be generated for the wildcard domain. If your site is hosted as `example.com` this will make the cookie available for sites underneath `example.com` such as `www.example.com`. Optional.

`parent_domain`

Default: `False`. An `auth_tkt` cookie will be generated for the parent domain of the current site. For example if your site is hosted under `www.example.com` a cookie will be generated for `.example.com`. This can be useful if you have multiple sites sharing the same domain. This option supercedes the `wild_domain` option. Optional.

This option is available as of Pyramid 1.5.

`domain`

Default: `None`. If provided the `auth_tkt` cookie will only be set for this domain. This option is not compatible with `wild_domain` and `parent_domain`. Optional.

This option is available as of Pyramid 1.5.

`hashalg`

Default: `sha512` (the literal string).

Any hash algorithm supported by Python's `hashlib.new()` function can be used as the `hashalg`.

Cookies generated by different instances of `AuthTktAuthenticationPolicy` using different `hashalg` options are not compatible. Switching the `hashalg` will imply that all existing users with a valid cookie will be required to re-login.

This option is available as of Pyramid 1.4.

Optional.

`debug`

Default: `False`. If `debug` is `True`, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

Objects of this class implement the interface described by `pyramid.interfaces.IAuthenticationPolicy`.

authenticated_userid (*request*)

Return the authenticated userid or `None`.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a `callback` is registered, this will return the userid if and only if the callback returns a value that is not `None`.

effective_principals (*request*)

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated userid, or the `callback` returns `None`, this will be the only principal:

```
return [Everyone]
```

If the `callback` does not return `None` and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the `callback`:

```
extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals
```

forget (*request*)

A list of headers which will delete appropriate cookies.

remember (*request, userid, **kw*)

Accepts the following kw args: `max_age=<int-seconds>`,
`tokens=<sequence-of-ascii-strings>`.

Return a list of headers which will set appropriate cookies on the response.

unauthenticated_userid (*request*)

The `userid` key within the `auth_tkt` cookie.

class RemoteUserAuthenticationPolicy (*environ_key='REMOTE_USER', call-*
back=None, debug=False)

A Pyramid *authentication policy* which obtains data from the `REMOTE_USER` WSGI environment variable.

Constructor Arguments

`environ_key`

Default: `REMOTE_USER`. The key in the WSGI environ which provides the userid.

`callback`

Default: `None`. A callback passed the userid and the request, expected to return `None` if the userid doesn't exist or a sequence of principal identifiers (possibly empty) representing groups if the user does exist. If `callback` is `None`, the userid will be assumed to exist with no group principals.

`debug`

Default: `False`. If `debug` is `True`, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

Objects of this class implement the interface described by `pyramid.interfaces.IAuthenticationPolicy`.

`authenticated_userid(request)`

Return the authenticated userid or `None`.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a `callback` is registered, this will return the userid if and only if the callback returns a value that is not `None`.

`effective_principals(request)`

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated userid, or the `callback` returns `None`, this will be the only principal:

```
return [Everyone]
```

If the `callback` does not return `None` and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the `callback`:

```
extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals
```


forget (*request*)

A no-op. The REMOTE_USER does not provide a protocol for forgetting the user. This will be application-specific and can be done somewhere else or in a subclass.

remember (*request, userid, **kw*)

A no-op. The REMOTE_USER does not provide a protocol for remembering the user. This will be application-specific and can be done somewhere else or in a subclass.

unauthenticated_userid (*request*)

The REMOTE_USER value found within the `environ`.

class SessionAuthenticationPolicy (*prefix='auth.', callback=None, debug=False*)

A Pyramid authentication policy which gets its data from the configured *session*. For this authentication policy to work, you will have to follow the instructions in the *Sessions* to configure a *session factory*.

Constructor Arguments

`prefix`

A prefix used when storing the authentication parameters in the session. Defaults to 'auth.'. Optional.

`callback`

Default: `None`. A callback passed the userid and the request, expected to return `None` if the userid doesn't exist or a sequence of principal identifiers (possibly empty) if the user does exist. If `callback` is `None`, the userid will be assumed to exist with no principals. Optional.

`debug`

Default: `False`. If `debug` is `True`, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

authenticated_userid (*request*)

Return the authenticated userid or `None`.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a `callback` is registered, this will return the userid if and only if the callback returns a value that is not `None`.

effective_principals (*request*)

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated userid, or the `callback` returns `None`, this will be the only principal:

```
return [Everyone]
```

If the callback does not return `None` and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the callback:

```
extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals
```

forget (*request*)

Remove the stored userid from the session.

remember (*request*, *userid*, ***kw*)

Store a userid in the session.

class BasicAuthAuthenticationPolicy (*check*, *realm='Realm'*, *debug=False*)

A Pyramid authentication policy which uses HTTP standard basic authentication protocol to authenticate users. To use this policy you will need to provide a callback which checks the supplied user credentials against your source of login data.

Constructor Arguments

check

A callback function passed a username, password and request, in that order as positional arguments. Expected to return `None` if the userid doesn't exist or a sequence of principal identifiers (possibly empty) if the user does exist.

realm

Default: "Realm". The Basic Auth Realm string. Usually displayed to the user by the browser in the login dialog.

debug

Default: `False`. If `debug` is `True`, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

Issuing a challenge

Regular browsers will not send username/password credentials unless they first receive a challenge from the server. The following recipe will register a view that will send a Basic Auth challenge to the user whenever there is an attempt to call a view which results in a Forbidden response:

```
from pyramid.httpexceptions import HTTPUnauthorized
from pyramid.security import forget
from pyramid.view import forbidden_view_config

@forbidden_view_config()
def basic_challenge(request):
    response = HTTPUnauthorized()
    response.headers.update(forget(request))
    return response
```

authenticated_userid (*request*)

Return the authenticated userid or None.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a callback is registered, this will return the userid if and only if the callback returns a value that is not None.

effective_principals (*request*)

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated userid, or the callback returns None, this will be the only principal:

```
return [Everyone]
```

If the callback does not return None and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the callback:

```
extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals
```

forget (*request*)

Returns challenge headers. This should be attached to a response to indicate that credentials are required.

remember (*request*, *userid*, ***kw*)

A no-op. Basic authentication does not provide a protocol for remembering the user. Credentials are sent on every request.

unauthenticated_userid(*request*)

The userid parsed from the `Authorization` request header.

class RepozeWho1AuthenticationPolicy(*identifier_name='auth_tkt', callback=None*)

A Pyramid *authentication policy* which obtains data from the `repoze.who` 1.X WSGI 'API' (the `repoze.who.identity` key in the WSGI environment).

Constructor Arguments

`identifier_name`

Default: `auth_tkt`. The `repoze.who` plugin name that performs remember/forget. Optional.

`callback`

Default: `None`. A callback passed the `repoze.who` identity and the *request*, expected to return `None` if the user represented by the identity doesn't exist or a sequence of principal identifiers (possibly empty) representing groups if the user does exist. If `callback` is `None`, the `userid` will be assumed to exist with no group principals.

Objects of this class implement the interface described by `pyramid.interfaces.IAuthenticationPolicy`.

authenticated_userid(*request*)

Return the authenticated userid or `None`.

If no `callback` is registered, this will be the same as `unauthenticated_userid`.

If a `callback` is registered, this will return the `userid` if and only if the `callback` returns a value that is not `None`.

effective_principals(*request*)

A list of effective principals derived from the identity.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no identity, or the `callback` returns `None`, this will be the only principal.

If the `callback` does not return `None` and an identity is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the `callback`.

forget (*request*)

Forget the current authenticated user.

Return headers that, if included in a response, will delete the cookie responsible for tracking the current user.

remember (*request*, *userid*, ***kw*)

Store the *userid* as *repoze.who.userid*.

The identity to authenticated to *repoze.who* will contain the given *userid* as *userid*, and provide all keyword arguments as additional identity keys. Useful keys could be *max_age* or *userdata*.

unauthenticated_userid (*request*)

Return the *repoze.who.userid* key from the detected identity.

Helper Classes

```
class AuthTktCookieHelper (secret, cookie_name='auth_tkt', secure=False, include_ip=False, time-out=None, reissue_time=None, max_age=None, http_only=False, path='/', wild_domain=True, hashalg='md5', parent_domain=False, domain=None)
```

A helper class for use in third-party authentication policy implementations. See *pyramid.authentication.AuthTktAuthenticationPolicy* for the meanings of the constructor arguments.

```
class AuthTicket (secret, userid, ip, tokens=(), user_data='', time=None, cookie_name='auth_tkt', secure=False, hashalg='md5')
```

This class represents an authentication token. You must pass in the shared secret, the *userid*, and the IP address. Optionally you can include *tokens* (a list of strings, representing role names), *'user_data'*, which is arbitrary data available for your own use in later scripts. Lastly, you can override the cookie name and timestamp.

Once you provide all the arguments, use *.cookie_value()* to generate the appropriate authentication ticket.

Usage:

```
token = AuthTicket('sharedsecret', 'username',
    os.environ['REMOTE_ADDR'], tokens=['admin'])
val = token.cookie_value()
```

exception `AuthTktCookieHelper.BadTicket` (*msg, expected=None*)

Exception raised when a ticket can't be parsed. If we get far enough to determine what the expected digest should have been, `expected` is set. This should not be shown by default, but can be useful for debugging.

`AuthTktCookieHelper.forget` (*request*)

Return a set of expires Set-Cookie headers, which will destroy any existing `auth_tkt` cookie when attached to a response

`AuthTktCookieHelper.identify` (*request*)

Return a dictionary with authentication information, or `None` if no valid `auth_tkt` is attached to request

static `AuthTktCookieHelper.parse_ticket` (*secret, ticket, ip,*
hashalg='md5')

Parse the ticket, returning (timestamp, userid, tokens, user_data).

If the ticket cannot be parsed, a `BadTicket` exception will be raised with an explanation.

`AuthTktCookieHelper.remember` (*request, userid, max_age=None, to-*
kens=())

Return a set of Set-Cookie headers; when set into a response, these headers will represent a valid authentication ticket.

max_age The max age of the `auth_tkt` cookie, in seconds. When this value is set, the cookie's `Max-Age` and `Expires` settings will be set, allowing the `auth_tkt` cookie to last between browser sessions. If this value is `None`, the `max_age` value provided to the helper itself will be used as the `max_age` value. Default: `None`.

tokens A sequence of strings that will be placed into the `auth_tkt` tokens field. Each string in the sequence must be of the Python `str` type and must match the regex `^[A-Za-z][A-Za-z0-9+_-]*$`. Tokens are available in the returned identity when an `auth_tkt` is found in the request and unpacked. Default: `()`.

pyramid.authorization

class ACLAuthorizationPolicy

An *authorization policy* which consults an *ACL* object attached to a *context* to determine authorization information about a *principal* or multiple principals. If the context is part of a *lineage*, the context's parents are consulted for ACL information too. The following is true about this security policy.

- When checking whether the 'current' user is permitted (via the `permits` method), the security policy consults the `context` for an ACL first. If no ACL exists on the context, or one does exist but the ACL does not explicitly allow or deny access for any of the effective principals, consult the context's parent ACL, and so on, until the lineage is exhausted or we determine that the policy permits or denies.

During this processing, if any `pyramid.security.Deny` ACE is found matching any principal in `principals`, stop processing by returning an `pyramid.security.ACLDenied` instance (equals `False`) immediately. If any `pyramid.security.Allow` ACE is found matching any principal, stop processing by returning an `pyramid.security.ACLAllowed` instance (equals `True`) immediately. If we exhaust the context's *lineage*, and no ACE has explicitly permitted or denied access, return an instance of `pyramid.security.ACLDenied` (equals `False`).

- When computing principals allowed by a permission via the `pyramid.security.principals_allowed_by_permission()` method, we compute the set of principals that are explicitly granted the permission in the provided context. We do this by walking 'up' the object graph *from the root* to the context. During this walking process, if we find an explicit `pyramid.security.Allow` ACE for a principal that matches the permission, the principal is included in the allow list. However, if later in the walking process that principal is mentioned in any `pyramid.security.Deny` ACE for the permission, the principal is removed from the allow list. If a `pyramid.security.Deny` to the principal `pyramid.security.Everyone` is encountered during the walking process that matches the permission, the allow list is cleared for all principals encountered in previous ACLs. The walking process ends after we've processed the any ACL directly attached to context; a set of principals is returned.

Objects of this class implement the `pyramid.interfaces.IAuthorizationPolicy` interface.

pyramid.compat

The `pyramid.compat` module provides platform and version compatibility for Pyramid and its add-ons across Python platform and version differences. APIs will be removed from this module over time as Pyramid ceases to support systems which require compatibility imports.

ascii_native_(s)

Python 3: If `s` is an instance of `text_type`, return `s.encode('ascii')`, otherwise return `str(s, 'ascii', 'strict')`

Python 2: If `s` is an instance of `text_type`, return `s.encode('ascii')`, otherwise return `str(s)`

binary_type

Binary type for this platform. For Python 3, it's `bytes`. For Python 2, it's `str`.

bytes_(s, encoding='latin-1', errors='strict')

If `s` is an instance of `text_type`, return `s.encode(encoding, errors)`, otherwise return `s`

class_types

Sequence of class types for this platform. For Python 3, it's `(type,)`. For Python 2, it's `(type, types.ClassType)`.

configparser

On Python 2, the `ConfigParser` module, on Python 3, the `configparser` module.

escape(v)

On Python 2, the `cgi.escape` function, on Python 3, the `html.escape` function.

exec_(code, globs=None, locs=None)

Exec code in a compatible way on both Python 2 and 3.

im_func

On Python 2, the string value `im_func`, on Python 3, the string value `__func__`.

input_(v)

On Python 2, the `raw_input` function, on Python 3, the `input` function.

integer_types

Sequence of integer types for this platform. For Python 3, it's `(int,)`. For Python 2, it's `(int, long)`.

is_nonstr_iter(v)

Return `True` if `v` is a non-`str` iterable on both Python 2 and Python 3.

iteritems_(d)

Return `d.items()` on Python 3, `d.iteritems()` on Python 2.

intervalvalues_(*d*)

Return `d.values()` on Python 3, `d.intervalvalues()` on Python 2.

iterkeys_(*d*)

Return `d.keys()` on Python 3, `d.iterkeys()` on Python 2.

long

Long type for this platform. For Python 3, it's `int`. For Python 2, it's `long`.

map_(*v*)

Return `list(map(v))` on Python 3, `map(v)` on Python 2.

pickle

`cPickle` module if it exists, `pickle` module otherwise.

PY3

True if running on Python 3, False otherwise.

PYPY

True if running on PyPy, False otherwise.

reraise(*tp, value, tb=None*)

Reraise an exception in a compatible way on both Python 2 and Python 3, e.g. `reraise(*sys.exc_info())`.

string_types

Sequence of string types for this platform. For Python 3, it's `(str,)`. For Python 2, it's `(basestring,)`.

SimpleCookie

On Python 2, the `Cookie.SimpleCookie` class, on Python 3, the `http.cookies.SimpleCookie` module.

text_(*s, encoding='latin-1', errors='strict'*)

If *s* is an instance of `binary_type`, return `s.decode(encoding, errors)`, otherwise return *s*

text_type

Text type for this platform. For Python 3, it's `str`. For Python 2, it's `unicode`.

native_(*s, encoding='latin-1', errors='strict'*)

Python 3: If *s* is an instance of `text_type`, return *s*, otherwise return `str(s, encoding, errors)`

Python 2: If *s* is an instance of `text_type`, return `s.encode(encoding, errors)`, otherwise return `str(s)`

urlparse

urlparse module on Python 2, urllib.parse module on Python 3.

url_quote

urllib.quote function on Python 2, urllib.parse.quote function on Python 3.

url_quote_plus

urllib.quote_plus function on Python 2, urllib.parse.quote_plus function on Python 3.

url_unquote

urllib.unquote function on Python 2, urllib.parse.unquote function on Python 3.

url_encode

urllib.urlencode function on Python 2, urllib.parse.urlencode function on Python 3.

url_open

urllib2.urlopen function on Python 2, urllib.request.urlopen function on Python 3.

url_unquote_text (*v, encoding='utf-8', errors='replace'*)

On Python 2, return `url_unquote(v).decode(encoding, errors)`; on Python 3, return the result of `urllib.parse.unquote`.

url_unquote_native (*v, encoding='utf-8', errors='replace'*)

On Python 2, return `native_(url_unquote_text_v, encoding, errors)`; on Python 3, return the result of `urllib.parse.unquote`.

pyramid.config

```
class Configurator (registry=None, package=None, settings=None, root_factory=None,  
                    authentication_policy=None, authorization_policy=None, ren-  
                    derers=None, debug_logger=None, locale_negotiator=None,  
                    request_factory=None, response_factory=None, de-  
                    fault_permission=None, session_factory=None, de-  
                    fault_view_mapper=None, autocommit=False, exceptionre-  
                    sponse_view=<function default_exceptionresponse_view>,  
                    route_prefix=None, introspection=True, root_package=None)
```

A Configurator is used to configure a Pyramid *application registry*.

If the *registry* argument is not *None*, it must be an instance of the *pyramid.registry.Registry* class representing the registry to configure. If *registry* is *None*, the configurator

will create a `pyramid.registry.Registry` instance itself; it will also perform some default configuration that would not otherwise be done. After its construction, the configurator may be used to add further configuration to the registry.



If `registry` is assigned the above-mentioned class instance, all other constructor arguments are ignored, with the exception of `package`.

If the `package` argument is passed, it must be a reference to a Python *package* (e.g. `sys.modules['thepackage']`) or a *dotted Python name* to the same. This value is used as a basis to convert relative paths passed to various configuration methods, such as methods which accept a `renderer` argument, into absolute paths. If `None` is passed (the default), the package is assumed to be the Python package in which the *caller* of the `Configurator` constructor lives.

If the `root_package` is passed, it will propagate through the configuration hierarchy as a way for included packages to locate resources relative to the package in which the main `Configurator` was created. If `None` is passed (the default), the `root_package` will be derived from the `package` argument. The `package` attribute is always pointing at the package being included when using `include()`, whereas the `root_package` does not change.

If the `settings` argument is passed, it should be a Python dictionary representing the *deployment settings* for this application. These are later retrievable using the `pyramid.registry.Registry.settings` attribute (aka `request.registry.settings`).

If the `root_factory` argument is passed, it should be an object representing the default *root factory* for your application or a *dotted Python name* to the same. If it is `None`, a default root factory will be used.

If `authentication_policy` is passed, it should be an instance of an *authentication policy* or a *dotted Python name* to the same.

If `authorization_policy` is passed, it should be an instance of an *authorization policy* or a *dotted Python name* to the same.



A `ConfigurationError` will be raised when an authorization policy is supplied without also supplying an authentication policy (authorization requires authentication).

If `renderers` is `None` (the default), a default set of *renderer* factories is used. Else, it should be a list of tuples representing a set of *renderer* factories which should be configured into this application,

and each tuple representing a set of positional values that should be passed to `pyramid.config.Configurator.add_renderer()`.

If `debug_logger` is not passed, a default debug logger that logs to a logger will be used (the logger name will be the package name of the *caller* of this configurator). If it is passed, it should be an instance of the `logging.Logger` (PEP 282) standard library class or a Python logger name. The debug logger is used by Pyramid itself to log warnings and authorization debugging information.

If `locale_negotiator` is passed, it should be a *locale negotiator* implementation or a *dotted Python name* to same. See *Using a Custom Locale Negotiator*.

If `request_factory` is passed, it should be a *request factory* implementation or a *dotted Python name* to the same. See *Changing the Request Factory*. By default it is `None`, which means use the default request factory.

If `response_factory` is passed, it should be a *response factory* implementation or a *dotted Python name* to the same. See *Changing the Response Factory*. By default it is `None`, which means use the default response factory.

If `default_permission` is passed, it should be a *permission* string to be used as the default permission for all view configuration registrations performed against this Configurator. An example of a permission string: `'view'`. Adding a default permission makes it unnecessary to protect each view configuration with an explicit permission, unless your application policy requires some exception for a particular view. By default, `default_permission` is `None`, meaning that view configurations which do not explicitly declare a permission will always be executable by entirely anonymous users (any authorization policy in effect is ignored).

See also:

See also *Setting a Default Permission*.

If `session_factory` is passed, it should be an object which implements the *session factory* interface. If a nondefault value is passed, the `session_factory` will be used to create a session object when `request.session` is accessed. Note that the same outcome can be achieved by calling `pyramid.config.Configurator.set_session_factory()`. By default, this argument is `None`, indicating that no session factory will be configured (and thus accessing `request.session` will throw an error) unless `set_session_factory` is called later during configuration.

If `autocommit` is `True`, every method called on the configurator will cause an immediate action, and no configuration conflict detection will be used. If `autocommit` is `False`, most methods of the configurator will defer their action until `pyramid.config.Configurator.commit()` is called. When `pyramid.config.Configurator.commit()` is called, the actions implied

by the called methods will be checked for configuration conflicts unless `autocommit` is `True`. If a conflict is detected, a `ConfigurationConflictError` will be raised. Calling `pyramid.config.Configurator.make_wsgi_app()` always implies a final commit.

If `default_view_mapper` is passed, it will be used as the default *view mapper* factory for view configurations that don't otherwise specify one (see `pyramid.interfaces.IViewMapperFactory`). If `default_view_mapper` is not passed, a superdefault view mapper will be used.

If `exceptionresponse_view` is passed, it must be a *view callable* or `None`. If it is a view callable, it will be used as an exception view callable when an *exception response* is raised. If `exceptionresponse_view` is `None`, no exception response view will be registered, and all raised exception responses will be bubbled up to Pyramid's caller. By default, the `pyramid.httpexceptions.default_exceptionresponse_view` function is used as the `exceptionresponse_view`.

If `route_prefix` is passed, all routes added with `pyramid.config.Configurator.add_route()` will have the specified path prepended to their pattern.

If `introspection` is passed, it must be a boolean value. If it's `True`, introspection values during actions will be kept for use for tools like the debug toolbar. If it's `False`, introspection values provided by registrations will be ignored. By default, it is `True`.

New in version 1.1: The `exceptionresponse_view` argument.

New in version 1.2: The `route_prefix` argument.

New in version 1.3: The `introspection` argument.

New in version 1.6: The `root_package` argument. The `response_factory` argument.

Controlling Configuration State

`commit()`

Commit any pending configuration actions. If a configuration conflict is detected in the pending configuration actions, this method will raise a `ConfigurationConflictError`; within the traceback of this error will be information about the source of the conflict, usually including file names and line numbers of the cause of the configuration conflicts.

`begin(request=None)`

Indicate that application or test configuration has begun. This pushes a dictionary containing the *application registry* implied by `registry` attribute of this configurator and the *request* implied by the `request` argument onto the *thread local* stack consulted by various `pyramid.threadlocal` API functions.

end()

Indicate that application or test configuration has ended. This pops the last value pushed onto the *thread local* stack (usually by the `begin` method) and returns that value.

include (*callable*, *route_prefix=None*)

Include a configuration callable, to support imperative application extensibility.



In versions of Pyramid prior to 1.2, this function accepted **callables*, but this has been changed to support only a single callable.

A configuration callable should be a callable that accepts a single argument named `config`, which will be an instance of a *Configurator*. However, be warned that it will not be the same configurator instance on which you call this method. The code which runs as a result of calling the callable should invoke methods on the configurator passed to it which add configuration state. The return value of a callable will be ignored.

Values allowed to be presented via the `callable` argument to this method: any callable Python object or any *dotted Python name* which resolves to a callable Python object. It may also be a Python *module*, in which case, the module will be searched for a callable named `includeme`, which will be treated as the configuration callable.

For example, if the `includeme` function below lives in a module named `myapp.myconfig`:

```

1 # myapp.myconfig module
2
3 def my_view(request):
4     from pyramid.response import Response
5     return Response('OK')
6
7 def includeme(config):
8     config.add_view(my_view)
```

You might cause it to be included within your Pyramid application like so:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     config = Configurator()
5     config.include('myapp.myconfig.includeme')
```

Because the function is named `includeme`, the function name can also be omitted from the dotted name reference:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     config = Configurator()
5     config.include('myapp.myconfig')
```

Included configuration statements will be overridden by local configuration statements if an included callable causes a configuration conflict by registering something with the same configuration parameters.

If the `route_prefix` is supplied, it must be a string. Any calls to `pyramid.config.Configurator.add_route()` within the included callable will have their pattern prefixed with the value of `route_prefix`. This can be used to help mount a set of routes at a different location than the included callable's author intended, while still maintaining the same route names. For example:

```
1 from pyramid.config import Configurator
2
3 def included(config):
4     config.add_route('show_users', '/show')
5
6 def main(global_config, **settings):
7     config = Configurator()
8     config.include(included, route_prefix='/users')
```

In the above configuration, the `show_users` route will have an effective route pattern of `/users/show`, instead of `/show` because the `route_prefix` argument will be prepended to the pattern.

New in version 1.2: The `route_prefix` parameter.

make_wsgi_app()

Commits any pending configuration statements, sends a *pyramid.events.ApplicationCreated* event to all listeners, adds this configuration's registry to *pyramid.config.global_registries*, and returns a Pyramid WSGI application representing the committed configuration state.

scan (*package=None, categories=None, onerror=None, ignore=None, **kw*)

Scan a Python package and any of its subpackages for objects marked with *configuration decoration* such as *pyramid.view.view_config*. Any decorated object found will influence the current configuration state.

The *package* argument should be a Python *package* or module object (or a *dotted Python name* which refers to such a package or module). If *package* is *None*, the package of the *caller* is used.

The *categories* argument, if provided, should be the *Venusian* 'scan categories' to use during scanning. Providing this argument is not often necessary; specifying scan categories is an extremely advanced usage. By default, *categories* is *None* which will execute *all* *Venusian* decorator callbacks including Pyramid-related decorators such as *pyramid.view.view_config*. See the *Venusian* documentation for more information about limiting a scan by using an explicit set of categories.

The *onerror* argument, if provided, should be a *Venusian* *onerror* callback function. The *onerror* function is passed to *venusian.Scanner.scan()* to influence error behavior when an exception is raised during the scanning process. See the *Venusian* documentation for more information about *onerror* callbacks.

The *ignore* argument, if provided, should be a *Venusian* *ignore* value. Providing an *ignore* argument allows the scan to ignore particular modules, packages, or global objects during a scan. *ignore* can be a string or a callable, or a list containing strings or callables. The simplest usage of *ignore* is to provide a module or package by providing a full path to its dotted name. For example: *config.scan(ignore='my.module.subpackage')* would ignore the *my.module.subpackage* package during a scan, which would prevent the subpackage and any of its submodules from being imported and scanned. See the *Venusian* documentation for more information about the *ignore* argument.

To perform a *scan*, Pyramid creates a *Venusian Scanner* object. The *kw* argument represents a set of keyword arguments to pass to the *Venusian Scanner* object's constructor. See the *venusian* documentation (its *Scanner* class) for more information about the constructor. By default, the only keyword arguments passed to the *Scanner* constructor are *{'config':self}* where *self* is this configuration object. This services the requirement of all built-in Pyramid decorators, but

extension systems may require additional arguments. Providing this argument is not often necessary; it's an advanced usage.

New in version 1.1: The `**kw` argument.

New in version 1.3: The `ignore` argument.

Adding Routes and Views

add_route (*name*, *pattern=None*, *permission=None*, *factory=None*, *for_=None*, *header=None*, *xhr=None*, *accept=None*, *path_info=None*, *request_method=None*, *request_param=None*, *traverse=None*, *custom_predicates=()*, *use_global_views=False*, *path=None*, *pregenerator=None*, *static=False*, ***predicates*)

Add a *route configuration* to the current configuration state, as well as possibly a *view configuration* to be used to specify a *view callable* that will be invoked when this route matches. The arguments to this method are divided into *predicate*, *non-predicate*, and *view-related* types. *Route predicate* arguments narrow the circumstances in which a route will be match a request; non-predicate arguments are informational.

Non-Predicate Arguments

name

The name of the route, e.g. `myroute`. This attribute is required. It must be unique among all defined routes in a given application.

factory

A Python object (often a function or a class) or a *dotted Python name* which refers to the same object that will generate a Pyramid root resource object when this route matches. For example, `mypackage.resources.MyFactory`. If this argument is not specified, a default root factory will be used. See *The Resource Tree* for more information about root factories.

traverse

If you would like to cause the *context* to be something other than the *root* object when this route matches, you can spell a traversal pattern as the *traverse* argument. This traversal pattern will be used as the traversal path: traversal will begin at the root object implied by this route (either the global root, or the object returned by the *factory* associated with this route).

The syntax of the *traverse* argument is the same as it is for *pattern*. For example, if the *pattern* provided to `add_route` is `articles/{article}/edit`, and the *traverse* argument provided to `add_route` is `/ {article}`, when a request comes in that

causes the route to match in such a way that the `article` match value is `'1'` (when the request URI is `/articles/1/edit`), the traversal path will be generated as `/1`. This means that the root object's `__getitem__` will be called with the name `'1'` during the traversal phase. If the `'1'` object exists, it will become the *context* of the request. *Traversal* has more information about traversal.

If the traversal path contains segment marker names which are not present in the `pattern` argument, a runtime error will occur. The `traverse` pattern should not contain segment markers that do not exist in the `pattern` argument.

A similar combining of routing and traversal is available when a route is matched which contains a `*traverse` remainder marker in its pattern (see *Using *traverse in a Route Pattern*). The `traverse` argument to `add_route` allows you to associate route patterns with an arbitrary traversal path without using a `*traverse` remainder marker; instead you can use other match information.

Note that the `traverse` argument to `add_route` is ignored when attached to a route that has a `*traverse` remainder marker in its pattern.

`pregenerator`

This option should be a callable object that implements the `pyramid.interfaces.IRoutePregenerator` interface. A *pregenerator* is a callable called by the `pyramid.request.Request.route_url()` function to augment or replace the arguments it is passed when generating a URL for the route. This is a feature not often used directly by applications, it is meant to be hooked by frameworks that use Pyramid as a base.

`use_global_views`

When a request matches this route, and view lookup cannot find a view which has a `route_name` predicate argument that matches the route, try to fall back to using a view that otherwise matches the context, request, and view name (but which does not match the `route_name` predicate).

`static`

If `static` is `True`, this route will never match an incoming request; it will only be useful for URL generation. By default, `static` is `False`. See *Static Routes*.

New in version 1.1.

`accept`

This value represents a match query for one or more mimetypes in the Accept HTTP request header. If this value is specified, it must be in one of the following forms: a mimetype match token in the form `text/plain`, a wildcard mimetype match token in the form `text/*` or a match-all wildcard mimetype match token in the form `*/*`. If any of the forms

matches the `Accept` header of the request, or if the `Accept` header isn't set at all in the request, this will match the current route. If this does not match the `Accept` header of the request, route matching continues.

Predicate Arguments

`pattern`

The pattern of the route e.g. `ideas/{idea}`. This argument is required. See *Route Pattern Syntax* for information about the syntax of route patterns. If the pattern doesn't match the current URL, route matching continues.



For backwards compatibility purposes (as of Pyramid 1.0), a `path` keyword argument passed to this function will be used to represent the pattern value if the `pattern` argument is `None`. If both `path` and `pattern` are passed, `pattern` wins.

`xhr`

This value should be either `True` or `False`. If this value is specified and is `True`, the *request* must possess an `HTTP_X_REQUESTED_WITH` (aka `X-Requested-With`) header for this route to match. This is useful for detecting AJAX requests issued from jQuery, Prototype and other Javascript libraries. If this predicate returns `False`, route matching continues.

`request_method`

A string representing an HTTP method name, e.g. `GET`, `POST`, `HEAD`, `DELETE`, `PUT` or a tuple of elements containing HTTP method names. If this argument is not specified, this route will match if the request has *any* request method. If this predicate returns `False`, route matching continues.

Changed in version 1.2: The ability to pass a tuple of items as `request_method`. Previous versions allowed only a string.

`path_info`

This value represents a regular expression pattern that will be tested against the `PATH_INFO` WSGI environment variable. If the regex matches, this predicate will return `True`. If this predicate returns `False`, route matching continues.

`request_param`

This value can be any string. A view declaration with this argument ensures that the associated route will only match when the request has a key in the `request.params` dictionary (an HTTP `GET` or `POST` variable) that has a name which matches the supplied value. If the value supplied as the argument has a `=` sign in it, e.g. `request_param="foo=123"`, then the key (`foo`) must both exist in the `request.params` dictionary, and

the value must match the right hand side of the expression (123) for the route to “match” the current request. If this predicate returns `False`, route matching continues.

header

This argument represents an HTTP header name or a header name/value pair. If the argument contains a `:` (colon), it will be considered a name/value pair (e.g. `User-Agent:Mozilla/*` or `Host:localhost`). If the value contains a colon, the value portion should be a regular expression. If the value does not contain a colon, the entire value will be considered to be the header name (e.g. `If-Modified-Since`). If the value evaluates to a header name only without a value, the header specified by the name must be present in the request for this predicate to be true. If the value evaluates to a header name/value pair, the header specified by the name must be present in the request *and* the regular expression specified as the value must match the header value. Whether or not the value represents a header name or a header name/value pair, the case of the header name is not significant. If this predicate returns `False`, route matching continues.

effective_principals

If specified, this value should be a *principal* identifier or a sequence of principal identifiers. If the `pyramid.request.Request.effective_principals` property indicates that every principal named in the argument list is present in the current request, this predicate will return `True`; otherwise it will return `False`. For example: `effective_principals=pyramid.security.Authenticated` or `effective_principals=('fred', 'group:admins')`.

New in version 1.4a4.

custom_predicates

Deprecated since version 1.5: This value should be a sequence of references to custom predicate callables. Use custom predicates when no set of predefined predicates does what you need. Custom predicates can be combined with predefined predicates as necessary. Each custom predicate callable should accept two arguments: `info` and `request` and should return either `True` or `False` after doing arbitrary evaluation of the `info` and/or the `request`. If all custom and non-custom predicate callables return `True` the associated route will be considered viable for a given request. If any predicate callable returns `False`, route matching continues. Note that the value `info` passed to a custom route predicate is a dictionary containing matching information; see *Custom Route Predicates* for more information about `info`.

predicates

Pass a key/value pair here to use a third-party predicate registered via `pyramid.config.Configurator.add_route_predicate()`. More than one key/value pair can be used at the same time. See *View and Route Predicates* for more information about third-party predicates.

New in version 1.4.

add_static_view (*name*, *path*, ***kw*)

Add a view used to render static assets such as images and CSS files.

The *name* argument is a string representing an application-relative local URL prefix. It may alternately be a full URL.

The *path* argument is the path on disk where the static files reside. This can be an absolute path, a package-relative path, or a *asset specification*.

The *cache_max_age* keyword argument is input to set the *Expires* and *Cache-Control* headers for static assets served. Note that this argument has no effect when the *name* is a *url prefix*. By default, this argument is *None*, meaning that no particular *Expires* or *Cache-Control* headers are set in the response.

The *permission* keyword argument is used to specify the *permission* required by a user to execute the static view. By default, it is the string `pyramid.security.NO_PERMISSION_REQUIRED`, a special sentinel which indicates that, even if a *default permission* exists for the current application, the static view should be rendered to completely anonymous users. This default value is permissive because, in most web apps, static assets seldom need protection from viewing. If *permission* is specified, the security checking will be performed against the default root factory ACL.

Any other keyword arguments sent to `add_static_view` are passed on to `pyramid.config.Configurator.add_route()` (e.g. *factory*, perhaps to define a custom factory with a custom ACL for this static view).

Usage

The `add_static_view` function is typically used in conjunction with the `pyramid.request.Request.static_url()` method. `add_static_view` adds a view which renders a static asset when some URL is visited; `pyramid.request.Request.static_url()` generates a URL to that asset.

The *name* argument to `add_static_view` is usually a simple URL prefix (e.g. `'images'`). When this is the case, the `pyramid.request.Request.static_url()` API will generate a URL which points to a Pyramid view, which will serve up a set of assets that live in the package itself. For example:

```
add_static_view('images', 'mypackage:images/')
```

Code that registers such a view can generate URLs to the view via `pyramid.request.Request.static_url()`:

```
request.static_url('mypackage:images/logo.png')
```

When `add_static_view` is called with a name argument that represents a URL prefix, as it is above, subsequent calls to `pyramid.request.Request.static_url()` with paths that start with the path argument passed to `add_static_view` will generate a URL something like `http://<Pyramid app URL>/images/logo.png`, which will cause the `logo.png` file in the `images` subdirectory of the `mypackage` package to be served.

`add_static_view` can alternately be used with a name argument which is a *URL*, causing static assets to be served from an external webserver. This happens when the name argument is a fully qualified URL (e.g. starts with `http://` or similar). In this mode, the name is used as the prefix of the full URL when generating a URL using `pyramid.request.Request.static_url()`. Furthermore, if a protocol-relative URL (e.g. `//example.com/images`) is used as the name argument, the generated URL will use the protocol of the request (`http` or `https`, respectively).

For example, if `add_static_view` is called like so:

```
add_static_view('http://example.com/images',  
    ↪ 'mypackage:images/')
```

Subsequently, the URLs generated by `pyramid.request.Request.static_url()` for that static view will be prefixed with `http://example.com/images` (the external webserver listening on `example.com` must be itself configured to respond properly to such a request.):

```
static_url('mypackage:images/logo.png', request)
```

See *Serving Static Assets* for more information.

```
add_view(view=None, name='', for_=None, permission=None, request_type=None, route_name=None, request_method=None, request_param=None, containment=None, attr=None, renderer=None, wrapper=None, xhr=None, accept=None, header=None, path_info=None, custom_predicates=(), context=None, decorator=None, mapper=None, http_cache=None, match_param=None, check_csrf=None, require_csrf=None, **view_options)
```

Add a *view configuration* to the current configuration state. Arguments to `add_view` are broken down below into *predicate* arguments and *non-predicate* arguments. Predicate arguments narrow the circumstances in which the view callable will be invoked when a request is presented to Pyramid; non-predicate arguments are informational.

Non-Predicate Arguments

view

A *view callable* or a *dotted Python name* which refers to a view callable. This argument is required unless a `renderer` argument also exists. If a `renderer` argument is passed, and a `view` argument is not provided, the view callable defaults to a callable that returns an empty dictionary (see *Writing View Callables Which Use a Renderer*).

permission

A *permission* that the user must possess in order to invoke the *view callable*. See *Configuring View Security* for more information about view security and permissions. This is often a string like `view` or `edit`.

If `permission` is omitted, a *default* permission may be used for this view registration if one was named as the `pyramid.config.Configurator` constructor's `default_permission` argument, or if `pyramid.config.Configurator.set_default_permission()` was used prior to this view registration. Pass the value `pyramid.security.NO_PERMISSION_REQUIRED` as the permission argument to explicitly indicate that the view should always be executable by entirely anonymous users, regardless of the default permission, bypassing any *authorization policy* that may be in effect.

attr

This knob is most useful when the view definition is a class.

The view machinery defaults to using the `__call__` method of the *view callable* (or the function itself, if the view callable is a function) to obtain a response. The `attr` value allows you to vary the method attribute used to obtain the response. For example, if your view was a class, and

the class has a method named `index` and you wanted to use this method instead of the class' `__call__` method to return the response, you'd say `attr="index"` in the view configuration for the view.

`renderer`

This is either a single string term (e.g. `json`) or a string implying a path or *asset specification* (e.g. `templates/views.pt`) naming a *renderer* implementation. If the `renderer` value does not contain a dot `.`, the specified string will be used to look up a renderer implementation, and that renderer implementation will be used to construct a response from the view return value. If the `renderer` value contains a dot `.`, the specified term will be treated as a path, and the filename extension of the last element in the path will be used to look up the renderer implementation, which will be passed the full path. The renderer implementation will be used to construct a *response* from the view return value.

Note that if the view itself returns a *response* (see *View Callable Responses*), the specified renderer implementation is never called.

When the `renderer` is a path, although a path is usually just a simple relative pathname (e.g. `templates/foo.pt`, implying that a template named “foo.pt” is in the “templates” directory relative to the directory of the current *package* of the Configurator), a path can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternately be a *asset specification* in the form `some.dotted.package_name:relative/path`, making it possible to address template assets which live in a separate package.

The `renderer` attribute is optional. If it is not defined, the “null” renderer is assumed (no rendering is performed and the value is passed back to the upstream Pyramid machinery unmodified).

`http_cache`

New in version 1.1.

When you supply an `http_cache` value to a view configuration, the `Expires` and `Cache-Control` headers of a response generated by the associated view callable are modified. The value for `http_cache` may be one of the following:

- A nonzero integer. If it's a nonzero integer, it's treated as a number of seconds. This number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=3600` instructs the requesting browser to ‘cache this response for an hour, please’.

- A `datetime.timedelta` instance. If it's a `datetime.timedelta` instance, it will be converted into a number of seconds, and that number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=datetime.timedelta(days=1)` instructs the requesting browser to 'cache this response for a day, please'.
- Zero (0). If the value is zero, the `Cache-Control` and `Expires` headers present in all responses from this view will be composed such that client browser cache (and any intermediate caches) are instructed to never cache the response.
- A two-tuple. If it's a two tuple (e.g. `http_cache=(1, {'public':True})`), the first value in the tuple may be a nonzero integer or a `datetime.timedelta` instance; in either case this value will be used as the number of seconds to cache the response. The second value in the tuple must be a dictionary. The values present in the dictionary will be used as input to the `Cache-Control` response header. For example: `http_cache=(3600, {'public':True})` means 'cache for an hour, and add `public` to the `Cache-Control` header of the response'. All keys and values supported by the `webob.cachecontrol.CacheControl` interface may be added to the dictionary. Supplying `{'public':True}` is equivalent to calling `response.cache_control.public = True`.

Providing a non-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value)` within your view's body.

Providing a two-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value[0], **value[1])` within your view's body.

If you wish to avoid influencing, the `Expires` header, and instead wish to only influence `Cache-Control` headers, pass a tuple as `http_cache` with the first element of `None`, e.g.: `(None, {'public':True})`.

If you wish to prevent a view that uses `http_cache` in its configuration from having its caching response headers changed by this machinery, set `response.cache_control.prevent_auto = True` before returning the response from the view. This effectively disables any HTTP caching done by `http_cache` for that response.

`require_csrf`

New in version 1.7.

A boolean option or `None`. Default: `None`.

If this option is set to `True` then CSRF checks will be enabled for requests to this view. The required token or header default to `csrf_token` and `X-CSRF-Token`, respectively.

CSRF checks only affect “unsafe” methods as defined by RFC2616. By default, these methods are anything except `GET`, `HEAD`, `OPTIONS`, and `TRACE`.

The defaults here may be overridden by `pyramid.config.Configurator.set_default_csrf_options()`.

This feature requires a configured *session factory*.

If this option is set to `False` then CSRF checks will be disabled regardless of the default `require_csrf` setting passed to `set_default_csrf_options`.

See *Checking CSRF Tokens Automatically* for more information.

wrapper

The *view name* of a different *view configuration* which will receive the response body of this view as the `request.wrapped_body` attribute of its own *request*, and the *response* returned by this view as the `request.wrapped_response` attribute of its own *request*. Using a wrapper makes it possible to “chain” views together to form a composite response. The response of the outermost wrapper view will be returned to the user. The wrapper view will be found as any view is found: see *View Configuration*. The “best” wrapper view will be found based on the lookup ordering: “under the hood” this wrapper view is looked up via `pyramid.view.render_view_to_response(context, request, 'wrapper_viewname')`. The context and request of a wrapper view is the same context and request of the inner view. If this attribute is unspecified, no view wrapping is done.

decorator

A *dotted Python name* to function (or the function itself, or an iterable of the aforementioned) which will be used to decorate the registered *view callable*. The decorator function(s) will be called with the view callable as a single argument. The view callable it is passed will accept `(context, request)`. The decorator(s) must return a replacement view callable which also accepts `(context, request)`.

If decorator is an iterable, the callables will be combined and used in the order provided as a decorator. For example:

```
@view_config(...,
    decorator=(decorator2,
               decorator1))
def myview(request):
    ....
```

Is similar to doing:

```
@view_config(...)
@decorator2
@decorator1
def myview(request):
    ...
```

Except with the existing benefits of `decorator=` (having a common decorator syntax for all view calling conventions and not having to think about preserving function attributes such as `__name__` and `__module__` within decorator logic).

An important distinction is that each decorator will receive a response object implementing `pyramid.interfaces.IResponse` instead of the raw value returned from the view callable. All decorators in the chain must return a response object or raise an exception:

```
def log_timer(wrapped):
    def wrapper(context, request):
        start = time.time()
        response = wrapped(context, request)
        duration = time.time() - start
        response.headers['X-View-Time'] = '%.3f' %_
        ↪(duration,)
        log.info('view took %.3f seconds', duration)
        return response
    return wrapper
```

Changed in version 1.4a4: Passing an iterable.

`mapper`

A Python object or *dotted Python name* which refers to a *view mapper*, or `None`. By default it is `None`, which indicates that the view should use the default view mapper. This plug-point is useful for Pyramid extension developers, but it's not very useful for 'civilians' who are just developing stock Pyramid applications. Pay no attention to the man behind the curtain.

accept

This value represents a match query for one or more mimetypes in the `Accept` HTTP request header. If this value is specified, it must be in one of the following forms: a mimetype match token in the form `text/plain`, a wildcard mimetype match token in the form `text/*` or a match-all wildcard mimetype match token in the form `*/*`. If any of the forms matches the `Accept` header of the request, or if the `Accept` header isn't set at all in the request, this will match the current view. If this does not match the `Accept` header of the request, view matching continues.

Predicate Arguments**name**

The *view name*. Read *Traversal* to understand the concept of a view name.

context

An object or a *dotted Python name* referring to an interface or class object that the *context* must be an instance of, or the *interface* that the *context* must provide in order for this view to be found and called. This predicate is true when the *context* is an instance of the represented class or if the *context* provides the represented interface; it is otherwise false. This argument may also be provided to `add_view` as `for_` (an older, still-supported spelling).

route_name

This value must match the *name* of a *route configuration* declaration (see *URL Dispatch*) that must match before this view will be called.

request_type

This value should be an *interface* that the *request* must provide in order for this view to be found and called. This value exists only for backwards compatibility purposes.

request_method

This value can be either a string (such as `"GET"`, `"POST"`, `"PUT"`, `"DELETE"`, `"HEAD"` or `"OPTIONS"`) representing an HTTP `REQUEST_METHOD`, or a tuple containing one or more of these strings. A view declaration with this argument ensures that the view will only be called when the `method` attribute of the request (aka the `REQUEST_METHOD` of the WSGI environment) matches a supplied value. Note that use of `GET` also implies that the view will respond to `HEAD` as of Pyramid 1.4.

Changed in version 1.2: The ability to pass a tuple of items as `request_method`. Previous versions allowed only a string.

request_param

This value can be any string or any sequence of strings. A view declaration with this argument ensures that the view will only be called when the *request* has a key in the `request.params` dictionary (an HTTP `GET` or

POST variable) that has a name which matches the supplied value (if the value is a string) or values (if the value is a tuple). If any value supplied has a = sign in it, e.g. `request_param="foo=123"`, then the key (`foo`) must both exist in the `request.params` dictionary, *and* the value must match the right hand side of the expression (`123`) for the view to “match” the current request.

`match_param`

New in version 1.2.

This value can be a string of the format “key=value” or a tuple containing one or more of these strings.

A view declaration with this argument ensures that the view will only be called when the *request* has key/value pairs in its *matchdict* that equal those supplied in the predicate. e.g. `match_param="action=edit"` would require the `action` parameter in the *matchdict* match the right hand side of the expression (`edit`) for the view to “match” the current request.

If the `match_param` is a tuple, every key/value pair must match for the predicate to pass.

`containment`

This value should be a Python class or *interface* (or a *dotted Python name*) that an object in the *lineage* of the context must provide in order for this view to be found and called. The nodes in your object graph must be “location-aware” to use this feature. See *Location-Aware Resources* for more information about location-awareness.

`xhr`

This value should be either `True` or `False`. If this value is specified and is `True`, the *request* must possess an `HTTP_X_REQUESTED_WITH` (aka `X-Requested-With`) header that has the value `XMLHttpRequest` for this view to be found and called. This is useful for detecting AJAX requests issued from jQuery, Prototype and other Javascript libraries.

`header`

This value represents an HTTP header name or a header name/value pair. If the value contains a `:` (colon), it will be considered a name/value pair (e.g. `User-Agent:Mozilla/.*` or `Host:localhost`). The value portion should be a regular expression. If the value does not contain a colon, the entire value will be considered to be the header name (e.g. `If-Modified-Since`). If the value evaluates to a header name only without a value, the header specified by the name must be present in the request for this predicate to be true. If the value evaluates to a header name/value pair, the header specified by the name must be present in the request *and* the regular expression specified as the value must match the header value. Whether or not the value represents a header name or a header name/value pair, the case of the header name is not significant.

path_info

This value represents a regular expression pattern that will be tested against the `PATH_INFO` WSGI environment variable. If the regex matches, this predicate will be `True`.

check_csrf

Deprecated since version 1.7: Use the `require_csrf` option or see *Checking CSRF Tokens Automatically* instead to have `pyramid.exceptions.BadCSRFToken` exceptions raised.

If specified, this value should be one of `None`, `True`, `False`, or a string representing the ‘check name’. If the value is `True` or a string, CSRF checking will be performed. If the value is `False` or `None`, CSRF checking will not be performed.

If the value provided is a string, that string will be used as the ‘check name’. If the value provided is `True`, `csrf_token` will be used as the check name.

If CSRF checking is performed, the checked value will be the value of `request.params[check_name]`. This value will be compared against the value of `request.session.get_csrf_token()`, and the check will pass if these two values are the same. If the check passes, the associated view will be permitted to execute. If the check fails, the associated view will not be permitted to execute.

Note that using this feature requires a *session factory* to have been configured.

New in version 1.4a2.

physical_path

If specified, this value should be a string or a tuple representing the *physical path* of the context found via traversal for this predicate to match as true. For example: `physical_path= '/'` or `physical_path= '/a/b/c'` or `physical_path= ('', 'a', 'b', 'c')`. This is not a path prefix match or a regex, it’s a whole-path match. It’s useful when you want to always potentially show a view when some object is traversed to, but you can’t be sure about what kind of object it will be, so you can’t use the `context` predicate. The individual path elements inbetween slash characters or in tuple elements should be the Unicode representation of the name of the resource and should not be encoded in any way.

New in version 1.4a3.

effective_principals

If specified, this value should be a *principal* identifier or a sequence of principal identifiers. If the `pyramid.request.Request.effective_principals` property indicates that every principal named in the argument list is present in the current request, this predicate will return True; otherwise it will return False. For example:

```
effective_principals=pyramid.security.  
Authenticated or effective_principals=('fred',  
'group:admins').
```

New in version 1.4a4.

`custom_predicates`

Deprecated since version 1.5: This value should be a sequence of references to custom predicate callables. Use custom predicates when no set of predefined predicates do what you need. Custom predicates can be combined with predefined predicates as necessary. Each custom predicate callable should accept two arguments: `context` and `request` and should return either True or False after doing arbitrary evaluation of the context and/or the request. The `predicates` argument to this method and the ability to register third-party view predicates via `pyramid.config.Configurator.add_view_predicate()` obsoletes this argument, but it is kept around for backwards compatibility.

`view_options`:

Pass a key/value pair here to use a third-party predicate or set a value for a view deriver. See `pyramid.config.Configurator.add_view_predicate()` and `pyramid.config.Configurator.add_view_deriver()`. See *View and Route Predicates* for more information about third-party predicates and *View Derivers* for information about view derivers.

`add_notfound_view`(`view=None`, `attr=None`, `renderer=None`, `wrapper=None`, `route_name=None`, `request_type=None`, `request_method=None`, `request_param=None`, `containment=None`, `xhr=None`, `accept=None`, `header=None`, `path_info=None`, `custom_predicates=()`, `decorator=None`, `mapper=None`, `match_param=None`, `append_slash=False`, `**view_options`)

Add a default Not Found View to the current configuration state. The view will be called when Pyramid or application code raises an `pyramid.httpexceptions.HTTPNotFound` exception (e.g. when a view cannot be found for the request). The simplest example is:

```
def notfound(request):  
    return Response('Not Found', status='404 Not_  
    Found')
```

```
config.add_notfound_view(notfound)
```

If `view` argument is not provided, the view callable defaults to `default_exceptionresponse_view()`.

All arguments except `append_slash` have the same meaning as `pyramid.config.Configurator.add_view()` and each predicate argument restricts the set of circumstances under which this notfound view will be invoked. Unlike `pyramid.config.Configurator.add_view()`, this method will raise an exception if passed `name`, `permission`, `context`, `for_`, or `http_cache` keyword arguments. These argument values make no sense in the context of a Not Found View.

If `append_slash` is `True`, when this Not Found View is invoked, and the current path info does not end in a slash, the notfound logic will attempt to find a *route* that matches the request's path info suffixed with a slash. If such a route exists, Pyramid will issue a redirect to the URL implied by the route; if it does not, Pyramid will return the result of the view callable provided as `view`, as normal.

If the argument provided as `append_slash` is not a boolean but instead implements `IResponse`, the `append_slash` logic will behave as if `append_slash=True` was passed, but the provided class will be used as the response class instead of the default `HTTPFound` response class when a redirect is performed. For example:

```
from pyramid.httpexceptions import _
↳ HTTPMovedPermanently
config.add_notfound_view(append_
↳ slash=HTTPMovedPermanently)
```

The above means that a redirect to a slash-appended route will be attempted, but instead of `HTTPFound` being used, `HTTPMovedPermanently` will be used for the redirect response if a slash-appended route is found.

Changed in version 1.6: The `append_slash` argument was modified to allow any object that implements the `IResponse` interface to specify the response class used when a redirect is performed.

New in version 1.3.


```
add_forbidden_view(view=None, attr=None, renderer=None,
                    wrapper=None, route_name=None, re-
                    quest_type=None, request_method=None, re-
                    quest_param=None, containment=None, xhr=None,
                    accept=None, header=None, path_info=None, cus-
                    tom_predicates=(), decorator=None, mapper=None,
                    match_param=None, **view_options)
```

Add a forbidden view to the current configuration state. The view will be called when Pyramid or application code raises a `pyramid.httpexceptions.HTTPForbidden` exception and the set of circumstances implied by the predicates provided are matched. The simplest example is:

```
def forbidden(request):
    return Response('Forbidden', status='403',
                    ␣↳Forbidden')

config.add_forbidden_view(forbidden)
```

If `view` argument is not provided, the view callable defaults to `default_exceptionresponse_view()`.

All arguments have the same meaning as `pyramid.config.Configurator.add_view()` and each predicate argument restricts the set of circumstances under which this forbidden view will be invoked. Unlike `pyramid.config.Configurator.add_view()`, this method will raise an exception if passed `name`, `permission`, `context`, `for_`, or `http_cache` keyword arguments. These argument values make no sense in the context of a forbidden view.

New in version 1.3.

Adding an Event Subscriber

```
add_subscriber(subscriber, iface=None, **predicates)
```

Add an event *subscriber* for the event stream implied by the supplied *iface* interface.

The `subscriber` argument represents a callable object (or a *dotted Python name* which identifies a callable); it will be called with a single object *event* whenever Pyramid emits an *event* associated with the *iface*, which may be an *interface* or a class or a *dotted Python name* to a global object representing an interface or a class.

Using the default `iface` value, `None` will cause the subscriber to be registered for all event types. See *Using Events* for more information about events and subscribers.


Any number of predicate keyword arguments may be passed in `**predicates`. Each predicate named will narrow the set of circumstances in which the subscriber will be invoked. Each named predicate must have been registered via `pyramid.config.Configurator.add_subscriber_predicate()` before it can be used. See *Subscriber Predicates* for more information.

New in version 1.4: The `**predicates` argument.

Using Security


set_authentication_policy (*policy*)

Override the Pyramid *authentication policy* in the current configuration. The `policy` argument must be an instance of an authentication policy or a *dotted Python name* that points at an instance of an authentication policy.

 Using the `authentication_policy` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_authorization_policy (*policy*)

Override the Pyramid *authorization policy* in the current configuration. The `policy` argument must be an instance of an authorization policy or a *dotted Python name* that points at an instance of an authorization policy.

 Using the `authorization_policy` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_default_csrf_options (*require_csrf=True*, *token='csrf_token'*, *header='X-CSRF-Token'*, *safe_methods=('GET', 'HEAD', 'OPTIONS', 'TRACE')*)

Set the default CSRF options used by subsequent view registrations.

`require_csrf` controls whether CSRF checks will be automatically enabled on each view in the application. This value is used as the fallback when `require_csrf` is left at the default of `None` on `pyramid.config.Configurator.add_view()`.

`token` is the name of the CSRF token used in the body of the request, accessed via `request.POST[token]`. Default: `csrf_token`.

`header` is the name of the header containing the CSRF token, accessed via `request.headers[header]`. Default: `X-CSRF-Token`.

If `token` or `header` are set to `None` they will not be used for checking CSRF tokens.

`safe_methods` is an iterable of HTTP methods which are expected to not contain side-effects as defined by RFC2616. Safe methods will never be automatically checked for CSRF tokens. Default: `('GET', 'HEAD', 'OPTIONS', 'TRACE')`.

New in version 1.7.

set_default_permission (*permission*)

Set the default permission to be used by all subsequent *view configuration* registrations. `permission` should be a *permission* string to be used as the default permission. An example of a permission string: `'view'`. Adding a default permission makes it unnecessary to protect each view configuration with an explicit permission, unless your application policy requires some exception for a particular view.

If a default permission is *not* set, views represented by view configuration registrations which do not explicitly declare a permission will be executable by entirely anonymous users (any authorization policy is ignored).

Later calls to this method override will conflict with earlier calls; there can be only one default permission active at a time within an application.



If a default permission is in effect, view configurations meant to create a truly anonymously accessible view (even *exception view* views) *must* use the value of the permission importable as `pyramid.security.NO_PERMISSION_REQUIRED`. When this string is used as the `permission` for a view configuration, the default permission is ignored, and the view is registered, making it available to all callers regardless of their credentials.

See also:

See also *Setting a Default Permission*.

i Using the `default_permission` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

add_permission (*permission_name*)

A configurator directive which registers a free-standing permission without associating it with a view callable. This can be used so that the permission shows up in the introspectable data under the `permissions` category (permissions mentioned via `add_view` already end up in there). For example:

```
config = Configurator()
config.add_permission('view')
```

Extending the Request Object**add_request_method** (*callable=None*, *name=None*, *property=False*, *reify=False*)

Add a property or method to the request object.

When adding a method to the request, `callable` may be any function that receives the request object as the first parameter. If `name` is `None` then it will be computed from the name of the `callable`.

When adding a property to the request, `callable` can either be a callable that accepts the request as its single positional parameter, or it can be a property descriptor. If `name` is `None`, the name of the property will be computed from the name of the `callable`.

If the `callable` is a property descriptor a `ValueError` will be raised if `name` is `None` or `reify` is `True`.

See `pyramid.request.Request.set_property()` for more details on property vs `reify`. When `reify` is `True`, the value of `property` is assumed to also be `True`.

In all cases, `callable` may also be a *dotted Python name* which refers to either a callable or a property descriptor.

If `callable` is `None` then the method is only used to assist in conflict detection between different addons requesting the same attribute on the request object.

This is the recommended method for extending the request object and should be used in favor of providing a custom request factory via `pyramid.config.Configurator.set_request_factory()`.

New in version 1.4.

set_request_property (*callable*, *name=None*, *reify=False*)
Add a property to the request object.

Deprecated since version 1.5: `pyramid.config.Configurator.add_request_method()` should be used instead. (This method was docs-deprecated in 1.4 and issues a real deprecation warning in 1.5).

New in version 1.3.

Using I18N

add_translation_dirs (**specs*)

Add one or more *translation directory* paths to the current configuration state. The *specs* argument is a sequence that may contain absolute directory paths (e.g. `/usr/share/locale`) or *asset specification* names naming a directory path (e.g. `some.package:locale`) or a combination of the two.

Example:

```
config.add_translation_dirs('/usr/share/locale',
                           'some.package:locale')
```

The translation directories are defined as a list in which translations defined later have precedence over translations defined earlier.

If multiple *specs* are provided in a single call to `add_translation_dirs`, the directories will be inserted in the order they're provided (earlier items are trumped by later items).



Consecutive calls to `add_translation_dirs` will sort the directories such that the later calls will add folders with lower precedence than earlier calls.

set_locale_negotiator (*negotiator*)

Set the *locale negotiator* for this application. The *locale negotiator* is a callable which accepts a *request* object and which returns a *locale name*. The *negotiator* argument should be the locale negotiator implementation or a *dotted Python name* which refers to such an implementation.

Later calls to this method override earlier calls; there can be only one locale negotiator active at a time within an application. See *Activating Translation* for more information.



Using the `locale_negotiator` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

Overriding Assets

override_asset (*to_override*, *override_with*, *_override=None*)

Add a Pyramid asset override to the current configuration state.

to_override is an *asset specification* to the asset being overridden.

override_with is an *asset specification* to the asset that is performing the override. This may also be an absolute path.

See *Static Assets* for more information about asset overrides.

Getting and Adding Settings

add_settings (*settings=None*, ***kw*)

Augment the *deployment settings* with one or more key/value pairs.

You may pass a dictionary:

```
config.add_settings({'external_uri': 'http://example.  
↪com'})
```


Or a set of key/value pairs:

```
config.add_settings(external_uri='http://example.com  
↪')
```

This function is useful when you need to test code that accesses the `pyramid.registry.Registry.settings` API (or the `pyramid.config.Configurator.get_settings()` API) and which uses values from that API.

get_settings()

Return a *deployment settings* object for the current application. A deployment settings object is a dictionary-like object that contains key/value pairs based on the dictionary passed as the `settings` argument to the `pyramid.config.Configurator` constructor.

 the `pyramid.registry.Registry.settings` API performs the same duty.

Hooking Pyramid Behavior

add_renderer(*name*, *factory*)

Add a Pyramid *renderer* factory to the current configuration state.

The `name` argument is the renderer name. Use `None` to represent the default renderer (a renderer which will be used for all views unless they name another renderer specifically).

The `factory` argument is Python reference to an implementation of a *renderer* factory or a *dotted Python name* to same.

add_resource_url_adapter(*adapter*, *resource_iface*=`None`)

New in version 1.3.

When you add a traverser as described in *Changing the Traverser*, it's convenient to continue to use the `pyramid.request.Request.resource_url()` API. However, since the way traversal is done may

have been modified, the URLs that `resource_url` generates by default may be incorrect when resources are returned by a custom traverser.

If you've added a traverser, you can change how `resource_url()` generates a URL for a specific type of resource by calling this method.

The `adapter` argument represents a class that implements the `IResourceURL` interface. The class constructor should accept two arguments in its constructor (the resource and the request) and the resulting instance should provide the attributes detailed in that interface (`virtual_path` and `physical_path`, in particular).

The `resource_iface` argument represents a class or interface that the resource should possess for this url adapter to be used when `pyramid.request.Request.resource_url()` looks up a resource url adapter. If `resource_iface` is not passed, or it is passed as `None`, the url adapter will be used for every type of resource.

See *Changing How `pyramid.request.Request.resource_url()` Generates a URL* for more information.

`add_response_adapter(adapter, type_or_iface)`

When an object of type (or interface) `type_or_iface` is returned from a view callable, Pyramid will use the adapter `adapter` to convert it into an object which implements the `pyramid.interfaces.IResponse` interface. If `adapter` is `None`, an object returned of type (or interface) `type_or_iface` will itself be used as a response object.

`adapter` and `type_or_interface` may be Python objects or strings representing dotted names to importable Python global objects.

See *Changing How Pyramid Treats View Responses* for more information.

`add_traverser(adapter, iface=None)`

The superdefault *traversal* algorithm that Pyramid uses is explained in *The Traversal Algorithm*. Though it is rarely necessary, this default algorithm can be swapped out selectively for a different traversal pattern via configuration. The section entitled *Changing the Traverser* details how to create a traverser class.

For example, to override the superdefault traverser used by Pyramid, you might do something like this:


```
from myapp.traversal import MyCustomTraverser
config.add_traverser(MyCustomTraverser)
```

This would cause the Pyramid superdefault traverser to never be used; instead all traversal would be done using your `MyCustomTraverser` class, no matter which object was returned by the *root factory* of this application. Note that we passed no arguments to the `iface` keyword parameter. The default value of `iface`, `None` represents that the registered traverser should be used when no other more specific traverser is available for the object returned by the root factory.

However, more than one traversal algorithm can be active at the same time. The traverser used can depend on the result of the *root factory*. For instance, if your root factory returns more than one type of object conditionally, you could claim that an alternate traverser adapter should be used against one particular class or interface returned by that root factory. When the root factory returned an object that implemented that class or interface, a custom traverser would be used. Otherwise, the default traverser would be used. The `iface` argument represents the class of the object that the root factory might return or an *interface* that the object might implement.

To use a particular traverser only when the root factory returns a particular class:

```
config.add_traverser(MyCustomTraverser, MyRootClass)
```

When more than one traverser is active, the “most specific” traverser will be used (the one that matches the class or interface of the value returned by the root factory most closely).


Note that either `adapter` or `iface` can be a *dotted Python name* or a Python object.

See *Changing the Traverser* for more information.

add_tween (*tween_factory*, *under=None*, *over=None*)
New in version 1.2.

Add a ‘tween factory’. A *tween* (a contraction of ‘between’) is a bit of code that sits between the Pyramid router’s main request handling function and the upstream WSGI component that uses Pyramid as its ‘app’.

Tweens are a feature that may be used by Pyramid framework extensions, to provide, for example, Pyramid-specific view timing support, bookkeeping code that examines exceptions before they are returned to the upstream WSGI application, or a variety of other features. Tweens behave a bit like WSGI ‘middleware’ but they have the benefit of running in a context in which they have access to the Pyramid *application registry* as well as the Pyramid rendering machinery.

 You can view the tween ordering configured into a given Pyramid application by using the `ptweens` command. See *Displaying “Tweens”*.

The `tween_factory` argument must be a *dotted Python name* to a global object representing the tween factory.

The `under` and `over` arguments allow the caller of `add_tween` to provide a hint about where in the tween chain this tween factory should be placed when an implicit tween chain is used. These hints are only used when an explicit tween chain is not used (when the `pyramid.tweens` configuration value is not set). Allowable values for `under` or `over` (or both) are:

- None (the default).
- A *dotted Python name* to a tween factory: a string representing the dotted name of a tween factory added in a call to `add_tween` in the same configuration session.
- One of the constants `pyramid.tweens.MAIN`, `pyramid.tweens.INGRESS`, or `pyramid.tweens.EXCVIEW`.
- An iterable of any combination of the above. This allows the user to specify fallbacks if the desired tween is not included, as well as compatibility with multiple other tweens.

`under` means ‘closer to the main Pyramid application than’, `over` means ‘closer to the request ingress than’.

For example, calling `add_tween('myapp.tfactory', over=pyramid.tweens.MAIN)` will attempt to place the tween factory represented by the dotted name `myapp.tfactory` directly ‘above’ (in `ptweens` order) the main Pyramid request handler. Likewise, calling `add_tween('myapp.tfactory', over=pyramid.tweens.MAIN, under='mypkg.someothertween')` will attempt to place this tween factory ‘above’ the main handler but ‘below’ (a fictional) ‘`mypkg.someothertween`’ tween factory.

If all options for `under` (or `over`) cannot be found in the current configuration, it is an error. If some options are specified purely for compatibility with other tweens, just add a fallback of `MAIN` or `INGRESS`. For example, `under=('mypkg.someothertween', 'mypkg.someothertween2', INGRESS)`. This constraint will require the tween to be located under both the `'mypkg.someothertween'` tween, the `'mypkg.someothertween2'` tween, and `INGRESS`. If any of these is not in the current configuration, this constraint will only organize itself based on the tweens that are present.

Specifying neither `over` nor `under` is equivalent to specifying `under=INGRESS`.

Implicit tween ordering is obviously only best-effort. Pyramid will attempt to present an implicit order of tweens as best it can, but the only surefire way to get any particular ordering is to use an explicit tween order. A user may always override the implicit tween ordering by using an explicit `pyramid.tweens` configuration value setting.

`under`, and `over` arguments are ignored when an explicit tween chain is specified using the `pyramid.tweens` configuration value.

For more information, see *Registering Tweens*.

`add_route_predicate` (*name*, *factory*, *weighs_more_than=None*,
 weighs_less_than=None)

Adds a route predicate factory. The view predicate can later be named as a keyword argument to `pyramid.config.Configurator.add_route()`.

name should be the name of the predicate. It must be a valid Python identifier (it will be used as a keyword argument to `add_route`).

factory should be a *predicate factory* or *dotted Python name* which refers to a predicate factory.

See *View and Route Predicates* for more information.

New in version 1.4.

```
add_subscriber_predicate (name, factory,  
                           weighs_more_than=None,  
                           weighs_less_than=None)
```

New in version 1.4.

Adds a subscriber predicate factory. The associated subscriber predicate can later be named as a keyword argument to `pyramid.config.Configurator.add_subscriber()` in the `**predicates` anonymous keyword argument dictionary.

`name` should be the name of the predicate. It must be a valid Python identifier (it will be used as a `**predicates` keyword argument to `add_subscriber()`).

`factory` should be a *predicate factory* or *dotted Python name* which refers to a predicate factory.

See *Subscriber Predicates* for more information.

```
add_view_predicate (name, factory, weighs_more_than=None,  
                    weighs_less_than=None)
```

New in version 1.4.

Adds a view predicate factory. The associated view predicate can later be named as a keyword argument to `pyramid.config.Configurator.add_view()` in the `predicates` anonymous keyword argument dictionary.

`name` should be the name of the predicate. It must be a valid Python identifier (it will be used as a keyword argument to `add_view` by others).

`factory` should be a *predicate factory* or *dotted Python name* which refers to a predicate factory.

See *View and Route Predicates* for more information.

```
add_view_deriver (deriver, name=None, under=None,  
                  over=None)
```

New in version 1.7.

Add a *view deriver* to the view pipeline. View derivers are a feature used by extension authors to wrap views in custom code controllable by view-specific options.

`deriver` should be a callable conforming to the `pyramid.interfaces.IViewDeriver` interface.

`name` should be the name of the view deriver. There are no restrictions on the name of a view deriver. If left unspecified, the name will be constructed from the name of the `deriver`.

The `under` and `over` options can be used to control the ordering of view derivers by providing hints about where in the view pipeline the deriver is used. Each option may be a string or a list of strings. At least one view deriver in each, the `over` and `under` directions, must exist to fully satisfy the constraints.

`under` means closer to the user-defined *view callable*, and `over` means closer to view pipeline ingress.

The default value for `over` is `rendered_view` and `under` is `decorated_view`. This places the deriver somewhere between the two in the view pipeline. If the deriver should be placed elsewhere in the pipeline, such as above `decorated_view`, then you **MUST** also specify `under` to something earlier in the order, or a `CyclicDependencyError` will be raised when trying to sort the derivers.

See *View Derivers* for more information.

set_request_factory (*factory*)

The object passed as `factory` should be an object (or a *dotted Python name* which refers to an object) which will be used by the Pyramid router to create all request objects. This factory object must have the same methods and attributes as the `pyramid.request.Request` class (particularly `__call__`, and `blank`).


See `pyramid.config.Configurator.add_request_method()` for a less intrusive way to extend the request objects with custom methods and properties.



Using the `request_factory` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.


set_root_factory (*factory*)

Add a *root factory* to the current configuration state. If the *factory* argument is `None` a default root factory will be registered.

 Using the `root_factory` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_session_factory (*factory*)

Configure the application with a *session factory*. If this method is called, the *factory* argument must be a session factory callable or a *dotted Python name* to that factory.

 Using the `session_factory` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.


set_view_mapper (*mapper*)

Setting a *view mapper* makes it possible to make use of *view callable* objects which implement different call signatures than the ones supported by Pyramid as described in its narrative documentation.

The *mapper* argument should be an object implementing `pyramid.interfaces.IViewMapperFactory` or a *dotted Python name* to such an object. The provided *mapper* will become the default view mapper to be used by all subsequent *view configuration* registrations.

See also:

See also *Using a View Mapper*.

 Using the `default_view_mapper` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

action (*discriminator*, *callable=None*, *args=()*, *kw=None*, *order=0*, *introspectables=()*, ***extra*)

Register an action which will be executed when `pyramid.config.Configurator.commit()` is called (or executed immediately if `autocommit` is `True`).



This method is typically only used by Pyramid framework extension authors, not by Pyramid application developers.

The `discriminator` uniquely identifies the action. It must be given, but it can be `None`, to indicate that the action never conflicts. It must be a hashable value.

The `callable` is a callable object which performs the task associated with the action when the action is executed. It is optional.

`args` and `kw` are tuple and dict objects respectively, which are passed to `callable` when this action is executed. Both are optional.

`order` is a grouping mechanism; an action with a lower order will be executed before an action with a higher order (has no effect when `autocommit` is `True`).

`introspectables` is a sequence of *introspectable* objects (or the empty sequence if no introspectable objects are associated with this action). If this configurator's `introspection` attribute is `False`, these introspectables will be ignored.

`extra` provides a facility for inserting extra keys and values into an action dictionary.

add_directive (*name*, *directive*, *action_wrap=True*)

Add a directive method to the configurator.



This method is typically only used by Pyramid framework extension authors, not by Pyramid application developers.

Framework extenders can add directive methods to a configurator by instructing their users to call `config.add_directive('somename', 'some.callable')`. This will make `some.callable` accessible as `config.somename.some.callable` should be a function which accepts `config` as a first argument, and arbitrary positional and keyword arguments following. It should use `config.action` as necessary to perform actions. Directive methods can then be invoked like ‘built-in’ directives such as `add_view`, `add_route`, etc.

The `action_wrap` argument should be `True` for directives which perform `config.action` with potentially conflicting discriminators. `action_wrap` will cause the directive to be wrapped in a decorator which provides more accurate conflict cause information.

`add_directive` does not participate in conflict detection, and later calls to `add_directive` will override earlier calls.

`with_package` (*package*)

Return a new `Configurator` instance with the same registry as this configurator. *package* may be an actual Python package object or a *dotted Python name* representing a package.

`derive_view` (*view*, *attr=None*, *renderer=None*)

Create a *view callable* using the function, instance, or class (or *dotted Python name* referring to the same) provided as *view* object.



This method is typically only used by Pyramid framework extension authors, not by Pyramid application developers.

This API is useful to framework extenders who create pluggable systems which need to register ‘proxy’ view callables for functions, instances, or classes which meet the requirements of being a Pyramid view callable. For example, a `some_other_framework` function in another framework may want to allow a user to supply a view callable, but he may want to wrap the view callable in his own before registering the wrapper as a Pyramid view callable. Because a Pyramid view callable can be any of a number of valid objects, the framework extender will not know how to call the user-supplied object. Running it through `derive_view` normalizes it to a callable which accepts two arguments: `context` and `request`.

For example:


```
def some_other_framework(user_supplied_view):
    config = Configurator(reg)
    proxy_view = config.derive_view(user_supplied_
↪view)
    def my_wrapper(context, request):
        do_something_that_mutates(request)
        return proxy_view(context, request)
    config.add_view(my_wrapper)
```

The `view` object provided should be one of the following:

- A function or another non-class callable object that accepts a *request* as a single positional argument and which returns a *response* object.
- A function or other non-class callable object that accepts two positional arguments, `context`, `request` and which returns a *response* object.
- A class which accepts a single positional argument in its constructor named `request`, and which has a `__call__` method that accepts no arguments that returns a *response* object.
- A class which accepts two positional arguments named `context`, `request`, and which has a `__call__` method that accepts no arguments that returns a *response* object.
- A *dotted Python name* which refers to any of the kinds of objects above.

This API returns a callable which accepts the arguments `context`, `request` and which returns the result of calling the provided `view` object.

The `attr` keyword argument is most useful when the `view` object is a class. It names the method that should be used as the callable. If `attr` is not provided, the attribute effectively defaults to `__call__`. See *Defining a View Callable as a Class* for more information.

The `renderer` keyword argument should be a *renderer* name. If supplied, it will cause the returned callable to use a *renderer* to convert the user-supplied `view` result to a *response* object. If a `renderer` argument is not supplied, the user-supplied `view` must itself return a *response* object.

Utility Methods

`absolute_asset_spec` (*relative_spec*)

Resolve the potentially relative *asset specification* string passed as `relative_spec` into an absolute asset specification string and return the string. Use the package of this configurator as the package to which

the asset specification will be considered relative when generating an absolute asset specification. If the provided `relative_spec` argument is already absolute, or if the `relative_spec` is not a string, it is simply returned.

maybe_dotted (*dotted*)

Resolve the *dotted Python name* `dotted` to a global Python object. If `dotted` is not a string, return it without attempting to do any name resolution. If `dotted` is a relative dotted name (e.g. `.foo.bar`, consider it relative to the `package` argument supplied to this Configurator's constructor.

ZCA-Related APIs

hook_zca ()

Call `zope.component.getSiteManager.sethook()` with the argument `pyramid.threadlocal.get_current_registry`, causing the Zope Component Architecture 'global' APIs such as `zope.component.getSiteManager()`, `zope.component.getAdapter()` and others to use the Pyramid *application registry* rather than the Zope 'global' registry.

unhook_zca ()

Call `zope.component.getSiteManager.reset()` to undo the action of `pyramid.config.Configurator.hook_zca()`.

setup_registry (*settings=None, root_factory=None, authentication_policy=None, authorization_policy=None, renderers=None, debug_logger=None, locale_negotiator=None, request_factory=None, response_factory=None, default_permission=None, session_factory=None, default_view_mapper=None, exceptionresponse_view=<function>, default_fault_exceptionresponse_view=<function>*)

When you pass a non-None registry argument to the *Configurator* constructor, no initial setup is performed against the registry. This is because the registry you pass in may have already been initialized for use under Pyramid via a different configurator. However, in some circumstances (such as when you want to use a global registry instead of a registry created as a result of the Configurator constructor), or when you want to reset the initial setup of a registry, you *do* want to explicitly initialize the registry associated with a Configurator for use under Pyramid. Use `setup_registry` to do this initialization.

`setup_registry` configures settings, a root factory, security policies, renderers, a debug logger, a locale negotiator, and various other settings using the configurator's current registry, as per the descriptions in the Configurator constructor.

Testing Helper APIs

testing_add_renderer (*path*, *renderer=None*)

Unit/integration testing helper: register a renderer at *path* (usually a relative filename ala `templates/foo.pt` or an asset specification) and return the renderer object. If the *renderer* argument is `None`, a 'dummy' renderer will be used. This function is useful when testing code that calls the `pyramid.renderers.render()` function or `pyramid.renderers.render_to_response()` function or any other `render_*` or `get_*` API of the `pyramid.renderers` module.

Note that calling this method for with a *path* argument representing a renderer factory type (e.g. for `foo.pt` usually implies the `chameleon_zpt` renderer factory) clobbers any existing renderer factory registered for that type.



This method is also available under the alias `testing_add_template` (an older name for it).

testing_add_subscriber (*event_iface=None*)

Unit/integration testing helper: Registers a *subscriber* which listens for events of the type *event_iface*. This method returns a list object which is appended to by the subscriber whenever an event is captured.

When an event is dispatched that matches the value implied by the *event_iface* argument, that event will be appended to the list. You can then compare the values in the list to expected event notifications. This method is useful when testing code that wants to call `pyramid.registry.Registry.notify()`, or `zope.component.event.dispatch()`.

The default value of *event_iface* (`None`) implies a subscriber registered for *any* kind of event.

testing_resources (*resources*)

Unit/integration testing helper: registers a dictionary of *resource* objects that can be resolved via the `pyramid.traversal.find_resource()` API.

The `pyramid.traversal.find_resource()` API is called with a path as one of its arguments. If the dictionary you register when calling this method contains that path as a string key (e.g. `/foo/bar` or `foo/bar`), the corresponding value will be returned to `find_resource` (and thus to your code) when `pyramid.traversal.find_resource()` is called with an equivalent path string or tuple.

testing_securitypolicy (*userid=None, groupids=(), permissive=True, remember_result=None, forget_result=None*)

Unit/integration testing helper: Registers a pair of faux Pyramid security policies: a *authentication policy* and a *authorization policy*.

The behavior of the registered *authorization policy* depends on the *permissive* argument. If *permissive* is true, a permissive *authorization policy* is registered; this policy allows all access. If *permissive* is false, a nonpermissive *authorization policy* is registered; this policy denies all access.

remember_result, if provided, should be the result returned by the `remember` method of the faux authentication policy. If it is not provided (or it is provided, and is `None`), the default value `[]` (the empty list) will be returned by `remember`.

forget_result, if provided, should be the result returned by the `forget` method of the faux authentication policy. If it is not provided (or it is provided, and is `None`), the default value `[]` (the empty list) will be returned by `forget`.

The behavior of the registered *authentication policy* depends on the values provided for the *userid* and *groupids* argument. The authentication policy will return the *userid* identifier implied by the *userid* argument and the *group ids* implied by the *groupids* argument when the `pyramid.request.Request.authenticated_userid` or `pyramid.request.Request.effective_principals` APIs are used.

This function is most useful when testing code that uses the APIs named `pyramid.request.Request.has_permission()`, `pyramid.request.Request.authenticated_userid`, `pyramid.request.Request.effective_principals`, and `pyramid.security.principals_allowed_by_permission()`.

New in version 1.4: The `remember_result` argument.

New in version 1.4: The `forget_result` argument.

Attributes

introspectable

A shortcut attribute which points to the `pyramid.registry.Introspectable` class (used during directives to provide introspection to actions).

New in version 1.3.

introspector

The *introspector* related to this configuration. It is an instance implementing the `pyramid.interfaces.IIntrospector` interface.

New in version 1.3.

registry

The *application registry* which holds the configuration associated with this configurator.

global_registries

The set of registries that have been created for Pyramid applications, one for each call to `pyramid.config.Configurator.make_wsgi_app()` in the current process. The object itself supports iteration and has a `last` property containing the last registry loaded.

The registries contained in this object are stored as weakrefs, thus they will only exist for the lifetime of the actual applications for which they are being used.

class not_(value)

You can invert the meaning of any predicate value by wrapping it in a call to `pyramid.config.not_`.

```

1 from pyramid.config import not_
2
3 config.add_view(
4     'mypackage.views.my_view',
5     route_name='ok',
6     request_method=not_('POST')
7 )

```

The above example will ensure that the view is called if the request method is *not* POST, at least if no other view is more specific.

This technique of wrapping a predicate value in `not_` can be used anywhere predicate values are accepted:

- `pyramid.config.Configurator.add_view()`
- `pyramid.config.Configurator.add_route()`
- `pyramid.config.Configurator.add_subscriber()`
- `pyramid.view.view_config()`
- `pyramid.events.subscriber()`

New in version 1.5.

PHASE0_CONFIG

PHASE1_CONFIG

PHASE2_CONFIG

PHASE3_CONFIG

pyramid.decorator

reify (*wrapped*)

Use as a class method decorator. It operates almost exactly like the Python `@property` decorator, but it puts the result of the method it decorates into the instance dict after the first call, effectively replacing the function it decorates with an instance variable. It is, in Python parlance, a non-data descriptor. The following is an example and its usage:

```
>>> from pyramid.decorator import reify

>>> class Foo(object):
...     @reify
...     def jammy(self):
...         print('jammy called')
...         return 1

>>> f = Foo()
>>> v = f.jammy
jammy called
>>> print(v)
1
>>> f.jammy
1
>>> # jammy func not called the second time; it replaced itself with 1
>>> # Note: reassignment is possible
>>> f.jammy = 2
>>> f.jammy
2
```

pyramid.events

Functions

subscriber (*ifaces, **predicates)

Decorator activated via a *scan* which treats the function being decorated as an event subscriber for the set of interfaces passed as **ifaces* and the set of predicate terms passed as ***predicates* to the decorator constructor.

For example:

```
from pyramid.events import NewRequest
from pyramid.events import subscriber

@subscriber(NewRequest)
def mysubscriber(event):
    event.request.foo = 1
```

More than one event type can be passed as a constructor argument. The decorated subscriber will be called for each event type.

```
from pyramid.events import NewRequest, NewResponse
from pyramid.events import subscriber

@subscriber(NewRequest, NewResponse)
def mysubscriber(event):
    print(event)
```

When the `subscriber` decorator is used without passing an arguments, the function it decorates is called for every event sent:

```
from pyramid.events import subscriber

@subscriber()
def mysubscriber(event):
    print(event)
```

This method will have no effect until a *scan* is performed against the package or module which contains it, ala:

```
from pyramid.config import Configurator
config = Configurator()
config.scan('somepackage_containing_subscribers')
```

Any ***predicate* arguments will be passed along to `pyramid.config.Configurator.add_subscriber()`. See *Subscriber Predicates* for a description of how predicates can narrow the set of circumstances in which a subscriber will be called.

Event Types

class `ApplicationCreated` (*app*)

An instance of this class is emitted as an *event* when the `pyramid.config.Configurator.make_wsgi_app()` is called. The instance has an attribute, `app`, which is an instance of the *router* that will handle WSGI requests. This class implements the `pyramid.interfaces.IApplicationCreated` interface.



For backwards compatibility purposes, this class can also be imported as `pyramid.events.WSGIApplicationCreatedEvent`. This was the name of the event class before Pyramid 1.0.

class NewRequest (*request*)


An instance of this class is emitted as an *event* whenever Pyramid begins to process a new request. The event instance has an attribute, *request*, which is a *request* object. This event class implements the *pyramid.interfaces.INewRequest* interface.

class ContextFound (*request*)

An instance of this class is emitted as an *event* after the Pyramid *router* finds a *context* object (after it performs traversal) but before any view code is executed. The instance has an attribute, *request*, which is the request object generated by Pyramid.

Notably, the request object will have an attribute named *context*, which is the context that will be provided to the view which will eventually be called, as well as other attributes attached by context-finding code.

This class implements the *pyramid.interfaces.IContextFound* interface.

 As of Pyramid 1.0, for backwards compatibility purposes, this event may also be imported as *pyramid.events.AfterTraversal*.

class BeforeTraversal (*request*)

An instance of this class is emitted as an *event* after the Pyramid *router* has attempted to find a *route* object but before any traversal or view code is executed. The instance has an attribute, *request*, which is the request object generated by Pyramid.

Notably, the request object **may** have an attribute named *matched_route*, which is the matched route if found. If no route matched, this attribute is not available.

This class implements the *pyramid.interfaces.IBeforeTraversal* interface.

class NewResponse (*request, response*)

An instance of this class is emitted as an *event* whenever any Pyramid *view* or *exception view* returns a *response*.

The instance has two attributes: *request*, which is the request which caused the response, and *response*, which is the response object returned by a view or renderer.

If the *response* was generated by an *exception view*, the request will have an attribute named *exception*, which is the exception object which caused the exception view to be executed. If the response was generated by a 'normal' view, this attribute of the request will be *None*.

This event will not be generated if a response cannot be created due to an exception that is not caught by an exception view (no response is created under this circumstance).

This class implements the `pyramid.interfaces.INewResponse` interface.

i Postprocessing a response is usually better handled in a WSGI *middleware* component than in subscriber code that is called by a `pyramid.interfaces.INewResponse` event. The `pyramid.interfaces.INewResponse` event exists almost purely for symmetry with the `pyramid.interfaces.INewRequest` event.

class BeforeRender (*system, rendering_val=None*)

Subscribers to this event may introspect and modify the set of *renderer globals* before they are passed to a *renderer*. This event object itself has a dictionary-like interface that can be used for this purpose. For example:

```
from pyramid.events import subscriber
from pyramid.events import BeforeRender

@subscriber(BeforeRender)
def add_global(event):
    event['mykey'] = 'foo'
```

An object of this type is sent as an event just before a *renderer* is invoked.

If a subscriber adds a key via `__setitem__` that already exists in the *renderer globals* dictionary, it will overwrite the older value there. This can be problematic because event subscribers to the `BeforeRender` event do not possess any relative ordering. For maximum interoperability with other third-party subscribers, if you write an event subscriber meant to be used as a `BeforeRender` subscriber, your subscriber code will need to ensure no value already exists in the *renderer globals* dictionary before setting an overriding value (which can be done using `.get` or `__contains__` of the event object).

The dictionary returned from the view is accessible through the `rendering_val` attribute of a `BeforeRender` event.

Suppose you return `{ 'mykey': 'somevalue', 'mykey2': 'somevalue2' }` from your view callable, like so:

```
from pyramid.view import view_config

@view_config(renderer='some_renderer')
def myview(request):
    return { 'mykey': 'somevalue', 'mykey2': 'somevalue2' }
```

`rendering_val` can be used to access these values from the `BeforeRender` object:

```
from pyramid.events import subscriber
from pyramid.events import BeforeRender

@subscriber(BeforeRender)
def read_return(event):
    # {'mykey': 'somevalue'} is returned from the view
    print(event.rendering_val['mykey'])
```

In other words, `rendering_val` is the (non-system) value returned by a view or passed to `render*` as value. This feature is new in Pyramid 1.2.

For a description of the values present in the renderer globals dictionary, see *System Values Used During Rendering*.

See also:

See also `pyramid.interfaces.IBeforeRender`.

update (*E*, ***F*)

Update *D* from dict/iterable *E* and *F*. If *E* has a `.keys()` method, does: for *k* in *E*: *D*[*k*] = *E*[*k*]
If *E* lacks `.keys()` method, does: for (*k*, *v*) in *E*: *D*[*k*] = *v*. In either case, this is followed by:
for *k* in *F*: *D*[*k*] = *F*[*k*].

clear () → None. Remove all items from *D*.

copy () → a shallow copy of *D*

fromkeys ()

Returns a new dict with keys from iterable and values equal to value.

get (*k*[, *d*]) → *D*[*k*] if *k* in *D*, else *d*. *d* defaults to None.

items () → a set-like object providing a view on *D*'s items

keys () → a set-like object providing a view on *D*'s keys

pop (*k*[, *d*]) → *v*, remove specified key and return the corresponding value.
If key is not found, *d* is returned if given, otherwise `KeyError` is raised

popitem () → (*k*, *v*), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if *D* is empty.

setdefault (*k*[, *d*]) → *D*.get(*k*,*d*), also set *D*[*k*]=*d* if *k* not in *D*

values () → an object providing a view on *D*'s values

See *Using Events* for more information about how to register code which subscribes to these events.

pyramid.exceptions

exception BadCSRFOrigin (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

This exception indicates the request has failed cross-site request forgery origin validation.

exception BadCSRFToken (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

This exception indicates the request has failed cross-site request forgery token validation.

exception PredicateMismatch (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

This exception is raised by multiviews when no view matches all given predicates.

This exception subclasses the `HTTPNotFound` exception for a specific reason: if it reaches the main exception handler, it should be treated as `HTTPNotFound` by any exception view registrations. Thus, typically, this exception will not be seen publicly.

However, this exception will be raised if the predicates of all views configured to handle another exception context cannot be successfully matched. For instance, if a view is configured to handle a context of `HTTPForbidden` and the configured with additional predicates, then *PredicateMismatch* will be raised if:

- An original view callable has raised `HTTPForbidden` (thus invoking an exception view); and
- The given request fails to match all predicates for said exception view associated with `HTTPForbidden`.

The same applies to any type of exception being handled by an exception view.

Forbidden

alias of `HTTPForbidden`

NotFound

alias of `HTTPNotFound`

exception ConfigurationError

Raised when inappropriate input values are supplied to an API method of a *Configurator*

exception URLDecodeError

This exception is raised when Pyramid cannot successfully decode a URL or a URL path segment. This exception behaves just like the Python builtin `UnicodeDecodeError`. It is a subclass of the builtin `UnicodeDecodeError` exception only for identity purposes, mostly so an exception view can be registered when a URL cannot be decoded.

`pyramid.httpexceptions`

HTTP Exceptions

This module contains Pyramid HTTP exception classes. Each class relates to a single HTTP status code. Each class is a subclass of the *HTTPException*. Each exception class is also a *response* object.

Each exception class has a status code according to **RFC 2068**: codes with 100-300 are not really errors; 400s are client errors, and 500s are server errors.

Exception

HTTPException

HTTPSuccessful

- 200 - HTTPOk
- 201 - HTTPCreated
- 202 - HTTPAccepted
- 203 - HTTPNonAuthoritativeInformation
- 204 - HTTPNoContent
- 205 - HTTPResetContent
- 206 - HTTPPartialContent

HTTPRedirection

- 300 - HTTPMultipleChoices
- 301 - HTTPMovedPermanently
- 302 - HTTPFound
- 303 - HTTPSeeOther
- 304 - HTTPNotModified
- 305 - HTTPUseProxy

- 307 - HTTPTemporaryRedirect

HTTPError

HTTPClientError

- 400 - HTTPBadRequest
- 401 - HTTPUnauthorized
- 402 - HTTPPaymentRequired
- 403 - HTTPForbidden
- 404 - HTTPNotFound
- 405 - HTTPMethodNotAllowed
- 406 - HTTPNotAcceptable
- 407 - HTTPProxyAuthenticationRequired
- 408 - HTTPRequestTimeout
- 409 - HTTPConflict
- 410 - HTTPGone
- 411 - HTTPLengthRequired
- 412 - HTTPPreconditionFailed
- 413 - HTTPRequestEntityTooLarge
- 414 - HTTPRequestURITooLong
- 415 - HTTPUnsupportedMediaType
- 416 - HTTPRequestRangeNotSatisfiable
- 417 - HTTPExpectationFailed
- 422 - HTTPUnprocessableEntity

- 423 - HTTPLocked
- 424 - HTTPFailedDependency
- 428 - HTTPPreconditionRequired
- 429 - HTTPTooManyRequests
- 431 - HTTPRequestHeaderFieldsTooLarge

HTTPServerError

- 500 - HTTPInternalServerError
- 501 - HTTPNotImplemented
- 502 - HTTPBadGateway
- 503 - HTTPServiceUnavailable
- 504 - HTTPGatewayTimeout
- 505 - HTTPVersionNotSupported
- 507 - HTTPInsufficientStorage

HTTP exceptions are also *response* objects, thus they accept most of the same parameters that can be passed to a regular *Response*. Each HTTP exception also has the following attributes:

code the HTTP status code for the exception

title remainder of the status line (stuff after the code)

explanation a plain-text explanation of the error message that is not subject to environment or header substitutions; it is accessible in the template via `${explanation}`

detail a plain-text message customization that is not subject to environment or header substitutions; accessible in the template via `${detail}`

body_template a `String.template`-format content fragment used for environment and header substitution; the default template includes both the explanation and further detail provided in the message.

Each HTTP exception accepts the following parameters, any others will be forwarded to its *Response* superclass:

detail a plain-text override of the default `detail`

headers a list of (k,v) header pairs

comment a plain-text additional information which is usually stripped/hidden for end-users

body_template a `string.Template` object containing a content fragment in HTML that frames the explanation and further detail

body a string that will override the `body_template` and be used as the body of the response.

Substitution of response headers into template values is always performed. Substitution of WSGI environment values is performed if a `request` is passed to the exception's constructor.

The subclasses of `_HTTPMove` (`HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPFound`, `HTTPSeeOther`, `HTTPUseProxy` and `HTTPTemporaryRedirect`) are redirections that require a `Location` field. Reflecting this, these subclasses have one additional keyword argument: `location`, which indicates the location to which to redirect.

status_map

A mapping of integer status code to HTTP exception class (eg. the integer "401" maps to `pyramid.httpexceptions.HTTPUnauthorized`). All mapped exception classes are children of `pyramid.httpexceptions`,

exception_response (`status_code`, `**kw`)

Creates an HTTP exception based on a status code. Example:

```
raise exception_response(404) # raises an HTTPNotFound exception.
```

The values passed as `kw` are provided to the exception's constructor.

exception HTTPException (`detail=None`, `headers=None`, `comment=None`,
`body_template=None`, `json_formatter=None`, `**kw`)

exception HTTPOk (`detail=None`, `headers=None`, `comment=None`, `body_template=None`,
`json_formatter=None`, `**kw`)
subclass of `HTTPSuccessful`

Indicates that the request has succeeded.

code: 200, title: OK

exception HTTPRedirection (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
base class for exceptions with status codes in the 300s (redirections)

This is an abstract base class for 3xx redirection. It indicates that further action needs to be taken by the user agent in order to fulfill the request. It does not necessarily signal an error condition.

exception HTTPError (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
base class for exceptions with status codes in the 400s and 500s

This is an exception which indicates that an error has occurred, and that any work in progress should not be committed.

exception HTTPClientError (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
base class for the 400s, where the client is in error

This is an error condition in which the client is presumed to be in-error. This is an expected problem, and thus is not considered a bug. A server-side traceback is not warranted. Unless specialized, this is a '400 Bad Request'

exception HTTPServerError (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
base class for the 500s, where the server is in-error

This is an error condition in which the server is presumed to be in-error. Unless specialized, this is a '500 Internal Server Error'.

exception HTTPCreated (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of HTTPSuccessful

This indicates that request has been fulfilled and resulted in a new resource being created.

code: 201, title: Created

exception HTTPAccepted (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of HTTPSuccessful

This indicates that the request has been accepted for processing, but the processing has not been completed.

code: 202, title: Accepted

exception HTTPNonAuthoritativeInformation (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of HTTPSuccessful

This indicates that the returned metainformation in the entity-header is not the definitive set as available from the origin server, but is gathered from a local or a third-party copy.

code: 203, title: Non-Authoritative Information

exception HTTPNoContent (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of HTTPSuccessful

This indicates that the server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation.

code: 204, title: No Content

exception HTTPResetContent (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of HTTPSuccessful

This indicates that the server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be sent.

code: 205, title: Reset Content

exception HTTPPartialContent (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of HTTPSuccessful

This indicates that the server has fulfilled the partial GET request for the resource.

code: 206, title: Partial Content

exception HTTPMultipleChoices (*location='', detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of _HTTPMove

This indicates that the requested resource corresponds to any one of a set of representations, each with its own specific location, and agent-driven negotiation information is being provided so that the user can select a preferred representation and redirect its request to that location.

code: 300, title: Multiple Choices

exception HTTPMovedPermanently (*location*='', *detail*=None, *headers*=None, *comment*=None, *body_template*=None, **kw)

subclass of `_HTTPMove`

This indicates that the requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs.

code: 301, title: Moved Permanently

exception HTTPFound (*location*='', *detail*=None, *headers*=None, *comment*=None, *body_template*=None, **kw)

subclass of `_HTTPMove`

This indicates that the requested resource resides temporarily under a different URI.

code: 302, title: Found

exception HTTPSeeOther (*location*='', *detail*=None, *headers*=None, *comment*=None, *body_template*=None, **kw)

subclass of `_HTTPMove`

This indicates that the response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource.

code: 303, title: See Other

exception HTTPNotModified (*detail*=None, *headers*=None, *comment*=None, *body_template*=None, *json_formatter*=None, **kw)

subclass of `HTTPRedirection`

This indicates that if the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server SHOULD respond with this status code.

code: 304, title: Not Modified

exception HTTPUseProxy (*location*='', *detail*=None, *headers*=None, *comment*=None, *body_template*=None, **kw)

subclass of `_HTTPMove`

This indicates that the requested resource MUST be accessed through the proxy given by the Location field.

code: 305, title: Use Proxy

exception HTTPTemporaryRedirect (*location='', detail=None, headers=None, comment=None, body_template=None, **kw*)

subclass of `_HTTPMove`

This indicates that the requested resource resides temporarily under a different URI.

code: 307, title: Temporary Redirect

exception HTTPBadRequest (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of `HTTPClientError`

This indicates that the body or headers failed validity checks, preventing the server from being able to continue processing.

code: 400, title: Bad Request

exception HTTPUnauthorized (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of `HTTPClientError`

This indicates that the request requires user authentication.

code: 401, title: Unauthorized

exception HTTPPaymentRequired (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of `HTTPClientError`

code: 402, title: Payment Required

exception HTTPForbidden (*detail=None, headers=None, comment=None, body_template=None, result=None, **kw*)

subclass of `HTTPClientError`

This indicates that the server understood the request, but is refusing to fulfill it.

code: 403, title: Forbidden

Raise this exception within *view* code to immediately return the *forbidden view* to the invoking user. Usually this is a basic 403 page, but the forbidden view can be customized as necessary. See *Changing the Forbidden View*. A `Forbidden` exception will be the `context` of a *Forbidden View*.

This exception's constructor treats two arguments specially. The first argument, `detail`, should be a string. The value of this string will be used as the `message` attribute of the exception object. The second special keyword argument, `result` is usually an instance of `pyramid.security.Denied` or `pyramid.security.ACLDenied` each of which indicates a reason for the forbidden error. However, `result` is also permitted to be just a plain boolean `False` object or `None`. The `result` value will be used as the `result` attribute of the exception object. It defaults to `None`.

The *Forbidden View* can use the attributes of a Forbidden exception as necessary to provide extended information in an error report shown to a user.

exception HTTPNotFound (`detail=None, headers=None, comment=None,`
`body_template=None, json_formatter=None, **kw`)
subclass of `HTTPClientError`

This indicates that the server did not find anything matching the Request-URI.

code: 404, title: Not Found

Raise this exception within *view* code to immediately return the *Not Found View* to the invoking user. Usually this is a basic 404 page, but the Not Found View can be customized as necessary. See *Changing the Not Found View*.

This exception's constructor accepts a `detail` argument (the first argument), which should be a string. The value of this string will be available as the `message` attribute of this exception, for availability to the *Not Found View*.

exception HTTPMethodNotAllowed (`detail=None, headers=None, comment=None,`
`body_template=None, json_formatter=None, **kw`)
subclass of `HTTPClientError`

This indicates that the method specified in the Request-Line is not allowed for the resource identified by the Request-URI.

code: 405, title: Method Not Allowed

exception HTTPNotAcceptable (`detail=None, headers=None, comment=None,`
`body_template=None, json_formatter=None, **kw`)
subclass of `HTTPClientError`

This indicates the resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

code: 406, title: Not Acceptable

exception HTTPProxyAuthenticationRequired (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This is similar to 401, but indicates that the client must first authenticate itself with the proxy.

code: 407, title: Proxy Authentication Required

exception HTTPRequestTimeout (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the client did not produce a request within the time that the server was prepared to wait.

code: 408, title: Request Timeout

exception HTTPConflict (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the request could not be completed due to a conflict with the current state of the resource.

code: 409, title: Conflict

exception HTTPGone (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the requested resource is no longer available at the server and no forwarding address is known.

code: 410, title: Gone

exception HTTPLengthRequired (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the server refuses to accept the request without a defined Content-Length.

code: 411, title: Length Required

exception HTTPPreconditionFailed (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the precondition given in one or more of the request-header fields evaluated to false when it was tested on the server.

code: 412, title: Precondition Failed

exception HTTPRequestEntityTooLarge (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the server is refusing to process a request because the request entity is larger than the server is willing or able to process.

code: 413, title: Request Entity Too Large

exception HTTPRequestURITooLong (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the server is refusing to service the request because the Request-URI is longer than the server is willing to interpret.

code: 414, title: Request-URI Too Long

exception HTTPUnsupportedMediaType (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

This indicates that the server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

code: 415, title: Unsupported Media Type

exception HTTPRequestRangeNotSatisfiable (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPClientError*

The server SHOULD return a response with this status code if a request included a Range request-header field, and none of the range-specifier values in this field overlap the current extent of the selected resource, and the request did not include an If-Range request-header field.

code: 416, title: Request Range Not Satisfiable

exception HTTPExpectationFailed (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of *HTTPClientError*

This indicates that the expectation given in an Expect request-header field could not be met by this server.

code: 417, title: Expectation Failed

exception HTTPUnprocessableEntity (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of *HTTPClientError*

This indicates that the server is unable to process the contained instructions.

May be used to notify the client that their JSON/XML is well formed, but not correct for the current request.

See RFC4918 section 11 for more information.

code: 422, title: Unprocessable Entity

exception HTTPLocked (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of *HTTPClientError*

This indicates that the resource is locked.

code: 423, title: Locked

exception HTTPFailedDependency (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)
subclass of *HTTPClientError*

This indicates that the method could not be performed because the requested action depended on another action and that action failed.

code: 424, title: Failed Dependency

exception HTTPInternalServerError (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

exception HTTPNotImplemented (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPServerError*

This indicates that the server does not support the functionality required to fulfill the request.

code: 501, title: Not Implemented

exception HTTPBadGateway (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPServerError*

This indicates that the server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

code: 502, title: Bad Gateway

exception HTTPServiceUnavailable (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPServerError*

This indicates that the server is currently unable to handle the request due to a temporary overloading or maintenance of the server.

code: 503, title: Service Unavailable

exception HTTPGatewayTimeout (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPServerError*

This indicates that the server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (e.g. HTTP, FTP, LDAP) or some other auxiliary server (e.g. DNS) it needed to access in attempting to complete the request.

code: 504, title: Gateway Timeout

exception HTTPVersionNotSupported (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPServerError*

This indicates that the server does not support, or refuses to support, the HTTP protocol version that was used in the request message.

code: 505, title: HTTP Version Not Supported

exception HTTPInsufficientStorage (*detail=None, headers=None, comment=None, body_template=None, json_formatter=None, **kw*)

subclass of *HTTPServerError*

This indicates that the server does not have enough space to save the resource.

code: 507, title: Insufficient Storage

pyramid.i18n

class TranslationString

The constructor for a *translation string*. A translation string is a Unicode-like object that has some extra metadata.

This constructor accepts one required argument named `msgid`. `msgid` must be the *message identifier* for the translation string. It must be a `unicode` object or a `str` object encoded in the default system encoding.

Optional keyword arguments to this object's constructor include `domain`, `default`, and `mapping`.

`domain` represents the *translation domain*. By default, the translation domain is `None`, indicating that this translation string is associated with the default translation domain (usually messages).

`default` represents an explicit *default text* for this translation string. Default text appears when the translation string cannot be translated. Usually, the `msgid` of a translation string serves double duty as its default text. However, using this option you can provide a different default text for this translation string. This feature is useful when the default of a translation string is too complicated or too long to be used as a message identifier. If `default` is provided, it must be a `unicode` object or a `str` object encoded in the default system encoding (usually means ASCII). If `default` is `None` (its default value), the `msgid` value used by this translation string will be assumed to be the value of `default`.

`mapping`, if supplied, must be a dictionary-like object which represents the replacement values for any *translation string replacement marker* instances found within the `msgid` (or `default`) value of this translation string.

`context` represents the *translation context*. By default, the translation context is `None`.

After a translation string is constructed, it behaves like most other `unicode` objects; its `msgid` value will be displayed when it is treated like a `unicode` object. Only when its `ugettext` method is called will it be translated.

Its default value is available as the `default` attribute of the object, its *translation domain* is available as the `domain` attribute, and the `mapping` is available as the `mapping` attribute. The object otherwise behaves much like a Unicode string.

TranslationStringFactory (*factory_domain*)

Create a factory which will generate translation strings without requiring that each call to the factory be passed a `domain` value. A single argument is passed to this class' constructor: `domain`. This value will be used as the domain values of `translationstring.TranslationString` objects generated by the `__call__` of this class. The `msgid`, `mapping`, and `default` values provided to the `__call__` method of an instance of this class have the meaning as described by the constructor of the `translationstring.TranslationString`

class `Localizer` (*locale_name*, *translations*)

An object providing translation and pluralizations related to the current request's locale name. A `pyramid.i18n.Localizer` object is created using the `pyramid.i18n.get_localizer()` function.

locale_name

The locale name for this localizer (e.g. `en` or `en_US`).

pluralize (*singular*, *plural*, *n*, *domain=None*, *mapping=None*)

Return a Unicode string translation by using two *message identifier* objects as a singular/plural pair and an *n* value representing the number that appears in the message using gettext plural forms support. The *singular* and *plural* objects should be unicode strings. There is no reason to use translation string objects as arguments as all metadata is ignored.

n represents the number of elements. *domain* is the translation domain to use to do the pluralization, and *mapping* is the interpolation mapping that should be used on the result. If the *domain* is not supplied, a default domain is used (usually `messages`).

Example:

```
num = 1
translated = localizer.pluralize('Add ${num} item',
                                'Add ${num} items',
                                num,
                                mapping={'num':num})
```

If using the gettext plural support, which is required for languages that have pluralisation rules other than `n != 1`, the *singular* argument must be the `message_id` defined in the translation file. The *plural* argument is not used in this case.

Example:

```
num = 1
translated = localizer.pluralize('item_plural',
                                '',
                                num,
                                mapping={'num':num})
```

translate (*tstring*, *domain=None*, *mapping=None*)

Translate a *translation string* to the current language and interpolate any *replacement markers* in the result. The `translate` method accepts three arguments: *tstring* (required), *domain* (optional) and *mapping* (optional). When called, it will translate the *tstring* translation string to a unicode object using the current locale. If the current locale could

not be determined, the result of interpolation of the default value is returned. The optional `domain` argument can be used to specify or override the domain of the `tstring` (useful when `tstring` is a normal string rather than a translation string). The optional `mapping` argument can specify or override the `tstring` interpolation mapping, useful when the `tstring` argument is a simple string instead of a translation string.

Example:

```
from pyramid.i18n import TranslationString
ts = TranslationString('Add ${item}', domain='mypackage',
                       mapping={'item': 'Item'})
translated = localizer.translate(ts)
```

Example:

```
translated = localizer.translate('Add ${item}', domain='mypackage',
                                mapping={'item': 'Item'})
```

get_localizer (*request*)

Deprecated since version 1.5: Use the `pyramid.request.Request.localizer` attribute directly instead. Retrieve a `pyramid.i18n.Localizer` object corresponding to the current request's locale name.

negotiate_locale_name (*request*)

Negotiate and return the *locale name* associated with the current request.

get_locale_name (*request*)

Deprecated since version 1.5: Use `pyramid.request.Request.locale_name` directly instead. Return the *locale name* associated with the current request.

default_locale_negotiator (*request*)

The default *locale negotiator*. Returns a locale name or `None`.

- First, the negotiator looks for the `__LOCALE__` attribute of the request object (possibly set by a view or a listener for an *event*). If the attribute exists and it is not `None`, its value will be used.
- Then it looks for the `request.params['__LOCALE__']` value.
- Then it looks for the `request.cookies['__LOCALE__']` value.

- Finally, the negotiator returns `None` if the locale could not be determined via any of the previous checks (when a locale negotiator returns `None`, it signifies that the *default locale name* should be used.)

make_localizer(*current_locale_name*, *translation_directories*)

Create a `pyramid.i18n.Localizer` object corresponding to the provided locale name from the translations found in the list of translation directories.

See *Internationalization and Localization* for more information about using Pyramid internationalization and localization services within an application.

pyramid.interfaces

Event-Related Interfaces

interface IApplicationCreated

Event issued when the `pyramid.config.Configurator.make_wsgi_app()` method is called. See the documentation attached to `pyramid.events.ApplicationCreated` for more information.



For backwards compatibility with Pyramid versions before 1.0, this interface can also be imported as `pyramid.interfaces.IWSGIApplicationCreatedEvent`.

app

Created application

interface INewRequest

An event type that is emitted whenever Pyramid begins to process a new request. See the documentation attached to `pyramid.events.NewRequest` for more information.

request

The request object

interface IContextFound

An event type that is emitted after Pyramid finds a *context* object but before it calls any view code. See the documentation attached to `pyramid.events.ContextFound` for more information.



For backwards compatibility with versions of Pyramid before 1.0, this event interface can also be imported as `pyramid.interfaces.IAfterTraversal`.

request

The request object

interface IBeforeTraversal

An event type that is emitted after Pyramid attempted to find a route but before it calls any traversal or view code. See the documentation attached to `pyramid.events.RouteFound` for more information.

request

The request object

interface INewResponse

An event type that is emitted whenever any Pyramid view returns a response. See the documentation attached to `pyramid.events.NewResponse` for more information.

request

The request object

response

The response object

interface IBeforeRender

Extends: `pyramid.interfaces.IDict`

Subscribers to this event may introspect and modify the set of *renderer globals* before they are passed to a *renderer*. The event object itself provides a dictionary-like interface for adding and removing *renderer globals*. The keys and values of the dictionary are those globals. For example:

```
from repoze.events import subscriber
from pyramid.interfaces import IBeforeRender

@subscriber(IBeforeRender)
def add_global(event):
    event['mykey'] = 'foo'
```

See also:

See also *Using the Before Render Event*.

rendering_val

The value returned by a view or passed to a `render` method for this rendering. This feature is new in Pyramid 1.2.

Other Interfaces**interface `IAuthenticationPolicy`**

An object representing a Pyramid authentication policy.

`unauthenticated_userid` (*request*)

Return the *unauthenticated* userid. This method performs the same duty as `authenticated_userid` but is permitted to return the userid based only on data present in the request; it needn't (and shouldn't) check any persistent store to ensure that the user record related to the request userid exists.

This method is intended primarily a helper to assist the `authenticated_userid` method in pulling credentials out of the request data, abstracting away the specific headers, query strings, etc that are used to authenticate the request.

`forget` (*request*)

Return a set of headers suitable for 'forgetting' the current user on subsequent requests.

`effective_principals` (*request*)

Return a sequence representing the effective principals typically including the *userid* and any groups belonged to by the current user, always including 'system' groups such as `pyramid.security.Everyone` and `pyramid.security.Authenticated`.

remember (*request*, *userid*, ***kw*)

Return a set of headers suitable for ‘remembering’ the *userid* named *userid* when set in a response. An individual authentication policy and its consumers can decide on the composition and meaning of ***kw*.

authenticated_userid (*request*)

Return the authenticated *userid* or *None* if no authenticated userid can be found. This method of the policy should ensure that a record exists in whatever persistent store is used related to the user (the user should not have been deleted); if a record associated with the current id does not exist in a persistent store, it should return *None*.

interface IAuthorizationPolicy

An object representing a Pyramid authorization policy.

principals_allowed_by_permission (*context*, *permission*)

Return a set of principal identifiers allowed by the permission in context. This behavior is optional; if you choose to not implement it you should define this method as something which raises a *NotImplementedError*. This method will only be called when the `pyramid.security.principals_allowed_by_permission` API is used.

permits (*context*, *principals*, *permission*)

Return *True* if any of the principals is allowed the permission in the current context, else return *False*

interface IExceptionResponse

Extends: `pyramid.interfaces.IException`, `pyramid.interfaces.IResponse`

An interface representing a WSGI response which is also an exception object. Register an exception view using this interface as a *context* to apply the registered view for all exception types raised by Pyramid internally (any exception that inherits from `pyramid.response.Response`, including `pyramid.httpexceptions.HTTPNotFound` and `pyramid.httpexceptions.HTTPForbidden`).

prepare (*environ*)

Prepares the response for being called as a WSGI application

interface IRoute

Interface representing the type of object returned from `IRoutesMapper.get_route`

pattern

The route pattern

pregenerator

This attribute should either be `None` or a callable object implementing the `IRoutePregenerator` interface

match (*path*)

If the *path* passed to this function can be matched by the *pattern* of this route, return a dictionary (the 'matchdict'), which will contain keys representing the dynamic segment markers in the pattern mapped to values extracted from the provided *path*.

If the *path* passed to this function cannot be matched by the *pattern* of this route, return `None`.

name

The route name

generate (*kw*)

Generate a URL based on filling in the dynamic segment markers in the pattern using the *kw* dictionary provided.

factory

The *root factory* used by the Pyramid router when this route matches (or `None`)

predicates

A sequence of *route predicate* objects used to determine if a request matches this route or not after basic pattern matching has been completed.

interface IRoutePregenerator**__call__** (*request, elements, kw*)

A pregenerator is a function associated by a developer with a *route*. The pregenerator for a route is called by `pyramid.request.Request.route_url()` in order to adjust the set of arguments passed to it by the user for special purposes, such as Pylons 'subdomain' support. It will influence the URL returned by `route_url`.

A pregenerator should return a two-tuple of (*elements*, *kw*) after examining the originals passed to this function, which are the arguments (*request*, *elements*, *kw*). The simplest pregenerator is:

```
def pregenerator(request, elements, kw):  
    return elements, kw
```

You can employ a pregenerator by passing a `pregenerator` argument to the `pyramid.config.Configurator.add_route()` function.

interface `ISession`

Extends: `pyramid.interfaces.IDict`

An interface representing a session (a web session object, usually accessed via `request.session`).

Keys and values of a session must be pickleable.

`new`

Boolean attribute. If `True`, the session is new.

`changed()`

Mark the session as changed. A user of a session should call this method after he or she mutates a mutable object that is *a value of the session* (it should not be required after mutating the session itself). For example, if the user has stored a dictionary in the session under the key `foo`, and he or she does `session['foo'] = {}`, `changed()` needn't be called. However, if subsequently he or she does `session['foo']['a'] = 1`, `changed()` must be called for the sessioning machinery to notice the mutation of the internal dictionary.

`created`

Integer representing Epoch time when created.

`new_csrf_token()`

Create and set into the session a new, random cross-site request forgery protection token. Return the token. It will be a string.

`invalidate()`

Invalidate the session. The action caused by `invalidate` is implementation-dependent, but it should have the effect of completely dissociating any data stored in the session with the current request. It might set response values (such as one which clears a cookie), or it might not.

An invalidated session may be used after the call to `invalidate` with the effect that a new session is created to store the data. This enables workflows requiring an entirely new session, such as in the case of changing privilege levels or preventing fixation attacks.

flash (*msg*, *queue*='', *allow_duplicate*=True)

Push a flash message onto the end of the flash queue represented by *queue*. An alternate flash message queue can be used by passing an optional *queue*, which must be a string. If *allow_duplicate* is false, if the *msg* already exists in the queue, it will not be re-added.

pop_flash (*queue*='')

Pop a queue from the flash storage. The queue is removed from flash storage after this message is called. The queue is returned; it is a list of flash messages added by *pyramid.interfaces.ISession.flash()*

get_csrf_token ()

Return a random cross-site request forgery protection token. It will be a string. If a token was previously added to the session via *new_csrf_token*, that token will be returned. If no CSRF token was previously set into the session, *new_csrf_token* will be called, which will create and set a token, and this token will be returned.

peek_flash (*queue*='')

Peek at a queue in the flash storage. The queue remains in flash storage after this message is called. The queue is returned; it is a list of flash messages added by *pyramid.interfaces.ISession.flash()*

interface ISessionFactory

An interface representing a factory which accepts a request object and returns an *ISession* object

__call__ (*request*)

Return an *ISession* object

interface IRendererInfo

An object implementing this interface is passed to every *renderer factory* constructor as its only argument (conventionally named *info*)

type

The renderer type name

clone ()

Return a shallow copy that does not share any mutable state.

name

The value passed by the user as the renderer name

settings

The deployment settings dictionary related to the current application

registry

The “current” application registry when the renderer was created

package

The “current package” when the renderer configuration statement was found

interface IRendererFactory**__call__** (*info*)

Return an object that implements *pyramid.interfaces.IRenderer*. *info* is an object that implements *pyramid.interfaces.IRendererInfo*.

interface IRenderer**__call__** (*value, system*)

Call the renderer with the result of the view (*value*) passed in and return a result (a string or unicode object useful as a response body). Values computed by the system are passed by the system in the *system* parameter, which is a dictionary. Keys in the dictionary include: *view* (the view callable that returned the *value*), *renderer_name* (the template name or simple name of the renderer), *context* (the context object passed to the view), and *request* (the request object passed to the view).

interface IRequestFactory

A utility which generates a request

blank (*path*)

Return an empty request object (see *pyramid.request.Request.blank()*)

__call__ (*environ*)

Return an instance of *pyramid.request.Request*

interface IResponseFactory

A utility which generates a response

__call__ (*request*)

Return a response object implementing *IResponse*, e.g. *pyramid.response.Response*). It should handle the case when *request* is *None*.

interface `IViewMapperFactory`

`__call__` (*self*, ***kw*)

Return an object which implements *pyramid.interfaces.IViewMapper*. *kw* will be a dictionary containing view-specific arguments, such as permission, predicates, attr, renderer, and other items. An *IViewMapperFactory* is used by *pyramid.config.Configurator.add_view()* to provide a plugpoint to extension developers who want to modify potential view callable invocation signatures and response values.

interface `IViewMapper`

`__call__` (*self*, *object*)

Provided with an arbitrary object (a function, class, or instance), returns a callable with the call signature (*context*, *request*). The callable returned should itself return a *Response* object. An *IViewMapper* is returned by *pyramid.interfaces.IViewMapperFactory*.

interface `IDict`

`update` (*d*)

Update the renderer dictionary with another dictionary *d*.

`clear` ()

Clear all values from the dictionary

`__delitem__` (*k*)

Delete an item from the dictionary which is passed to the renderer as the renderer globals dictionary.

`keys` ()

Return a list of keys from the dictionary

`__iter__` ()

Return an iterator over the keys of this dictionary

`__setitem__` (*k*, *value*)

Set a key/value pair into the dictionary

`__contains__` (*k*)

Return *True* if key *k* exists in the dictionary.

get (*k*, *default=None*)

Return the value for key *k* from the renderer dictionary, or the default if no such value exists.

popitem ()

Pop the item with key *k* from the dictionary and return it as a two-tuple (*k*, *v*). If *k* doesn't exist, raise a `KeyError`.

pop (*k*, *default=None*)

Pop the key *k* from the dictionary and return its value. If *k* doesn't exist, and *default* is provided, return the default. If *k* doesn't exist and *default* is not provided, raise a `KeyError`.

__getitem__ (*k*)

Return the value for key *k* from the dictionary or raise a `KeyError` if the key doesn't exist

values ()

Return a list of values from the dictionary

items ()

Return a list of [(*k*,*v*)] pairs from the dictionary

setdefault (*k*, *default=None*)

Return the existing value for key *k* in the dictionary. If no value with *k* exists in the dictionary, set the *default* value into the dictionary under the *k* name passed. If a value already existed in the dictionary, return it. If a value did not exist in the dictionary, return the default

interface IMultiDict

Extends: *pyramid.interfaces.IDict*

An ordered dictionary that can have multiple values for each key. A multidict adds the methods `getall`, `getone`, `mixed`, `extend`, `add`, and `dict_of_lists` to the normal dictionary interface. A multidict data structure is used as `request.POST`, `request.GET`, and `request.params` within an Pyramid application.

mixed ()

Returns a dictionary where the values are either single values, or a list of values when a key/value appears more than once in this dictionary. This is similar to the kind of dictionary often used to represent the variables in a web request.

dict_of_lists ()

Returns a dictionary where each key is associated with a list of values.

getone (*key*)

Get one value matching the key, raising a `KeyError` if multiple values were found.

getall (*key*)

Return a list of all values matching the key (may be an empty list)

add (*key*, *value*)

Add the key and value, not overwriting any previous value.

extend (*other=None*, ***kwargs*)

Add a set of keys and values, not overwriting any previous values. The `other` structure may be a list of two-tuples or a dictionary. If `**kwargs` is passed, its value *will* overwrite existing values.

interface IResponse

Represents a WSGI response using the WebOb response interface. Some attribute and method documentation of this interface references **RFC 2616**.

This interface is most famously implemented by `pyramid.response.Response` and the HTTP exception classes in `pyramid.httpexceptions`.

body_file

A file-like object that can be used to write to the body. If you passed in a list `app_iter`, that `app_iter` will be modified by writes.

location

Gets and sets and deletes the Location header. For more information on Location see RFC 2616 section 14.30.

environ

Get/set the request environ associated with this response, if any.

content_length

Gets and sets and deletes the Content-Length header. For more information on Content-Length see RFC 2616 section 14.17. Converts using `int`.

merge_cookies (*resp*)

Merge the cookies that were set on this response with the given `resp` object (which can be any WSGI application). If the `resp` is a `webob.Response` object, then the other object will be modified in-place.

unicode_body

Get/set the unicode value of the body (using the charset of the Content-Type)

copy()

Makes a copy of the response and returns the copy.

accept_ranges

Gets and sets and deletes the Accept-Ranges header. For more information on Accept-Ranges see RFC 2616, section 14.5

status

The status string.

charset

Get/set the charset (in the Content-Type)

content_range

Gets and sets and deletes the Content-Range header. For more information on Content-Range see section 14.16. Converts using ContentRange object.

cache_expires

Get/set the Cache-Control and Expires headers. This sets the response to expire in the number of seconds passed when set.

delete_cookie (*key, path='/', domain=None*)

Delete a cookie from the client. Note that path and domain must match how the cookie was originally set. This sets the cookie to the empty string, and max_age=0 so that it should expire immediately.

vary

Gets and sets and deletes the Vary header. For more information on Vary see section 14.44. Converts using list.

content_location

Gets and sets and deletes the Content-Location header. For more information on Content-Location see RFC 2616 section 14.14.

conditional_response_app (*environ, start_response*)

Like the normal `__call__` interface, but checks conditional headers:

- If-Modified-Since (304 Not Modified; only on GET, HEAD)
- If-None-Match (304 Not Modified; only on GET, HEAD)
- Range (406 Partial Content; only on GET, HEAD)

app_iter_range (*start, stop*)

Return a new app_iter built from the response app_iter that serves up only the given start:stop range.

expires

Gets and sets and deletes the Expires header. For more information on Expires see RFC 2616 section 14.21. Converts using HTTP date.

www_authenticate

Gets and sets and deletes the WWW-Authenticate header. For more information on WWW-Authenticate see RFC 2616 section 14.47. Converts using 'parse_auth' and 'serialize_auth'.

last_modified

Gets and sets and deletes the Last-Modified header. For more information on Last-Modified see RFC 2616 section 14.29. Converts using HTTP date.

encode_content (*encoding='gzip', lazy=False*)

Encode the content with the given encoding (only gzip and identity are supported).

retry_after

Gets and sets and deletes the Retry-After header. For more information on Retry-After see RFC 2616 section 14.37. Converts using HTTP date or delta seconds.

content_disposition

Gets and sets and deletes the Content-Disposition header. For more information on Content-Disposition see RFC 2616 section 19.5.1.

__call__ (*environ, start_response*)

WSGI call interface, should call the start_response callback and should return an iterable

unset_cookie (*key, strict=True*)

Unset a cookie with the given name (remove it from the response).

headers

The headers in a dictionary-like object

content_md5

Gets and sets and deletes the Content-MD5 header. For more information on Content-MD5 see RFC 2616 section 14.14.

app_iter

Returns the app_iter of the response.

If body was set, this will create an app_iter from that body (a single-item list)

content_encoding

Gets and sets and deletes the Content-Encoding header. For more information about Content-Encoding see RFC 2616 section 14.11.

etag

Gets and sets and deletes the ETag header. For more information on ETag see RFC 2616 section 14.19. Converts using Entity tag.

age

Gets and sets and deletes the Age header. Converts using int. For more information on Age see RFC 2616, section 14.6.

allow

Gets and sets and deletes the Allow header. Converts using list. For more information on Allow see RFC 2616, Section 14.7.

pragma

Gets and sets and deletes the Pragma header. For more information on Pragma see RFC 2616 section 14.32.

server

Gets and sets and deletes the Server header. For more information on Server see RFC216 section 14.38.

set_cookie (*key*, *value*='', *max_age*=None, *path*='/', *domain*=None, *secure*=False, *httponly*=False, *comment*=None, *expires*=None, *overwrite*=False)

Set (add) a cookie for the response

content_type

Get/set the Content-Type header (or None), without the charset or any parameters. If you include parameters (or ; at all) when setting the content_type, any existing parameters will be deleted; otherwise they will be preserved.

body

The body of the response, as a str. This will read in the entire app_iter if necessary.

content_language

Gets and sets and deletes the Content-Language header. Converts using list. For more information about Content-Language see RFC 2616 section 14.12.

request

Return the request associated with this response if any.

cache_control

Get/set/modify the Cache-Control header (RFC 2616 section 14.9)

content_type_params

A dictionary of all the parameters in the content type. This is not a view, set to change, modifications of the dict would not be applied otherwise.

RequestClass

Alias for *pyramid.request.Request*

status_int

The status as an integer

headerlist

The list of response headers.

md5_etag (*body=None, set_content_md5=False*)

Generate an etag for the response object using an MD5 hash of the body (the body parameter, or *self.body* if not given). Sets *self.etag*. If *set_content_md5* is True sets *self.content_md5* as well

date

Gets and sets and deletes the Date header. For more information on Date see RFC 2616 section 14.18. Converts using HTTP date.

interface IIntrospectable

An introspectable object used for configuration introspection. In addition to the methods below, objects which implement this interface must also implement all the methods of Python's *collections.MutableMapping* (the “dictionary interface”), and must be hashable.

discriminator

introspectable discriminator (within category) (must be hashable)

unrelate (*category_name, discriminator*)

Indicate an intent to break the relationship between this *IIntrospectable* with another *IIntrospectable* (the one associated with the *category_name* and *discriminator*) during action execution.

action_info

An *IActionInfo* object representing the caller that invoked the creation of this introspectable (usually a sentinel until updated during *self.register*)

title

Text title describing this introspectable

category_name

introspection category name

relate (*category_name*, *discriminator*)

Indicate an intent to relate this `IIntrospectable` with another `IIntrospectable` (the one associated with the `category_name` and `discriminator`) during action execution.

type_name

Text type name describing this introspectable

discriminator_hash

an integer hash of the discriminator

register (*introspector*, *action_info*)

Register this `IIntrospectable` with an introspector. This method is invoked during action execution. Adds the introspectable and its relations to the introspector. `introspector` should be an object implementing `IIntrospector`. `action_info` should be a object implementing the interface `pyramid.interfaces.IActionInfo` representing the call that registered this introspectable. Pseudocode for an implementation of this method:

```
def register(self, introspector, action_info):
    self.action_info = action_info
    introspector.add(self)
    for methodname, category_name, discriminator in self._
    relations:
        method = getattr(introspector, methodname)
        method((i.category_name, i.discriminator),
               (category_name, discriminator))
```

order

integer order in which registered with introspector (managed by introspector, usually)

__hash__ ()

Introspectables must be hashable. The typical implementation of an introspectable's `__hash__` is:

```
return hash((self.category_name,) + (self.discriminator,))
```

interface IIntrospector

relate (*pairs)

Given any number of (category_name, discriminator) pairs passed as positional arguments, relate the associated introspectables to each other. The introspectable related to each pair must have already been added via .add or .add_intr; a KeyError will result if this is not true. An error will not be raised if any pair has already been associated with another.

This method is not typically called directly, instead it's called indirectly by pyramid.interfaces.IIntrospector.register()

get_category (category_name, default=None, sort_key=None)

Get a sequence of dictionaries in the form [{"introspectable": IIntrospectable, "related": [sequence of related IIntrospectables]}, ...] where each introspectable is part of the category associated with category_name.

If the category named category_name does not exist in the introspector the value passed as default will be returned.

If sort_key is None, the sequence will be returned in the order the introspectables were added to the introspector. Otherwise, sort_key should be a function that accepts an IIntrospectable and returns a value from it (ala the key function of Python's sorted callable).

categories ()

Return a sorted sequence of category names known by this introspector

related (intr)

Return a sequence of IIntrospectables related to the IIntrospectable intr. Return the empty sequence if no relations for exist.

remove (category_name, discriminator)

Remove the IIntrospectable related to category_name and discriminator from the introspector, and fix up any relations that the introspectable participates in. This method will not raise an error if an introspectable related to the category name and discriminator does not exist.

unrelate (*pairs)

Given any number of (category_name, discriminator) pairs passed as positional arguments, unrelate the associated introspectables from each other. The introspectable related to each pair must have already been added via `.add` or `.add_intr`; a `KeyError` will result if this is not true. An error will not be raised if any pair is not already related to another.

This method is not typically called directly, instead it's called indirectly by `pyramid.interfaces.IIntrospector.register()`

categorized (sort_key=None)

Get a sequence of tuples in the form [(category_name, [{'introspectable': IIntrospectable, 'related': [sequence of related IIntrospectables]}, ...])] representing all known introspectables. If sort_key is None, each introspectables sequence will be returned in the order the introspectables were added to the introspector. Otherwise, sort_key should be a function that accepts an IIntrospectable and returns a value from it (ala the key function of Python's sorted callable).

add (intr)

Add the IIntrospectable intr (use instead of `pyramid.interfaces.IIntrospector.add()` when you have a custom IIntrospectable). Replaces any existing introspectable registered using the same category/discriminator.

This method is not typically called directly, instead it's called indirectly by `pyramid.interfaces.IIntrospector.register()`

get (category_name, discriminator, default=None)

Get the IIntrospectable related to the category_name and the discriminator (or discriminator hash) discriminator. If it does not exist in the introspector, return the value of default

interface IActionInfo

Class which provides code introspection capability associated with an action. The ParserInfo class used by ZCML implements the same interface.

file

Filename of action-invoking code as a string

__str__ ()

Return a representation of the action information (including source code from file, if possible)

line

Starting line number in file (as an integer) of action-invoking code. This will be `None` if the value could not be determined.

interface IAssetDescriptor

Describes an *asset*.

isdir()

Returns `True` if the asset is a directory, otherwise returns `False`.

abspath()

Returns an absolute path in the filesystem to the asset.

absspec()

Returns the absolute asset specification for this asset (e.g. `mypackage:templates/foo.pt`).

exists()

Returns `True` if asset exists, otherwise returns `False`.

stream()

Returns an input stream for reading asset contents. Raises an exception if the asset is a directory or does not exist.

listdir()

Returns iterable of filenames of directory contents. Raises an exception if asset is not a directory.

interface IResourceURL**virtual_path_tuple**

The virtual url path of the resource as a tuple. (New in 1.5)

physical_path

The physical url path of the resource as a string.

virtual_path

The virtual url path of the resource as a string.

physical_path_tuple

The physical url path of the resource as a tuple. (New in 1.5)

interface ICacheBuster

A cache buster modifies the URL generation machinery for `static_url()`. See *Cache Busting*.

New in version 1.6.

`__call__(request, subpath, kw)`

Modifies a subpath and/or keyword arguments from which a static asset URL will be computed during URL generation.

The `subpath` argument is a path of `/`-delimited segments that represent the portion of the asset URL which is used to find the asset. The `kw` argument is a dict of keywords that are to be passed eventually to `static_url()` for URL generation. The return value should be a two-tuple of (`subpath`, `kw`) where `subpath` is the relative URL from where the file is served and `kw` is the same input argument. The return value should be modified to include the cache bust token in the generated URL.

The `kw` dictionary contains extra arguments passed to `static_url()` as well as some extra items that may be useful including:

- `pathspec` is the path specification for the resource to be cache busted.
- `rawspec` is the original location of the file, ignoring any calls to `pyramid.config.Configurator.override_asset()`.

The `pathspec` and `rawspec` values are only different in cases where an asset has been mounted into a virtual location using `pyramid.config.Configurator.override_asset()`. For example, with a call to `request.static_url('myapp:static/foo.png')`, the `pathspec` is `myapp:static/foo.png` whereas the `rawspec` may be `themepkg:bar.png`, assuming a call to `config.override_asset('myapp:static/foo.png', 'themepkg:bar.png')`.

interface IViewDeriver

`__call__(view, info)`

Derive a new view from the supplied view.

View options, package information and registry are available on `info`, an instance of `pyramid.interfaces.IViewDeriverInfo`.

The `view` is a callable accepting (`context`, `request`).

options

A list of supported options to be passed to `pyramid.config.Configurator.add_view()`. This attribute is optional.

interface IViewDeriverInfo

An object implementing this interface is passed to every *view deriver* during configuration.

original_view

The original view object being wrapped

settings

The deployment settings dictionary related to the current application

predicates

The list of predicates active on the view

registry

The “current” application registry where the view was created

package

The “current package” where the view configuration statement was found

options

The view options passed to the view, including any default values that were not overridden

pyramid.location**lineage** (*resource*)

Return a generator representing the *lineage* of the *resource* object implied by the *resource* argument. The generator first returns *resource* unconditionally. Then, if *resource* supplies a `__parent__` attribute, return the resource represented by `resource.__parent__`. If *that* resource has a `__parent__` attribute, return that resource’s parent, and so on, until the resource being inspected either has no `__parent__` attribute or which has a `__parent__` attribute of `None`. For example, if the resource tree is:

```
thing1 = Thing()
thing2 = Thing()
thing2.__parent__ = thing1
```

Calling `lineage(thing2)` will return a generator. When we turn it into a list, we will get:

```
list(lineage(thing2))
[ <Thing object at thing2>, <Thing object at thing1> ]
```

inside (*resource1*, *resource2*)

Is *resource1* ‘inside’ *resource2*? Return True if so, else False.

resource1 is ‘inside’ *resource2* if *resource2* is a *lineage* ancestor of *resource1*. It is a lineage ancestor if its parent (or one of its parent’s parents, etc.) is an ancestor.

pyramid.paster

bootstrap (*config_uri*, *request=None*, *options=None*)

Load a WSGI application from the PasteDeploy config file specified by *config_uri*. The environment will be configured as if it is currently serving *request*, leaving a natural environment in place to write scripts that can generate URLs and utilize renderers.

This function returns a dictionary with *app*, *root*, *closer*, *request*, and *registry* keys. *app* is the WSGI app loaded (based on the *config_uri*), *root* is the traversal root resource of the Pyramid application, and *closer* is a parameterless callback that may be called when your script is complete (it pops a threadlocal stack).



Most operations within Pyramid expect to be invoked within the context of a WSGI request, thus it’s important when loading your application to anchor it when executing scripts and other code that is not normally invoked during active WSGI requests.



For a complex config file containing multiple Pyramid applications, this function will setup the environment under the context of the last-loaded Pyramid application. You may load a specific application yourself by using the lower-level functions *pyramid.paster.get_app()* and *pyramid.scripting.prepare()* in conjunction with *pyramid.config.global_registries*.

config_uri – specifies the PasteDeploy config file to use for the interactive shell. The format is *inifile#name*. If the name is left off, *main* will be assumed.

request – specified to anchor the script to a given set of WSGI parameters. For example, most people would want to specify the host, scheme and port such that their script will generate URLs

in relation to those parameters. A request with default parameters is constructed for you if none is provided. You can mutate the request's `environ` later to setup a specific host/port/scheme/etc.

`options` Is passed to `get_app` for use as variable assignments like `{ 'http_port': 8080 }` and then use `%(http_port)s` in the config file.

See *Writing a Script* for more information about how to use this function.

get_app (*config_uri*, *name=None*, *options=None*)

Return the WSGI application named `name` in the PasteDeploy config file specified by `config_uri`.

`options`, if passed, should be a dictionary used as variable assignments like `{ 'http_port': 8080 }`. This is useful if e.g. `%(http_port)s` is used in the config file.

If the `name` is `None`, this will attempt to parse the name from the `config_uri` string expecting the format `inifile#name`. If no name is found, the name will default to “main”.

get_appsettings (*config_uri*, *name=None*, *options=None*)

Return a dictionary representing the key/value pairs in an `app` section within the file represented by `config_uri`.

`options`, if passed, should be a dictionary used as variable assignments like `{ 'http_port': 8080 }`. This is useful if e.g. `%(http_port)s` is used in the config file.

If the `name` is `None`, this will attempt to parse the name from the `config_uri` string expecting the format `inifile#name`. If no name is found, the name will default to “main”.

setup_logging (*config_uri*, *global_conf=None*)

Set up logging via `logging.config.fileConfig()` with the filename specified via `config_uri` (a string in the form `filename#sectionname`).

`ConfigParser` defaults are specified for the special `__file__` and `here` variables, similar to PasteDeploy config loading. Extra defaults can optionally be specified as a dict in `global_conf`.

pyramid.path**CALLER_PACKAGE**

A constant used by the constructor of `pyramid.path.DottedNameResolver` and `pyramid.path.AssetResolver`.

class DottedNameResolver (*package=pyramid.path.CALLER_PACKAGE*)

A class used to resolve a *dotted Python name* to a package or module object.

New in version 1.3.

The constructor accepts a single argument named `package` which may be any of:

- A fully qualified (not relative) dotted name to a module or package
- a Python module or package object
- The value `None`
- The constant value `pyramid.path.CALLER_PACKAGE`.

The default value is `pyramid.path.CALLER_PACKAGE`.

The `package` is used when a relative dotted name is supplied to the `resolve()` method. A dotted name which has a `.` (dot) or `:` (colon) as its first character is treated as relative.

If `package` is `None`, the resolver will only be able to resolve fully qualified (not relative) names. Any attempt to resolve a relative name will result in an `ValueError` exception.

If `package` is `pyramid.path.CALLER_PACKAGE`, the resolver will treat relative dotted names as relative to the caller of the `resolve()` method.

If `package` is a *module* or *module name* (as opposed to a package or package name), its containing package is computed and this package used to derive the package name (all names are resolved relative to packages, never to modules). For example, if the `package` argument to this type was passed the string `xml.dom.expatbuilder`, and `.minidom` is supplied to the `resolve()` method, the resulting import would be for `xml.minidom`, because `xml.dom.expatbuilder` is a module object, not a package object.

If `package` is a *package* or *package name* (as opposed to a module or module name), this package will be used to relative compute dotted names. For example, if the `package` argument to this type was passed the string `xml.dom`, and `.minidom` is supplied to the `resolve()` method, the resulting import would be for `xml.minidom`.

maybe_resolve (*dotted*)

This method behaves just like `resolve()`, except if the dotted value passed is not a string, it is simply returned. For example:

```
import xml
r = DottedNameResolver()
v = r.maybe_resolve(xml)
# v is the xml module; no exception raised
```

resolve (*dotted*)

This method resolves a dotted name reference to a global Python object (an object which can be imported) to the object itself.

Two dotted name styles are supported:

- `pkg_resources`-style dotted names where non-module attributes of a package are separated from the rest of the path using a `:` e.g. `package.module:attr`.
- `zope.dottedname`-style dotted names where non-module attributes of a package are separated from the rest of the path using a `.` e.g. `package.module.attr`.

These styles can be used interchangeably. If the supplied name contains a `:` (colon), the `pkg_resources` resolution mechanism will be chosen, otherwise the `zope.dottedname` resolution mechanism will be chosen.

If the `dotted` argument passed to this method is not a string, a `ValueError` will be raised.

When a dotted name cannot be resolved, a `ValueError` error is raised.

Example:

```
r = DottedNameResolver()
v = r.resolve('xml') # v is the xml module
```

class AssetResolver (*package=pyramid.path.CALLER_PACKAGE*)

A class used to resolve an *asset specification* to an *asset descriptor*.

New in version 1.3.

The constructor accepts a single argument named `package` which may be any of:

- A fully qualified (not relative) dotted name to a module or package
- a Python module or package object
- The value `None`

- The constant value `pyramid.path.CALLER_PACKAGE`.

The default value is `pyramid.path.CALLER_PACKAGE`.

The `package` is used when a relative asset specification is supplied to the `resolve()` method. An asset specification without a colon in it is treated as relative.

If `package` is `None`, the resolver will only be able to resolve fully qualified (not relative) asset specifications. Any attempt to resolve a relative asset specification will result in a `ValueError` exception.

If `package` is `pyramid.path.CALLER_PACKAGE`, the resolver will treat relative asset specifications as relative to the caller of the `resolve()` method.

If `package` is a *module* or *module name* (as opposed to a package or package name), its containing package is computed and this package is used to derive the package name (all names are resolved relative to packages, never to modules). For example, if the `package` argument to this type was passed the string `xml.dom.expatbuilder`, and `template.pt` is supplied to the `resolve()` method, the resulting absolute asset spec would be `xml.minidom:template.pt`, because `xml.dom.expatbuilder` is a module object, not a package object.

If `package` is a *package* or *package name* (as opposed to a module or module name), this package will be used to compute relative asset specifications. For example, if the `package` argument to this type was passed the string `xml.dom`, and `template.pt` is supplied to the `resolve()` method, the resulting absolute asset spec would be `xml.minidom:template.pt`.

resolve(*spec*)

Resolve the asset spec named as `spec` to an object that has the attributes and methods described in `pyramid.interfaces.IAssetDescriptor`.

If `spec` is an absolute filename (e.g. `/path/to/myproject/templates/foo.pt`) or an absolute asset spec (e.g. `myproject:templates.foo.pt`), an asset descriptor is returned without taking into account the `package` passed to this class' constructor.

If `spec` is a *relative* asset specification (an asset specification without a `:` in it, e.g. `templates/foo.pt`), the `package` argument of the constructor is used as the package portion of the asset spec. For example:

```
a = AssetResolver('myproject')
resolver = a.resolve('templates/foo.pt')
print(resolver.abstractmethod())
# -> /path/to/myproject/templates/foo.pt
```

If the `AssetResolver` is constructed without a `package` argument of `None`, and a relative asset specification is passed to `resolve`, a `ValueError` exception is raised.

pyramid.registry**class Registry** (*arg, **kw)

A registry object is an *application registry*. It is used by the framework itself to perform mappings of URLs to view callables, as well as servicing other various framework duties. A registry has its own internal API, but this API is rarely used by Pyramid application developers (it's usually only used by developers of the Pyramid framework). But it has a number of attributes that may be useful to application developers within application code, such as `settings`, which is a dictionary containing application deployment settings.

For information about the purpose and usage of the application registry, see *Using the Zope Component Architecture in Pyramid*.

The application registry is usually accessed as `request.registry` in application code.

settings

The dictionary-like *deployment settings* object. See *Deployment Settings* for information. This object is often accessed as `request.registry.settings` or `config.registry.settings` in a typical Pyramid application.

package_name

New in version 1.6.

When a registry is set up (or created) by a *Configurator*, this attribute will be the shortcut for `pyramid.config.Configurator.package_name`.

This attribute is often accessed as `request.registry.package_name` or `config.registry.package_name` or `config.package_name` in a typical Pyramid application.

introspector

New in version 1.3.

When a registry is set up (or created) by a *Configurator*, the registry will be decorated with an instance named `introspector` implementing the *pyramid.interfaces.IIntrospector* interface.

See also:

See also `pyramid.config.Configurator.introspector`.

When a registry is created “by hand”, however, this attribute will not exist until set up by a configurator.

This attribute is often accessed as `request.registry.introspector` in a typical Pyramid application.

notify (*events)

Fire one or more events. All event subscribers to the event(s) will be notified. The subscribers will be called synchronously. This method is often accessed as `request.registry.notify` in Pyramid applications to fire custom events. See *Creating Your Own Events* for more information.

class Introspectable

New in version 1.3.

The default implementation of the interface `pyramid.interfaces.IIntrospectable` used by framework exenders. An instance of this class is created when `pyramid.config.Configurator.introspectable` is called.

class Deferred (func)

Can be used by a third-party configuration extender to wrap a *discriminator* during configuration if an immediately hashable discriminator cannot be computed because it relies on unresolved values. The function should accept no arguments and should return a hashable discriminator.

New in version 1.4.

undefers (v)

Function which accepts an object and returns it unless it is a `pyramid.registry.Deferred` instance. If it is an instance of that class, its `resolve` method is called, and the result of the method is returned.

New in version 1.4.

class predvalseq

A subtype of tuple used to represent a sequence of predicate values

New in version 1.4.

pyramid.renderers**get_renderer** (renderer_name, package=None)

Return the renderer object for the renderer `renderer_name`.

You may supply a relative asset spec as `renderer_name`. If the `package` argument is supplied, a relative renderer name will be converted to an absolute asset specification by combining the package `package` with the relative asset specification `renderer_name`. If `package` is `None` (the default), the package name of the *caller* of this function will be used as the package.

render (*renderer_name*, *value*, *request=None*, *package=None*)

Using the renderer *renderer_name* (a template or a static renderer), render the value (or set of values) present in *value*. Return the result of the renderer's `__call__` method (usually a string or Unicode).

If the *renderer_name* refers to a file on disk, such as when the renderer is a template, it's usually best to supply the name as an *asset specification* (e.g. `package:package/path/to/template.pt`).

You may supply a relative asset spec as *renderer_name*. If the *package* argument is supplied, a relative renderer path will be converted to an absolute asset specification by combining the package *package* with the relative asset specification *renderer_name*. If *package* is `None` (the default), the package name of the *caller* of this function will be used as the package.

The *value* provided will be supplied as the input to the renderer. Usually, for template renderings, this should be a dictionary. For other renderers, this will need to be whatever sort of value the renderer expects.

The 'system' values supplied to the renderer will include a basic set of top-level system names, such as `request`, `context`, `renderer_name`, and `view`. See *System Values Used During Rendering* for the full list. If *renderer globals* have been specified, these will also be used to augment the value.

Supply a *request* parameter in order to provide the renderer with the most correct 'system' values (*request* and *context* in particular).

render_to_response (*renderer_name*, *value*, *request=None*, *package=None*, *response=None*)

Using the renderer *renderer_name* (a template or a static renderer), render the value (or set of values) using the result of the renderer's `__call__` method (usually a string or Unicode) as the response body.

If the renderer name refers to a file on disk (such as when the renderer is a template), it's usually best to supply the name as a *asset specification*.

You may supply a relative asset spec as *renderer_name*. If the *package* argument is supplied, a relative renderer name will be converted to an absolute asset specification by combining the package *package* with the relative asset specification *renderer_name*. If you do not supply a package (or *package* is `None`) the package name of the *caller* of this function will be used as the package.

The *value* provided will be supplied as the input to the renderer. Usually, for template renderings, this should be a dictionary. For other renderers, this will need to be whatever sort of value the renderer expects.

The ‘system’ values supplied to the renderer will include a basic set of top-level system names, such as `request`, `context`, `renderer_name`, and `view`. See *System Values Used During Rendering* for the full list. If *renderer globals* have been specified, these will also be used to argument the value.

Supply a `request` parameter in order to provide the renderer with the most correct ‘system’ values (`request` and `context` in particular). Keep in mind that any changes made to `request.response` prior to calling this function will not be reflected in the resulting response object. A new response object will be created for each call unless one is passed as the `response` argument.

Changed in version 1.6: In previous versions, any changes made to `request.response` outside of this function call would affect the returned response. This is no longer the case. If you wish to send in a pre-initialized response then you may pass one in the `response` argument.

class `JSON` (*serializer=<function dumps>*, *adapters=()*, ***kw*)

Renderer that returns a JSON-encoded string.

Configure a custom JSON renderer using the `add_renderer()` API at application startup time:

```
from pyramid.config import Configurator

config = Configurator()
config.add_renderer('myjson', JSON(indent=4))
```

Once this renderer is registered as above, you can use `myjson` as the `renderer=` parameter to `@view_config` or `add_view()`:

```
from pyramid.view import view_config

@view_config(renderer='myjson')
def myview(request):
    return {'greeting': 'Hello world'}
```

Custom objects can be serialized using the renderer by either implementing the `__json__` magic method, or by registering adapters with the renderer. See *Serializing Custom Objects* for more information.



The default serializer uses `json.JSONEncoder`. A different serializer can be specified via the `serializer` argument. Custom serializers should accept the object, a callback default, and any extra `kw` keyword arguments passed during renderer construction. This feature isn’t widely used but it can be used to replace the stock JSON serializer with, say, `simplejson`. If all you want

to do, however, is serialize custom objects, you should use the method explained in *Serializing Custom Objects* instead of replacing the serializer.

New in version 1.4: Prior to this version, there was no public API for supplying options to the underlying serializer without defining a custom renderer.

add_adapter (*type_or_iface*, *adapter*)

When an object of the type (or interface) *type_or_iface* fails to automatically encode using the serializer, the renderer will use the adapter *adapter* to convert it into a JSON-serializable object. The adapter must accept two arguments: the object and the currently active request.

```
class Foo(object):
    x = 5

def foo_adapter(obj, request):
    return obj.x

renderer = JSON(indent=4)
renderer.add_adapter(Foo, foo_adapter)
```

When you've done this, the JSON renderer will be able to serialize instances of the `Foo` class when they're encountered in your view results.

class JSONP (*param_name*='callback', **kw)

JSONP renderer factory helper which implements a hybrid json/jsonp renderer. JSONP is useful for making cross-domain AJAX requests.

Configure a JSONP renderer using the `pyramid.config.Configurator.add_renderer()` API at application startup time:

```
from pyramid.config import Configurator

config = Configurator()
config.add_renderer('jsonp', JSONP(param_name='callback'))
```

The class' constructor also accepts arbitrary keyword arguments. All keyword arguments except *param_name* are passed to the `json.dumps` function as its keyword arguments.

```
from pyramid.config import Configurator

config = Configurator()
config.add_renderer('jsonp', JSONP(param_name='callback', indent=4))
```

Changed in version 1.4: The ability of this class to accept a `**kw` in its constructor.

The arguments passed to this class' constructor mean the same thing as the arguments passed to `pyramid.renderers.JSON` (including `serializer` and `adapters`).

Once this renderer is registered via `add_renderer()` as above, you can use `jsonp` as the `renderer=` parameter to `@view_config` or `pyramid.config.Configurator.add_view`()`:

```
from pyramid.view import view_config

@view_config(renderer='jsonp')
def myview(request):
    return {'greeting': 'Hello world'}
```

When a view is called that uses the JSONP renderer:

- If there is a parameter in the request's HTTP query string that matches the `param_name` of the registered JSONP renderer (by default, `callback`), the renderer will return a JSONP response.
- If there is no `callback` parameter in the request's query string, the renderer will return a 'plain' JSON response.

New in version 1.1.

See also:

See also *JSONP Renderer*.

add_adapter (*type_or_iface*, *adapter*)

When an object of the type (or interface) `type_or_iface` fails to automatically encode using the serializer, the renderer will use the adapter `adapter` to convert it into a JSON-serializable object. The adapter must accept two arguments: the object and the currently active request.

```
class Foo(object):
    x = 5

def foo_adapter(obj, request):
    return obj.x

renderer = JSON(indent=4)
renderer.add_adapter(Foo, foo_adapter)
```

When you've done this, the JSON renderer will be able to serialize instances of the `Foo` class when they're encountered in your view results.

`null_renderer`

An object that can be used in advanced integration cases as input to the view configuration `renderer=` argument. When the null renderer is used as a view renderer argument, Pyramid avoids converting the view callable result into a `Response` object. This is useful if you want to reuse the view configuration and lookup machinery outside the context of its use by the Pyramid router.

`pyramid.request`

class `Request` (*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*,
 ***kw*)

A subclass of the *WebOb* Request class. An instance of this class is created by the *router* and is provided to a view callable (and to other subsystems) as the `request` argument.

The documentation below (save for the `add_response_callback` and `add_finished_callback` methods, which are defined in this subclass itself, and the attributes `context`, `registry`, `root`, `subpath`, `traversed`, `view_name`, `virtual_root`, and `virtual_root_path`, each of which is added to the request by the *router* at request ingress time) are autogenerated from the *WebOb* source code used when this documentation was generated.

Due to technical constraints, we can't yet display the *WebOb* version number from which this documentation is autogenerated, but it will be the 'prevailing *WebOb* version' at the time of the release of this Pyramid version. See <http://webob.org/> for further information.

`context`

The *context* will be available as the `context` attribute of the *request* object. It will be the context object implied by the current request. See *Traversal* for information about context objects.

`registry`

The *application registry* will be available as the `registry` attribute of the *request* object. See *Using the Zope Component Architecture in Pyramid* for more information about the application registry.

`root`

The *root* object will be available as the `root` attribute of the *request* object. It will be the resource object at which traversal started (the root). See *Traversal* for information about root objects.

subpath

The traversal *subpath* will be available as the `subpath` attribute of the *request* object. It will be a sequence containing zero or more elements (which will be Unicode objects). See *Traversal* for information about the subpath.

traversed

The “traversal path” will be available as the `traversed` attribute of the *request* object. It will be a sequence representing the ordered set of names that were used to traverse to the *context*, not including the view name or subpath. If there is a virtual root associated with the request, the virtual root path is included within the traversal path. See *Traversal* for more information.

view_name

The *view name* will be available as the `view_name` attribute of the *request* object. It will be a single string (possibly the empty string if we’re rendering a default view). See *Traversal* for information about view names.

virtual_root

The *virtual root* will be available as the `virtual_root` attribute of the *request* object. It will be the virtual root object implied by the current request. See *Virtual Hosting* for more information about virtual roots.

virtual_root_path

The *virtual root path* will be available as the `virtual_root_path` attribute of the *request* object. It will be a sequence representing the ordered set of names that were used to traverse to the virtual root object. See *Virtual Hosting* for more information about virtual roots.

exception

If an exception was raised by a *root factory* or a *view callable*, or at various other points where Pyramid executes user-defined code during the processing of a request, the exception object which was caught will be available as the `exception` attribute of the request within a *exception view*, a *response callback* or a *finished callback*. If no exception occurred, the value of `request.exception` will be `None` within response and finished callbacks.

exc_info

If an exception was raised by a *root factory* or a *view callable*, or at various other points where Pyramid executes user-defined code during the processing of a request, result of `sys.exc_info()` will be available as the `exc_info` attribute of the request within a *exception view*, a *response callback* or a *finished callback*. If no exception occurred, the value of `request.exc_info` will be `None` within response and finished callbacks.

response

This attribute is actually a “reified” property which returns an instance of the `pyramid.response.Response` class. The response object returned does not exist until this attribute is accessed. Once it is accessed, subsequent accesses to this request object will return the same `Response` object.

The `request.response` API can be used by renderers. A renderer obtains the response object it will return from a view that uses that renderer by accessing `request.response`. Therefore, it’s possible to use the `request.response` API to set up a response object with “the right” attributes (e.g. by calling `request.response.set_cookie(...)` or `request.response.content_type = 'text/plain'`, etc) within a view that uses a renderer. For example, within a view that uses a *renderer*:

```
response = request.response
response.set_cookie('mycookie', 'mine, all mine!')
return {'text': 'Value that will be used by the renderer'}
```

Mutations to this response object will be preserved in the response sent to the client after rendering. For more information about using `request.response` in conjunction with a renderer, see *Varying Attributes of Rendered Responses*.

Non-renderer code can also make use of `request.response` instead of creating a response “by hand”. For example, in view code:

```
response = request.response
response.body = 'Hello!'
response.content_type = 'text/plain'
return response
```

Note that the response in this circumstance is not “global”; it still must be returned from the view code if a renderer is not used.

session

If a *session factory* has been configured, this attribute will represent the current user’s *session* object. If a session factory *has not* been configured, requesting the `request.session` attribute will cause a `pyramid.exceptions.ConfigurationError` to be raised.

matchdict

If a *route* has matched during this request, this attribute will be a dictionary containing the values matched by the URL pattern associated with the route. If a route has not matched during this request, the value of this attribute will be `None`. See *The Matchdict*.

matched_route

If a *route* has matched during this request, this attribute will be an object representing the route matched by the URL pattern associated with the route. If a route has not matched during this request, the value of this attribute will be `None`. See *The Matched Route*.

authenticated_userid

New in version 1.5.

A property which returns the *userid* of the currently authenticated user or `None` if there is no *authentication policy* in effect or there is no currently authenticated user. This differs from *unauthenticated_userid*, because the effective authentication policy will have ensured that a record associated with the *userid* exists in persistent storage; if it has not, this value will be `None`.

unauthenticated_userid

New in version 1.5.

A property which returns a value which represents the *claimed* (not verified) *userid* of the credentials present in the request. `None` if there is no *authentication policy* in effect or there is no user data associated with the current request. This differs from *authenticated_userid*, because the effective authentication policy will not ensure that a record associated with the *userid* exists in persistent storage. Even if the *userid* does not exist in persistent storage, this value will be the value of the *userid claimed* by the request data.

effective_principals

New in version 1.5.

A property which returns the list of ‘effective’ *principal* identifiers for this request. This list typically includes the *userid* of the currently authenticated user if a user is currently authenticated, but this depends on the *authentication policy* in effect. If no *authentication policy* is in effect, this will return a sequence containing only the *pyramid.security.Everyone* principal.

invoke_subrequest (*request*, *use_tweens=False*)

New in version 1.4a1.

Obtain a response object from the Pyramid application based on information in the *request* object provided. The *request* object must be an object that implements the Pyramid request interface (such as a *pyramid.request.Request* instance). If *use_tweens* is `True`, the request will be sent to the *tween* in the tween stack closest to the request ingress. If *use_tweens* is `False`, the request will be sent to the main router handler, and no tweens will be invoked.

This function also:

- manages the threadlocal stack (so that `get_current_request()` and `get_current_registry()` work during a request)
- Adds a `registry` attribute (the current Pyramid registry) and a `invoke_subrequest` attribute (a callable) to the request object it's handed.
- sets request extensions (such as those added via `add_request_method()` or `set_request_property()`) on the request it's passed.
- causes a `NewRequest` event to be sent at the beginning of request processing.
- causes a `ContextFound` event to be sent when a context resource is found.
- Ensures that the user implied by the request passed has the necessary authorization to invoke view callable before calling it.
- Calls any *response callback* functions defined within the request's lifetime if a response is obtained from the Pyramid application.
- causes a `NewResponse` event to be sent if a response is obtained.
- Calls any *finished callback* functions defined within the request's lifetime.

`invoke_subrequest` isn't *actually* a method of the Request object; it's a callable added when the Pyramid router is invoked, or when a subrequest is invoked. This means that it's not available for use on a request provided by e.g. the `pshell` environment.

See also:

See also *Invoking a Subrequest*.

`invoke_exception_view` (*exc_info=None, request=None, secure=True*)

Executes an exception view related to the request it's called upon. The arguments it takes are these:

`exc_info`

If provided, should be a 3-tuple in the form provided by `sys.exc_info()`. If not provided, `sys.exc_info()` will be called to obtain the current interpreter exception information. Default: `None`.

`request`

If the request to be used is not the same one as the instance that this method is called upon, it may be passed here. Default: `None`.

`secure`

If the exception view should not be rendered if the current user does not have the appropriate permission, this should be `True`. Default: `True`.

If called with no arguments, it uses the global exception information returned by `sys.exc_info()` as `exc_info`, the request object that this method is attached to as the `request`, and `True` for `secure`.

This method returns a *response* object or raises `pyramid.httpexceptions.HTTPNotFound` if a matching view cannot be found.

New in version 1.7.

has_permission (*permission*, *context=None*)

Given a permission and an optional context, returns an instance of `pyramid.security.Allowed` if the permission is granted to this request with the provided context, or the context already associated with the request. Otherwise, returns an instance of `pyramid.security.Denied`. This method delegates to the current authentication and authorization policies. Returns `pyramid.security.Allowed` unconditionally if no authentication policy has been registered for this request. If *context* is not supplied or is supplied as `None`, the context used is the `request.context` attribute.

Parameters

- **permission** (*unicode*, *str*) – Does this request have the given permission?
- **context** (*object*) – A resource object or `None`

Returns `pyramid.security.PermitsResult`

New in version 1.5.

add_response_callback (*callback*)

Add a callback to the set of callbacks to be called by the *router* at a point after a *response* object is successfully created. Pyramid does not have a global response object: this functionality allows an application to register an action to be performed against the response once one is created.

A ‘callback’ is a callable which accepts two positional parameters: `request` and `response`. For example:

```
1 def cache_callback(request, response):
2     'Set the cache_control max_age for the response'
3     response.cache_control.max_age = 360
4     request.add_response_callback(cache_callback)
```

Response callbacks are called in the order they're added (first-to-most-recently-added). No response callback is called if an exception happens in application code, or if the response object returned by *view* code is invalid.

All response callbacks are called *after* the tweens and *before* the *pyramid.events.NewResponse* event is sent.

Errors raised by callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

See also:

See also *Using Response Callbacks*.

add_finished_callback (*callback*)

Add a callback to the set of callbacks to be called unconditionally by the *router* at the very end of request processing.

callback is a callable which accepts a single positional parameter: *request*. For example:

```
1 import transaction
2
3 def commit_callback(request):
4     '''commit or abort the transaction associated with request'''
5     if request.exception is not None:
6         transaction.abort()
7     else:
8         transaction.commit()
9     request.add_finished_callback(commit_callback)
```

Finished callbacks are called in the order they're added (first- to most-recently- added). Finished callbacks (unlike response callbacks) are *always* called, even if an exception happens in application code that prevents a response from being generated.

The set of finished callbacks associated with a request are called *very late* in the processing of that request; they are essentially the last thing called by the *router*. They are called after

response processing has already occurred in a top-level `finally:` block within the router request processing code. As a result, mutations performed to the `request` provided to a finished callback will have no meaningful effect, because response processing will have already occurred, and the request's scope will expire almost immediately after all finished callbacks have been processed.

Errors raised by finished callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

See also:

See also *Using Finished Callbacks*.

route_url (*route_name*, **elements*, ***kw*)

Generates a fully qualified URL for a named Pyramid *route configuration*.

Use the route's `name` as the first positional argument. Additional positional arguments (`*elements`) are appended to the URL as path segments after it is generated.

Use keyword arguments to supply values which match any dynamic path elements in the route definition. Raises a `KeyError` exception if the URL cannot be generated for any reason (not enough arguments, for example).


For example, if you've defined a route named "foobar" with the path `{foo}/{bar}/{*traverse}`:

```
request.route_url('foobar',
                  foo='1')           => <KeyError exception>
request.route_url('foobar',
                  foo='1',
                  bar='2')          => <KeyError exception>
request.route_url('foobar',
                  foo='1',
                  bar='2',
                  traverse=('a', 'b')) => http://e.com/1/2/a/b
request.route_url('foobar',
                  foo='1',
                  bar='2',
                  traverse='/a/b')   => http://e.com/1/2/a/b
```

Values replacing `:segment` arguments can be passed as strings or Unicode objects. They will be encoded to UTF-8 and URL-quoted before being placed into the generated URL.


Values replacing `*remainder` arguments can be passed as strings *or* tuples of Unicode/string values. If a tuple is passed as a `*remainder` replacement value, its values are URL-quoted and encoded to UTF-8. The resulting strings are joined with slashes and rendered into the URL. If a string is passed as a `*remainder` replacement value, it is tacked on to the URL after being URL-quoted-except-for-embedded-slashes.

If no `_query` keyword argument is provided, the request query string will be returned in the URL. If it is present, it will be used to compose a query string that will be tacked on to the end of the URL, replacing any request query string. The value of `_query` may be a sequence of two-tuples *or* a data structure with an `.items()` method that returns a sequence of two-tuples (presumably a dictionary). This data structure will be turned into a query string per the documentation of `pyramid.url.urlencode()` function. This will produce a query string in the `x-www-form-urlencoded` format. A `non-x-www-form-urlencoded` query string may be used by passing a *string* value as `_query` in which case it will be URL-quoted (e.g. `query="foo bar"` will become `"foo%20bar"`). However, the result will not need to be in `k=v` form as required by `x-www-form-urlencoded`. After the query data is turned into a query string, a leading `?` is prepended, and the resulting string is appended to the generated URL.

 Python data structures that are passed as `_query` which are sequences or dictionaries are turned into a string under the same rules as when run through `urllib.urlencode()` with the `doseq` argument equal to `True`. This means that sequences can be passed as values, and a `k=v` pair will be placed into the query string for each value.

Changed in version 1.5: Allow the `_query` option to be a string to enable alternative encodings.

If a keyword argument `_anchor` is present, its string representation will be quoted per **RFC 3986#section-3.5** and used as a named anchor in the generated URL (e.g. if `_anchor` is passed as `foo` and the route URL is `http://example.com/route/url`, the resulting generated URL will be `http://example.com/route/url#foo`).

 If `_anchor` is passed as a string, it should be UTF-8 encoded. If `_anchor` is passed as a Unicode object, it will be converted to UTF-8 before being appended to the URL.

Changed in version 1.5: The `_anchor` option will be escaped instead of using its raw string representation.

If both `_anchor` and `_query` are specified, the anchor element will always follow the query element, e.g. `http://example.com?foo=1#bar`.

If any of the keyword arguments `_scheme`, `_host`, or `_port` is passed and is non-None, the provided value will replace the named portion in the generated URL. For example, if you pass `_host='foo.com'`, and the URL that would have been generated without the host replacement is `http://example.com/a`, the result will be `http://foo.com/a`.

Note that if `_scheme` is passed as `https`, and `_port` is not passed, the `_port` value is assumed to have been passed as 443. Likewise, if `_scheme` is passed as `http` and `_port` is not passed, the `_port` value is assumed to have been passed as 80. To avoid this behavior, always explicitly pass `_port` whenever you pass `_scheme`.

If a keyword `_app_url` is present, it will be used as the protocol/hostname/port/leading path prefix of the generated URL. For example, using an `_app_url` of `http://example.com:8080/foo` would cause the URL `http://example.com:8080/foo/fleeb/flub` to be returned from this function if the expansion of the route pattern associated with the `route_name` expanded to `/fleeb/flub`. If `_app_url` is not specified, the result of `request.application_url` will be used as the prefix (the default).

If both `_app_url` and any of `_scheme`, `_host`, or `_port` are passed, `_app_url` takes precedence and any values passed for `_scheme`, `_host`, and `_port` will be ignored.

This function raises a `KeyError` if the URL cannot be generated due to missing replacement names. Extra replacement names are ignored.

If the route object which matches the `route_name` argument has a *pregenerator*, the `*elements` and `**kw` arguments passed to this function might be augmented or changed.

`route_path`(*route_name*, **elements*, ***kw*)

Generates a path (aka a ‘relative URL’, a URL minus the host, scheme, and port) for a named Pyramid *route configuration*.

This function accepts the same argument as `pyramid.request.Request.route_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the script_name, path, query parameters, and anchor data are present in the returned string.

For example, if you’ve defined a route named ‘foobar’ with the path `/ {foo} / {bar}`, this call to `route_path`:

```
request.route_path('foobar', foo='1', bar='2')
```

Will return the string `/1/2`.

i Calling `request.route_path('route')` is the same as calling `request.route_url('route', _app_url=request.script_name)`. *pyramid.request.Request.route_path()* is, in fact, implemented in terms of *pyramid.request.Request.route_url()* in just this way. As a result, any `_app_url` passed within the `**kw` values to `route_path` will be ignored.

`current_route_url(*elements, **kw)`

Generates a fully qualified URL for a named Pyramid *route configuration* based on the ‘current route’.

This function supplements *pyramid.request.Request.route_url()*. It presents an easy way to generate a URL for the ‘current route’ (defined as the route which matched when the request was generated).

The arguments to this method have the same meaning as those with the same names passed to *pyramid.request.Request.route_url()*. It also understands an extra argument which `route_url` does not named `_route_name`.

The route name used to generate a URL is taken from either the `_route_name` keyword argument or the name of the route which is currently associated with the request if `_route_name` was not passed. Keys and values from the current request *matchdict* are combined with the `kw` arguments to form a set of defaults named `newkw`. Then `request.route_url(route_name, *elements, **newkw)` is called, returning a URL.

Examples follow.

If the ‘current route’ has the route pattern `/foo/{page}` and the current url path is `/foo/1`, the *matchdict* will be `{'page': '1'}`. The result of `request.current_route_url()` in this situation will be `/foo/1`.

If the ‘current route’ has the route pattern `/foo/{page}` and the current url path is `/foo/1`, the *matchdict* will be `{'page': '1'}`. The result of `request.current_route_url(page='2')` in this situation will be `/foo/2`.

Usage of the `_route_name` keyword argument: if our routing table defines routes `/foo/{action}` named ‘foo’ and `/foo/{action}/{page}` named `fooaction`, and the current url pattern is `/foo/view` (which has matched the `/foo/{action}` route), we may want to use the *matchdict* args to generate a URL to the `fooaction` route. In this scenario, `request.current_route_url(_route_name='fooaction', page='5')` Will return string like: `/foo/view/5`.

current_route_path(*elements, **kw)

Generates a path (aka a 'relative URL', a URL minus the host, scheme, and port) for the Pyramid *route configuration* matched by the current request.

This function accepts the same argument as `pyramid.request.Request.current_route_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the script_name, path, query parameters, and anchor data are present in the returned string.

For example, if the route matched by the current request has the pattern `{foo}/{bar}`, this call to `current_route_path`:

```
request.current_route_path(foo='1', bar='2')
```

Will return the string `/1/2`.



Calling `request.current_route_path('route')` is the same as calling `request.current_route_url('route', _app_url=request.script_name)`. `pyramid.request.Request.current_route_path()` is, in fact, implemented in terms of `pyramid.request.Request.current_route_url()` in just this way. As a result, any `_app_url` passed within the `**kw` values to `current_route_path` will be ignored.

static_url(path, **kw)

Generates a fully qualified URL for a static *asset*. The asset must live within a location defined via the `pyramid.config.Configurator.add_static_view()` *configuration declaration* (see *Serving Static Assets*).

Example:

```
request.static_url('mypackage:static/foo.css') =>
http://example.com/static/foo.css
```

The path argument points at a file or directory on disk which a URL should be generated for. The path may be either a relative path (e.g. `static/foo.css`) or an absolute path (e.g. `/abspath/to/static/foo.css`) or a *asset specification* (e.g. `mypackage:static/foo.css`).

The purpose of the `**kw` argument is the same as the purpose of the `pyramid.request.Request.route_url()` `**kw` argument. See the documentation for that function to understand the arguments which you can provide to it. However, typically, you don't need to pass anything as `*kw` when generating a static asset URL.

This function raises a `ValueError` if a static view definition cannot be found which matches the path specification.


static_path (*path*, ***kw*)

Generates a path (aka a 'relative URL', a URL minus the host, scheme, and port) for a static resource.

This function accepts the same argument as `pyramid.request.Request.static_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the script_name, path, query parameters, and anchor data are present in the returned string.

Example:

```
request.static_path('mypackage:static/foo.css') =>
                                         /static/foo.css
```

 Calling `request.static_path(aphath)` is the same as calling `request.static_url(aphath, _app_url=request.script_name)`. `pyramid.request.Request.static_path()` is, in fact, implemented in terms of `:meth:'pyramid.request.Request.static_url'` in just this way. As a result, any `_app_url` passed within the `**kw` values to `static_path` will be ignored.

resource_url (*resource*, **elements*, ***kw*)

Generate a string representing the absolute URL of the *resource* object based on the `wsgi.url_scheme`, `HTTP_HOST` or `SERVER_NAME` in the request, plus any `SCRIPT_NAME`. The overall result of this method is always a UTF-8 encoded string.

Examples:

```
request.resource_url(resource) =>
                               http://example.com/
request.resource_url(resource, 'a.html') =>
```

```

http://example.com/a.html

request.resource_url(resource, 'a.html', query={'q':'1'}) =>

http://example.com/a.html?q=1

request.resource_url(resource, 'a.html', anchor='abc') =>

http://example.com/a.html#abc

request.resource_url(resource, app_url='') =>

/

```

Any positional arguments passed in as `elements` must be strings Unicode objects, or integer objects. These will be joined by slashes and appended to the generated resource URL. Each of the elements passed in is URL-quoted before being appended; if any element is Unicode, it will be converted to a UTF-8 bytestring before being URL-quoted. If any element is an integer, it will be converted to its string representation before being URL-quoted.



if no `elements` arguments are specified, the resource URL will end with a trailing slash. If any `elements` are used, the generated URL will *not* end in a trailing slash.

If a keyword argument `query` is present, it will be used to compose a query string that will be tacked on to the end of the URL. The value of `query` may be a sequence of two-tuples *or* a data structure with an `.items()` method that returns a sequence of two-tuples (presumably a dictionary). This data structure will be turned into a query string per the documentation of `pyramid.url.urlencode()` function. This will produce a query string in the `x-www-form-urlencoded` encoding. A non-`x-www-form-urlencoded` query string may be used by passing a *string* value as `query` in which case it will be URL-quoted (e.g. `query="foo bar"` will become `"foo%20bar"`). However, the result will not need to be in `k=v` form as required by `x-www-form-urlencoded`. After the query data is turned into a query string, a leading `?` is prepended, and the resulting string is appended to the generated URL.



Python data structures that are passed as `query` which are sequences or dictionaries are turned into a string under the same rules as when run through `urllib.urlencode()`

with the `doseq` argument equal to `True`. This means that sequences can be passed as values, and a `k=v` pair will be placed into the query string for each value.

Changed in version 1.5: Allow the `query` option to be a string to enable alternative encodings.

If a keyword argument `anchor` is present, its string representation will be used as a named anchor in the generated URL (e.g. if `anchor` is passed as `foo` and the resource URL is `http://example.com/resource/url`, the resulting generated URL will be `http://example.com/resource/url#foo`).



If `anchor` is passed as a string, it should be UTF-8 encoded. If `anchor` is passed as a Unicode object, it will be converted to UTF-8 before being appended to the URL.

Changed in version 1.5: The `anchor` option will be escaped instead of using its raw string representation.

If both `anchor` and `query` are specified, the anchor element will always follow the query element, e.g. `http://example.com?foo=1#bar`.

If any of the keyword arguments `scheme`, `host`, or `port` is passed and is non-`None`, the provided value will replace the named portion in the generated URL. For example, if you pass `host='foo.com'`, and the URL that would have been generated without the host replacement is `http://example.com/a`, the result will be `http://foo.com/a`.

If `scheme` is passed as `https`, and an explicit `port` is not passed, the `port` value is assumed to have been passed as 443. Likewise, if `scheme` is passed as `http` and `port` is not passed, the `port` value is assumed to have been passed as 80. To avoid this behavior, always explicitly pass `port` whenever you pass `scheme`.

If a keyword argument `app_url` is passed and is not `None`, it should be a string that will be used as the port/hostname/initial path portion of the generated URL instead of the default request application URL. For example, if `app_url='http://foo'`, then the resulting url of a resource that has a path of `/baz/bar` will be `http://foo/baz/bar`. If you want to generate completely relative URLs with no leading scheme, host, port, or initial path, you can pass `app_url=''`. Passing `app_url=''` when the resource path is `/baz/bar` will return `/baz/bar`.

New in version 1.3: `app_url`

If `app_url` is passed and any of `scheme`, `port`, or `host` are also passed, `app_url` will take precedence and the values passed for `scheme`, `host`, and/or `port` will be ignored.

If the `resource` passed in has a `__resource_url__` method, it will be used to generate the URL (scheme, host, port, path) for the base resource which is operated upon by this function.

See also:

See also *Overriding Resource URL Generation*.

New in version 1.5: `route_name`, `route_kw`, and `route_remainder_name`

If `route_name` is passed, this function will delegate its URL production to the `route_url` function. Calling `resource_url(someresource, 'element1', 'element2', query={'a':1}, route_name='blogentry')` is roughly equivalent to doing:

```
traversal_path = request.resource_path(someobject)
url = request.route_url(
    'blogentry',
    'element1',
    'element2',
    _query={'a':1},
    traverse=traversal_path,
)
```

It is only sensible to pass `route_name` if the route being named has a `*remainder` stararg value such as `*traverse`. The remainder value will be ignored in the output otherwise.


By default, the resource path value will be passed as the name `traverse` when `route_url` is called. You can influence this by passing a different `route_remainder_name` value if the route has a different `*stararg` value at its end. For example if the route pattern you want to replace has a `*subpath` stararg ala `/foo*subpath`:


```
request.resource_url(
    resource,
    route_name='myroute',
    route_remainder_name='subpath'
)
```


If `route_name` is passed, it is also permissible to pass `route_kw`, which will be passed as additional keyword arguments to `route_url`. Saying `resource_url(someresource, 'element1', 'element2', route_name='blogentry', route_kw={'id':'4'}, _query={'a':1})` is roughly equivalent to:

```
traversal_path = request.resource_path_tuple(someobject)
kw = {'id': '4', '_query': {'a': '1'}, 'traverse': traversal_path}
url = request.route_url(
    'blogentry',
    'element1',
    'element2',
    **kw,
)
```

If `route_kw` or `route_remainder_name` is passed, but `route_name` is not passed, both `route_kw` and `route_remainder_name` will be ignored. If `route_name` is passed, the `__resource_url__` method of the resource passed is ignored unconditionally. This feature is incompatible with resources which generate their own URLs.

 If the *resource* used is the result of a *traversal*, it must be *location*-aware. The resource can also be the context of a *URL dispatch*; contexts found this way do not need to be *location*-aware.

 If a ‘virtual root path’ is present in the request environment (the value of the WSGI environ key `HTTP_X_VHM_ROOT`), and the resource was obtained via *traversal*, the URL path will not include the virtual root prefix (it will be stripped off the left hand side of the generated URL).

 For backwards compatibility purposes, this method is also aliased as the `model_url` method of request.

resource_path (*resource*, **elements*, ***kw*)

Generates a path (aka a ‘relative URL’, a URL minus the host, scheme, and port) for a *resource*.

This function accepts the same argument as `pyramid.request.Request.resource_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the `script_name`, `path`, `query` parameters, and `anchor` data are present in the returned string.



Calling `request.resource_path(resource)` is the same as calling `request.resource_path(resource, app_url=request.script_name)`. `pyramid.request.Request.resource_path()` is, in fact, implemented in terms of `pyramid.request.Request.resource_url()` in just this way. As a result, any `app_url` passed within the `**kw` values to `route_path` will be ignored. `scheme`, `host`, and `port` are also ignored.

json_body

This property will return the JSON-decoded variant of the request body. If the request body is not well-formed JSON, or there is no body associated with this request, this property will raise an exception.

See also:

See also *Dealing with a JSON-Encoded Request Body*.

set_property (*callable*, *name=None*, *reify=False*)

Add a callable or a property descriptor to the request instance.

Properties, unlike attributes, are lazily evaluated by executing an underlying callable when accessed. They can be useful for adding features to an object without any cost if those features go unused.

A property may also be reified via the `pyramid.decorator.reify` decorator by setting `reify=True`, allowing the result of the evaluation to be cached. Thus the value of the property is only computed once for the lifetime of the object.

`callable` can either be a callable that accepts the request as its single positional parameter, or it can be a property descriptor.

If the `callable` is a property descriptor a `ValueError` will be raised if `name` is `None` or `reify` is `True`.

If `name` is `None`, the name of the property will be computed from the name of the `callable`.

```

1 def _connect(request):
2     conn = request.registry.dbsession()
3     def cleanup(request):
4         # since version 1.5, request.exception is no
5         # longer eagerly cleared
6         if request.exception is not None:
7             conn.rollback()

```

```
8         else:
9             conn.commit()
10            conn.close()
11            request.add_finished_callback(cleanup)
12            return conn
13
14    @subscriber(NewRequest)
15    def new_request(event):
16        request = event.request
17        request.set_property(_connect, 'db', reify=True)
```

The subscriber doesn't actually connect to the database, it just provides the API which, when accessed via `request.db`, will create the connection. Thanks to `reify`, only one connection is made per-request even if `request.db` is accessed many times.

This pattern provides a way to augment the `request` object without having to subclass it, which can be useful for extension authors.

New in version 1.3.

localizer

A *localizer* which will use the current locale name to translate values.

New in version 1.5.

locale_name

The locale name of the current request as computed by the *locale negotiator*.

New in version 1.5.

GET

Return a MultiDict containing all the variables from the `QUERY_STRING`.

POST

Return a MultiDict containing all the variables from a form request. Returns an empty dict-like object for non-form requests.

Form requests are typically POST requests, however PUT & PATCH requests with an appropriate Content-Type are also supported.

accept

Gets and sets the `Accept` header (HTTP spec section 14.1).

accept_charset

Gets and sets the `Accept-Charset` header (HTTP spec section 14.2).

accept_encoding

Gets and sets the `Accept-Encoding` header (HTTP spec section 14.3).

accept_language

Gets and sets the `Accept-Language` header (HTTP spec section 14.4).

application_url

The URL including `SCRIPT_NAME` (no `PATH_INFO` or query string)

as_bytes (*skip_body=False*)

Return HTTP bytes representing this request. If `skip_body` is `True`, exclude the body. If `skip_body` is an integer larger than one, skip body only if its length is bigger than that number.

authorization

Gets and sets the `Authorization` header (HTTP spec section 14.8). Converts it using `parse_auth` and `serialize_auth`.

blank (*path, environ=None, base_url=None, headers=None, POST=None, **kw*)

Create a blank request `environ` (and Request wrapper) with the given path (path should be urlencoded), and any keys from `environ`.

The path will become `path_info`, with any query string split off and used.

All necessary keys will be added to the `environ`, but the values you pass in will take precedence. If you pass in `base_url` then `wsgi.url_scheme`, `HTTP_HOST`, and `SCRIPT_NAME` will be filled in from that value.

Any extra keyword will be passed to `__init__`.

body

Return the content of the request body.

body_file

Input stream of the request (`wsgi.input`). Setting this property resets the `content_length` and seekable flag (unlike setting `req.body_file_raw`).

body_file_raw

Gets and sets the `wsgi.input` key in the environment.

body_file_seekable

Get the body of the request (`wsgi.input`) as a seekable file-like object. Middleware and routing applications should use this attribute over `.body_file`.

If you access this value, `CONTENT_LENGTH` will also be updated.

cache_control

Get/set/modify the Cache-Control header (HTTP spec section 14.9)

call_application (*application*, *catch_exc_info=False*)

Call the given WSGI application, returning (`status_string`, `headerlist`, `app_iter`)

Be sure to call `app_iter.close()` if it's there.

If `catch_exc_info` is true, then returns (`status_string`, `headerlist`, `app_iter`, `exc_info`), where the fourth item may be `None`, but won't be if there was an exception. If you don't do this and there was an exception, the exception will be raised directly.

client_addr

The effective client IP address as a string. If the `HTTP_X_FORWARDED_FOR` header exists in the WSGI environ, this attribute returns the client IP address present in that header (e.g. if the header value is `192.168.1.1, 192.168.1.2`, the value will be `192.168.1.1`). If no `HTTP_X_FORWARDED_FOR` header is present in the environ at all, this attribute will return the value of the `REMOTE_ADDR` header. If the `REMOTE_ADDR` header is unset, this attribute will return the value `None`.



It is possible for user agents to put someone else's IP or just any string in `HTTP_X_FORWARDED_FOR` as it is a normal HTTP header. Forward proxies can also provide incorrect values (private IP addresses etc). You cannot "blindly" trust the result of this method to provide you with valid data unless you're certain that `HTTP_X_FORWARDED_FOR` has the correct values. The WSGI server must be behind a trusted proxy for this to be true.

content_length

Gets and sets the `Content-Length` header (HTTP spec section 14.13). Converts it using `int`.

content_type

Return the content type, but leaving off any parameters (like `charset`, but also things like the type in `application/atom+xml; type=entry`)

If you set this property, you can include parameters, or if you don't include any parameters in the value then existing parameters will be preserved.

cookies

Return a dictionary of cookies as found in the request.

copy()

Copy the request and environment object.

This only does a shallow copy, except of `wsgi.input`

copy_body()

Copies the body, in cases where it might be shared with another request object and that is not desired.

This copies the body either into a BytesIO object (through setting `req.body`) or a temporary file.

copy_get()

Copies the request and environment object, but turning this request into a GET along the way. If this was a POST request (or any other verb) then it becomes GET, and the request body is thrown away.

date

Gets and sets the `Date` header (HTTP spec section 14.8). Converts it using HTTP date.

domain

Returns the domain portion of the host value. Equivalent to:

```
domain = request.host
if ':' in domain:
    domain = domain.split(':', 1)[0]
```

This will be equivalent to the domain portion of the `HTTP_HOST` value in the environment if it exists, or the `SERVER_NAME` value in the environment if it doesn't. For example, if the environment contains an `HTTP_HOST` value of `foo.example.com:8000`, `request.domain` will return `foo.example.com`.

Note that this value cannot be *set* on the request. To set the host value use `webob.request.Request.host()` instead.

from_bytes(b)

Create a request from HTTP bytes data. If the bytes contain extra data after the request, raise a `ValueError`.

from_file (*fp*)

Read a request from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the request, not the end of the file (unless the request is a POST or PUT and has no Content-Length, in that case, the entire file is read).

This reads the request as represented by `str(req)`; it may not read every valid HTTP request properly.

get_response (*application=None, catch_exc_info=False*)

Like `.call_application(application)`, except returns a response object with `.status`, `.headers`, and `.body` attributes.

This will use `self.ResponseClass` to figure out the class of the response object to return.

If `application` is not given, this will send the request to `self.make_default_send_app()`

headers

All the request headers as a case-insensitive dictionary-like object.

host

Host name provided in `HTTP_HOST`, with fall-back to `SERVER_NAME`

host_port

The effective server port number as a string. If the `HTTP_HOST` header exists in the WSGI environ, this attribute returns the port number present in that header. If the `HTTP_HOST` header exists but contains no explicit port number: if the WSGI url scheme is “https”, this attribute returns “443”, if the WSGI url scheme is “http”, this attribute returns “80”. If no `HTTP_HOST` header is present in the environ at all, this attribute will return the value of the `SERVER_PORT` header (which is guaranteed to be present).

host_url

The URL through the host (no path)

http_version

Gets and sets the `SERVER_PROTOCOL` key in the environment.

if_match

Gets and sets the `If-Match` header (HTTP spec section 14.24). Converts it as a Etag.

if_modified_since

Gets and sets the `If-Modified-Since` header (HTTP spec section 14.25). Converts it using HTTP date.

if_none_match

Gets and sets the If-None-Match header (HTTP spec section 14.26). Converts it as a Etag.

if_range

Gets and sets the If-Range header (HTTP spec section 14.27). Converts it using IfRange object.

if_unmodified_since

Gets and sets the If-Unmodified-Since header (HTTP spec section 14.28). Converts it using HTTP date.

is_body_readable

webob.is_body_readable is a flag that tells us that we can read the input stream even though CONTENT_LENGTH is missing.

is_body_seekable

Gets and sets the webob.is_body_seekable key in the environment.

is_response (*ob*)

Return True if the object passed as ob is a valid response object, False otherwise.

is_xhr

Is X-Requested-With header present and equal to XMLHttpRequest?

Note: this isn't set by every XMLHttpRequest request, it is only set if you are using a Javascript library that sets it (or you set the header yourself manually). Currently Prototype and jQuery are known to set this header.

json

Access the body of the request as JSON

localizer

Convenience property to return a localizer

make_body_seekable ()

This forces `environ['wsgi.input']` to be seekable. That means that, the content is copied into a BytesIO or temporary file and flagged as seekable, so that it will not be unnecessarily copied again.

After calling this method the `.body_file` is always seeked to the start of file and `.content_length` is not None.

The choice to copy to BytesIO is made from `self.request_body_tempfile_limit`

make_tempfile()

Create a tempfile to store big request body. This API is not stable yet. A 'size' argument might be added.

max_forwards

Gets and sets the `Max-Forwards` header (HTTP spec section 14.31). Converts it using `int`.

method

Gets and sets the `REQUEST_METHOD` key in the environment.

params

A dictionary-like object containing both the parameters from the query string and request body.

path

The path of the request, without host or query string

path_info

Gets and sets the `PATH_INFO` key in the environment.

path_info_peek()

Returns the next segment on `PATH_INFO`, or `None` if there is no next segment. Doesn't modify the environment.

path_info_pop (pattern=None)

'Pops' off the next segment of `PATH_INFO`, pushing it onto `SCRIPT_NAME`, and returning the popped segment. Returns `None` if there is nothing left on `PATH_INFO`.

Does not return `' '` when there's an empty segment (like `/path//path`); these segments are just ignored.

Optional `pattern` argument is a regexp to match the return value before returning. If there is no match, no changes are made to the request and `None` is returned.

path_qs

The path of the request, without host but with query string

path_url

The URL including `SCRIPT_NAME` and `PATH_INFO`, but not `QUERY_STRING`

pragma

Gets and sets the `Pragma` header (HTTP spec section 14.32).

query_string

Gets and sets the `QUERY_STRING` key in the environment.

range

Gets and sets the `Range` header (HTTP spec section 14.35). Converts it using `Range` object.

referer

Gets and sets the `Referer` header (HTTP spec section 14.36).

referrer

Gets and sets the `Referer` header (HTTP spec section 14.36).

relative_url (*other_url*, *to_application=False*)

Resolve *other_url* relative to the request URL.

If *to_application* is `True`, then resolve it relative to the URL with only `SCRIPT_NAME`

remote_addr

Gets and sets the `REMOTE_ADDR` key in the environment.

remote_user

Gets and sets the `REMOTE_USER` key in the environment.

remove_conditional_headers (*remove_encoding=True*, *remove_range=True*, *remove_match=True*, *remove_modified=True*)

Remove headers that make the request conditional.

These headers can cause the response to be 304 Not Modified, which in some cases you may not want to be possible.

This does not remove headers like `If-Match`, which are used for conflict detection.

request_iface = <InterfaceClass `pyramid.interfaces.IRequest`>**response**

This attribute is actually a “reified” property which returns an instance of the `pyramid.response.Response` class. The response object returned does not exist until this attribute is accessed. Subsequent accesses will return the same Response object.

The `request.response` API is used by renderers. A render obtains the response object it will return from a view that uses that renderer by accessing `request.response`. Therefore, it’s possible to use the `request.response` API to set up a response object with “the right” attributes (e.g. by calling `request.response.set_cookie()`) within a view that uses a renderer. Mutations to this response object will be preserved in the response sent to the client.

scheme

Gets and sets the `wsgi.url_scheme` key in the environment.

script_name

Gets and sets the `SCRIPT_NAME` key in the environment.

send (*application=None, catch_exc_info=False*)

Like `.call_application(application)`, except returns a response object with `.status`, `.headers`, and `.body` attributes.

This will use `self.ResponseClass` to figure out the class of the response object to return.

If `application` is not given, this will send the request to `self.make_default_send_app()`

server_name

Gets and sets the `SERVER_NAME` key in the environment.

server_port

Gets and sets the `SERVER_PORT` key in the environment. Converts it using `int`.

session

Obtain the *session* object associated with this request. If a *session factory* has not been registered during application configuration, a `pyramid.exceptions.ConfigurationError` will be raised

text

Get/set the text value of the body

upath_info

Gets and sets the `PATH_INFO` key in the environment.

url

The full request URL, including `QUERY_STRING`

url_encoding

Gets and sets the `webob.url_encoding` key in the environment.

urlargs

Return any *positional* variables matched in the URL.

Takes values from `environ['wsgiorg.routing_args']`. Systems like `routes` set this value.

urlvars

Return any *named* variables matched in the URL.


Takes values from `environ['wsgiorg.routing_args']`. Systems like routes set this value.

uscript_name

Gets and sets the `SCRIPT_NAME` key in the environment.

user_agent

Gets and sets the User-Agent header (HTTP spec section 14.43).

 For information about the API of a *multidict* structure (such as that used as `request.GET`, `request.POST`, and `request.params`), see `pyramid.interfaces.IMultiDict`.

apply_request_extensions (*request*)

Apply request extensions (methods and properties) to an instance of `pyramid.interfaces.IRequest`. This method is dependent on the `request` containing a properly initialized registry.

After invoking this method, the `request` should have the methods and properties that were defined using `pyramid.config.Configurator.add_request_method()`.

pyramid.response

class Response (*body=None, status=None, headerlist=None, app_iter=None, content_type=None, conditional_response=None, charset=<object object>, **kw*)

accept_ranges

Gets and sets the Accept-Ranges header (HTTP spec section 14.5).

age

Gets and sets the Age header (HTTP spec section 14.6). Converts it using `int`.

allow

Gets and sets the Allow header (HTTP spec section 14.7). Converts it using `list`.

app_iter

Returns the `app_iter` of the response.

If `body` was set, this will create an `app_iter` from that `body` (a single-item list)

app_iter_range (*start, stop*)

Return a new `app_iter` built from the response `app_iter`, that serves up only the given `start:stop` range.

body

The body of the response, as a `bytes`. This will read in the entire `app_iter` if necessary.

body_file

A file-like object that can be used to write to the body. If you passed in a list `app_iter`, that `app_iter` will be modified by writes.

cache_control

Get/set/modify the Cache-Control header (HTTP spec section 14.9)

charset

Get/set the charset specified in Content-Type.

There is no checking to validate that a `content_type` actually allows for a charset parameter.

conditional_response_app (*environ, start_response*)

Like the normal `__call__` interface, but checks conditional headers:

- If-Modified-Since (304 Not Modified; only on GET, HEAD)
- If-None-Match (304 Not Modified; only on GET, HEAD)
- Range (406 Partial Content; only on GET, HEAD)

content_disposition

Gets and sets the Content-Disposition header (HTTP spec section 19.5.1).

content_encoding

Gets and sets the Content-Encoding header (HTTP spec section 14.11).

content_language

Gets and sets the Content-Language header (HTTP spec section 14.12). Converts it using list.

content_length

Gets and sets the Content-Length header (HTTP spec section 14.17). Converts it using int.

content_location

Gets and sets the `Content-Location` header (HTTP spec section 14.14).

content_md5

Gets and sets the `Content-MD5` header (HTTP spec section 14.14).

content_range

Gets and sets the `Content-Range` header (HTTP spec section 14.16). Converts it using `ContentRange` object.

content_type

Get/set the `Content-Type` header. If no `Content-Type` header is set, this will return `None`.

Changed in version 1.7: Setting a new `Content-Type` will remove all `Content-Type` parameters and reset the charset to the default if the `Content-Type` is `text/*` or XML (`application/xml`, or `*/+xml`)

To preserve all `Content-Type` parameters you may use the following code:

```
resp = Response()
params = resp.content_type_params
resp.content_type = 'application/something'
resp.content_type_params = params
```

content_type_params

A dictionary of all the parameters in the content type.

(This is not a view, set to change, modifications of the dict will not be applied otherwise)

copy()

Makes a copy of the response

date

Gets and sets the `Date` header (HTTP spec section 14.18). Converts it using `HTTP date`.

delete_cookie (*name*, *path*='/', *domain*=None)

Delete a cookie from the client. Note that `path` and `domain` must match how the cookie was originally set.

This sets the cookie to the empty string, and `max_age=0` so that it should expire immediately.

encode_content (*encoding*='gzip', *lazy*=False)

Encode the content with the given encoding (only `gzip` and `identity` are supported).

etag

Gets and sets the `ETag` header (HTTP spec section 14.19). Converts it using Entity tag.

expires

Gets and sets the `Expires` header (HTTP spec section 14.21). Converts it using HTTP date.

from_file (*fp*)

Reads a response from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the response, not the end of the file.

This reads the response as represented by `str(resp)`; it may not read every valid HTTP response properly. Responses must have a `Content-Length`

has_body

Determine if the the response has a *body*. In contrast to simply accessing *body* this method will **not** read the underlying *app_iter*.

headerlist

The list of response headers

headers

The headers in a dictionary-like object

json

Set/get the body of the response as JSON



This will automatically `decode()` the *body* as UTF-8 on `get`, and `encode()` the `json.dumps()` as UTF-8 before assigning to *body*.

json_body

Set/get the body of the response as JSON



This will automatically `decode()` the *body* as UTF-8 on `get`, and `encode()` the `json.dumps()` as UTF-8 before assigning to *body*.

last_modified

Gets and sets the `Last-Modified` header (HTTP spec section 14.29). Converts it using HTTP date.

location

Gets and sets the `Location` header (HTTP spec section 14.30).

md5_etag (*body=None, set_content_md5=False*)

Generate an etag for the response object using an MD5 hash of the body (the *body* parameter, or `self.body` if not given)

Sets `self.etag` If `set_content_md5` is `True` sets `self.content_md5` as well

merge_cookies (*resp*)

Merge the cookies that were set on this response with the given *resp* object (which can be any WSGI application).

If the *resp* is a `webob.Response` object, then the other object will be modified in-place.

pragma

Gets and sets the `Pragma` header (HTTP spec section 14.32).

retry_after

Gets and sets the `Retry-After` header (HTTP spec section 14.37). Converts it using HTTP date or delta seconds.

server

Gets and sets the `Server` header (HTTP spec section 14.38).

set_cookie (*name=None, value='', max_age=None, path='/', domain=None, secure=False, httponly=False, comment=None, expires=None, overwrite=False*)

Set (add) a cookie for the response.

Arguments are:

`name`

The cookie name.

`value`

The cookie value, which should be a string or `None`. If `value` is `None`, it's equivalent to calling the `webob.response.Response.unset_cookie()` method for this cookie key (it effectively deletes the cookie on the client).

`max_age`

An integer representing a number of seconds, `datetime.timedelta`, or `None`. This value is used as the Max-Age of the generated cookie. If `expires` is not passed and this value is not `None`, the `max_age` value will also influence the Expires value of the cookie (Expires will be set to now + `max_age`). If this value is `None`, the cookie will not have a Max-Age value (unless `expires` is set). If both `max_age` and `expires` are set, this value takes precedence.

`path`

A string representing the cookie Path value. It defaults to `/`.

`domain`

A string representing the cookie Domain, or `None`. If `domain` is `None`, no Domain value will be sent in the cookie.

`secure`

A boolean. If it's `True`, the `secure` flag will be sent in the cookie, if it's `False`, the `secure` flag will not be sent in the cookie.

`httponly`

A boolean. If it's `True`, the `HttpOnly` flag will be sent in the cookie, if it's `False`, the `HttpOnly` flag will not be sent in the cookie.

`comment`

A string representing the cookie Comment value, or `None`. If `comment` is `None`, no Comment value will be sent in the cookie.

`expires`

A `datetime.timedelta` object representing an amount of time, `datetime.datetime` or `None`. A non-`None` value is used to generate the Expires value of the generated cookie. If `max_age` is not passed, but this value is not `None`, it will influence the Max-Age header. If this value is `None`, the Expires cookie value will be unset (unless `max_age` is set). If `max_age` is set, it will be used to generate the `expires` and this value is ignored.

If a `datetime.datetime` is provided it has to either be timezone aware or be based on UTC. `datetime.datetime` objects that are local time are not supported. Timezone aware `datetime.datetime` objects are converted to UTC.

This argument will be removed in future versions of WebOb (version 1.9).

`overwrite`

If this key is `True`, before setting the cookie, unset any existing cookie.

status

The status string

status_code

The status as an integer

status_int

The status as an integer

text

Get/set the text value of the body using the charset of the Content-Type or the default_body_encoding.

ubody

Deprecated alias for `.text`

unicode_body

Deprecated alias for `.text`

unset_cookie (*name*, *strict=True*)

Unset a cookie with the given name (remove it from the response).

vary

Gets and sets the Vary header (HTTP spec section 14.44). Converts it using list.

www_authenticate

Gets and sets the WWW-Authenticate header (HTTP spec section 14.47). Converts it using `parse_auth` and `serialize_auth`.

class FileResponse (*path*, *request=None*, *cache_max_age=None*, *content_type=None*, *content_encoding=None*)

A Response object that can be used to serve a static file from disk simply.

`path` is a file path on disk.

`request` must be a Pyramid *request* object. Note that a request *must* be passed if the response is meant to attempt to use the `wsgi.file_wrapper` feature of the web server that you're using to serve your Pyramid application.

`cache_max_age` is the number of seconds that should be used to HTTP cache this response.

`content_type` is the `content_type` of the response.

`content_encoding` is the `content_encoding` of the response. It's generally safe to leave this set to `None` if you're serving a binary file. This argument will be ignored if you also leave `content-type` as `None`.

class FileIter (*file*, *block_size=262144*)

A fixed-block-size iterator for use as a WSGI app_iter.

file is a Python file pointer (or at least an object with a `read` method that takes a size hint).

block_size is an optional block size for iteration.

Functions

response_adapter (**types_or_ifaces*)

Decorator activated via a *scan* which treats the function being decorated as a *response adapter* for the set of types or interfaces passed as **types_or_ifaces* to the decorator constructor.

For example, if you scan the following response adapter:

```
from pyramid.response import Response
from pyramid.response import response_adapter

@response_adapter(int)
def myadapter(i):
    return Response(status=i)
```

You can then return an integer from your view callables, and it will be converted into a response with the integer as the status code.

More than one type or interface can be passed as a constructor argument. The decorated response adapter will be called for each type or interface.

```
import json

from pyramid.response import Response
from pyramid.response import response_adapter

@response_adapter(dict, list)
def myadapter(ob):
    return Response(json.dumps(ob))
```

This method will have no effect until a *scan* is performed against the package or module which contains it, ala:

```
from pyramid.config import Configurator
config = Configurator()
config.scan('somepackage_containing_adapters')
```

pyramid.scaffolds

class Template (*name*)

Inherit from this base class and override methods to use the Pyramid scaffolding system.

post (*command*, *output_dir*, *vars*)

Called after template is applied.

pre (*command*, *output_dir*, *vars*)

Called before template is applied.

render_template (*content*, *vars*, *filename=None*)

Return a bytestring representing a templated file based on the input (*content*) and the variable names defined (*vars*). *filename* is used for exception reporting.

template_dir ()

Return the template directory of the scaffold. By default, it returns the value of `os.path.join(self.module_dir(), self._template_dir)` (`self.module_dir()` returns the module in which your subclass has been defined). If `self._template_dir` is a tuple this method just returns the value instead of trying to construct a path. If `_template_dir` is a tuple, it should be a 2-element tuple: (`package_name`, `package_relative_path`).

class PyramidTemplate (*name*)

A class that can be used as a base class for Pyramid scaffolding templates.

post (*command*, *output_dir*, *vars*)

Overrides `pyramid.scaffolds.template.Template.post()`, to print “Welcome to Pyramid. Sorry for the convenience.” after a successful scaffolding rendering.

pre (*command*, *output_dir*, *vars*)

Overrides `pyramid.scaffolds.template.Template.pre()`, adding several variables to the default variables list (including `random_string`, and `package_logger`). It also prevents common misnamings (such as naming a package “site” or naming a package logger “root”).

pyramid.scripting

get_root (*app*, *request=None*)

Return a tuple composed of (*root*, *closer*) when provided a *router* instance as the *app* argument. The *root* returned is the application root object. The *closer* returned is a callable (accepting no arguments) that should be called when your scripting application is finished using the *root*.

request is passed to the Pyramid application root factory to compute the root. If *request* is *None*, a default will be constructed using the registry's *Request Factory* via the *pyramid.interfaces.IRequestFactory.blank()* method.

prepare (*request=None*, *registry=None*)

This function pushes data onto the Pyramid threadlocal stack (*request* and *registry*), making those objects 'current'. It returns a dictionary useful for bootstrapping a Pyramid application in a scripting environment.

request is passed to the Pyramid application root factory to compute the root. If *request* is *None*, a default will be constructed using the registry's *Request Factory* via the *pyramid.interfaces.IRequestFactory.blank()* method.

If *registry* is not supplied, the last registry loaded from *pyramid.config.global_registries* will be used. If you have loaded more than one Pyramid application in the current process, you may not want to use the last registry loaded, thus you can search the *global_registries* and supply the appropriate one based on your own criteria.

The function returns a dictionary composed of *root*, *closer*, *registry*, *request* and *root_factory*. The *root* returned is the application's root resource object. The *closer* returned is a callable (accepting no arguments) that should be called when your scripting application is finished using the *root*. *registry* is the registry object passed or the last registry loaded into *pyramid.config.global_registries* if no registry is passed. *request* is the request object passed or the constructed request if no request is passed. *root_factory* is the root factory used to construct the root.

pyramid.security

Authentication API Functions

authenticated_userid (*request*)

A function that returns the value of the property *pyramid.request.Request.authenticated_userid*.

Deprecated since version 1.5: Use *pyramid.request.Request.authenticated_userid* instead.

unauthenticated_userid(*request*)

A function that returns the value of the property `pyramid.request.Request.unauthenticated_userid`.

Deprecated since version 1.5: Use `pyramid.request.Request.unauthenticated_userid` instead.

effective_principals(*request*)

A function that returns the value of the property `pyramid.request.Request.effective_principals`.

Deprecated since version 1.5: Use `pyramid.request.Request.effective_principals` instead.

forget(*request*)

Return a sequence of header tuples (e.g. `[('Set-Cookie', 'foo=abc')]`) suitable for ‘forgetting’ the set of credentials possessed by the currently authenticated user. A common usage might look like so within the body of a view function (`response` is assumed to be an *WebOb* -style *response* object computed previously by the view code):

```
from pyramid.security import forget
headers = forget(request)
response.headerlist.extend(headers)
return response
```

If no *authentication policy* is in use, this function will always return an empty sequence.

remember(*request*, *userid*, ***kwargs*)

Returns a sequence of header tuples (e.g. `[('Set-Cookie', 'foo=abc')]`) on this request’s response. These headers are suitable for ‘remembering’ a set of credentials implied by the data passed as *userid* and **kw* using the current *authentication policy*. Common usage might look like so within the body of a view function (`response` is assumed to be a *WebOb* -style *response* object computed previously by the view code):

```
from pyramid.security import remember
headers = remember(request, 'chrism', password='123', max_age='86400')
response = request.response
response.headerlist.extend(headers)
return response
```

If no *authentication policy* is in use, this function will always return an empty sequence. If used, the composition and meaning of **kw* must be agreed upon by the calling code and the effective authentication policy.

Deprecated since version 1.6: Renamed the *principal* argument to *userid* to clarify its purpose.

Authorization API Functions

has_permission (*permission, context, request*)


A function that calls `pyramid.request.Request.has_permission()` and returns its result.

Deprecated since version 1.5: Use `pyramid.request.Request.has_permission()` instead.

Changed in version 1.5a3: If context is None, then attempt to use the context attribute of self; if not set, then the `AttributeError` is propagated.

principals_allowed_by_permission (*context, permission*)

Provided a *context* (a resource object), and a *permission* (a string or unicode object), if a *authorization policy* is in effect, return a sequence of *principal* ids that possess the permission in the *context*. If no authorization policy is in effect, this will return a sequence with the single value `pyramid.security.Everyone` (the special principal identifier representing all principals).

 even if an *authorization policy* is in effect, some (exotic) authorization policies may not implement the required machinery for this function; those will cause a `NotImplementedError` exception to be raised when this function is invoked.

view_execution_permitted (*context, request, name=''*)

If the view specified by *context* and *name* is protected by a *permission*, check the permission associated with the view using the effective authentication/authorization policies and the *request*. Return a boolean result. If no *authorization policy* is in effect, or if the view is not protected by a permission, return `True`. If no view can view found, an exception will be raised.

Changed in version 1.4a4: An exception is raised if no view is found.

Constants

Everyone

The special principal id named 'Everyone'. This principal id is granted to all requests. Its actual value is the string 'system.Everyone'.

Authenticated

The special principal id named 'Authenticated'. This principal id is granted to all requests which contain any other non-Everyone principal id (according to the *authentication policy*). Its actual value is the string 'system.Authenticated'.

ALL_PERMISSIONS

An object that can be used as the `permission` member of an ACE which matches all permissions unconditionally. For example, an ACE that uses `ALL_PERMISSIONS` might be composed like so: `('Deny', 'system.Everyone', ALL_PERMISSIONS)`.

DENY_ALL

A convenience shorthand ACE that defines `('Deny', 'system.Everyone', ALL_PERMISSIONS)`. This is often used as the last ACE in an ACL in systems that use an “inheriting” security policy, representing the concept “don’t inherit any other ACEs”.

NO_PERMISSION_REQUIRED

A special permission which indicates that the view should always be executable by entirely anonymous users, regardless of the default permission, bypassing any *authorization policy* that may be in effect. Its actual value is the string `'__no_permission_required__'`.

Return Values**Allow**

The ACE “action” (the first element in an ACE e.g. `(Allow, Everyone, 'read')`) that means allow access. A sequence of ACEs makes up an ACL. It is a string, and its actual value is “Allow”.

Deny

The ACE “action” (the first element in an ACE e.g. `(Deny, 'george', 'read')`) that means deny access. A sequence of ACEs makes up an ACL. It is a string, and its actual value is “Deny”.

class ACLDenied

An instance of `ACLDenied` represents that a security check made explicitly against ACL was denied. It evaluates equal to all boolean false types. It also has the following attributes: `acl`, `ace`, `permission`, `principals`, and `context`. These attributes indicate the security values involved in the request. Its `__str__` method prints a summary of these attributes for debugging purposes. The same summary is available as the `msg` attribute.

class ACLAllowed

An instance of `ACLLowed` represents that a security check made explicitly against ACL was allowed. It evaluates equal to all boolean true types. It also has the following attributes: `acl`, `ace`, `permission`, `principals`, and `context`. These attributes indicate the security values involved in the request. Its `__str__` method prints a summary of these attributes for debugging purposes. The same summary is available as the `msg` attribute.

class Denied

An instance of `Denied` is returned when a security-related API or other Pyramid code denies an action unrelated to an ACL check. It evaluates equal to all boolean false types. It has an attribute named `msg` describing the circumstances for the deny.

class Allowed

An instance of `Allowed` is returned when a security-related API or other Pyramid code allows an action unrelated to an ACL check. It evaluates equal to all boolean true types. It has an attribute named `msg` describing the circumstances for the allow.

pyramid.session**signed_serialize** (*data, secret*)

Serialize any pickleable structure (*data*) and sign it using the *secret* (must be a string). Return the serialization, which includes the signature as its first 40 bytes. The `signed_deserialize` method will deserialize such a value.

This function is useful for creating signed cookies. For example:

```
cookieval = signed_serialize({'a':1}, 'secret')
response.set_cookie('signed_cookie', cookieval)
```

signed_deserialize (*serialized, secret, hmac=<module 'hmac' from
'/home/docs/checkouts/readthedocs.org/user_builds/pyramid/envs/1.7-
branch/lib/python3.5/hmac.py'>*)

Deserialize the value returned from `signed_serialize`. If the value cannot be deserialized for any reason, a `ValueError` exception will be raised.

This function is useful for deserializing a signed cookie value created by `signed_serialize`. For example:

```
cookieval = request.cookies['signed_cookie']
data = signed_deserialize(cookieval, 'secret')
```

check_csrf_origin (*request, trusted_origins=None, raises=True*)

Check the Origin of the request to see if it is a cross site request or not.

If the value supplied by the Origin or Referer header isn't one of the trusted origins and *raises* is True, this function will raise a `pyramid.exceptions.BadCSRFOrigin` exception but if *raises* is False this function will return False instead. If the CSRF origin checks are successful this function will return True unconditionally.

Additional trusted origins may be added by passing a list of domain (and ports if nonstandard like `['example.com', 'dev.example.com:8080']`) in with the `trusted_origins` parameter. If `trusted_origins` is `None` (the default) this list of additional domains will be pulled from the `pyramid.csrf_trusted_origins` setting.

Note that this function will do nothing if `request.scheme` is not `https`.

New in version 1.7.

check_csrf_token(*request*, *token*='csrf_token', *header*='X-CSRF-Token', *raises*=True)

Check the CSRF token in the request's session against the value in `request.POST.get(token)` (if a POST request) or `request.headers.get(header)`. If a token keyword is not supplied to this function, the string `csrf_token` will be used to look up the token in `request.POST`. If a header keyword is not supplied to this function, the string `X-CSRF-Token` will be used to look up the token in `request.headers`.

If the value supplied by post or by header doesn't match the value supplied by `request.session.get_csrf_token()`, and `raises` is `True`, this function will raise an `pyramid.exceptions.BadCSRFToken` exception. If the values differ and `raises` is `False`, this function will return `False`. If the CSRF check is successful, this function will return `True` unconditionally.

Note that using this function requires that a *session factory* is configured.

See *Checking CSRF Tokens Automatically* for information about how to secure your application automatically against CSRF attacks.

New in version 1.4a2.

Changed in version 1.7a1: A CSRF token passed in the query string of the request is no longer considered valid. It must be passed in either the request body or a header.

SignedCookieSessionFactory(*secret*, *cookie_name*='session', *max_age*=None, *path*='/', *domain*=None, *secure*=False, *httponly*=False, *set_on_exception*=True, *timeout*=1200, *reissue_time*=0, *hashalg*='sha512', *salt*='pyramid.session.', *serializer*=None)

New in version 1.5.

Configure a *session factory* which will provide signed cookie-based sessions. The return value of this function is a *session factory*, which may be provided as the `session_factory` argument of a `pyramid.config.Configurator` constructor, or used as the `session_factory` argument of the `pyramid.config.Configurator.set_session_factory()` method.

The session factory returned by this function will create sessions which are limited to storing fewer than 4000 bytes of data (as the payload must fit into a single cookie).

Parameters:

secret A string which is used to sign the cookie. The secret should be at least as long as the block size of the selected hash algorithm. For `sha512` this would mean a 128 bit (64 character) secret. It should be unique within the set of secret values provided to Pyramid for its various subsystems (see *Admonishment Against Secret-Sharing*).

hashalg The HMAC digest algorithm to use for signing. The algorithm must be supported by the `hashlib` library. Default: `'sha512'`.

salt A namespace to avoid collisions between different uses of a shared secret. Reusing a secret for different parts of an application is strongly discouraged (see *Admonishment Against Secret-Sharing*). Default: `'pyramid.session.'`.

cookie_name The name of the cookie used for sessioning. Default: `'session'`.

max_age The maximum age of the cookie used for sessioning (in seconds). Default: `None` (browser scope).

path The path used for the session cookie. Default: `'/'`.

domain The domain used for the session cookie. Default: `None` (no domain).

secure The ‘secure’ flag of the session cookie. Default: `False`.

httponly Hide the cookie from Javascript by setting the ‘HttpOnly’ flag of the session cookie. Default: `False`.

timeout A number of seconds of inactivity before a session times out. If `None` then the cookie never expires. This lifetime only applies to the *value* within the cookie. Meaning that if the cookie expires due to a lower `max_age`, then this setting has no effect. Default: `1200`.

reissue_time The number of seconds that must pass before the cookie is automatically reissued as the result of accessing the session. The duration is measured as the number of seconds since the last session cookie was issued and ‘now’. If this value is 0, a new cookie will be reissued on every request accessing the session. If `None` then the cookie’s lifetime will never be extended.

A good rule of thumb: if you want auto-expired cookies based on inactivity: set the `timeout` value to 1200 (20 mins) and set the `reissue_time` value to perhaps a tenth of the `timeout` value (120 or 2 mins). It’s nonsensical to set the `timeout` value lower than the `reissue_time` value, as the ticket will never be reissued. However, such a configuration is not explicitly prevented.

Default: 0.

set_on_exception If True, set a session cookie even if an exception occurs while rendering a view. Default: True.

serializer An object with two methods: `loads` and `dumps`. The `loads` method should accept bytes and return a Python object. The `dumps` method should accept a Python object and return bytes. A `ValueError` should be raised for malformed inputs. If a serializer is not passed, the `pyramid.session.PickleSerializer` serializer will be used.

UnencryptedCookieSessionFactoryConfig (*secret*, *timeout=1200*,
cookie_name='session',
cookie_max_age=None,
cookie_path='/', *cookie_domain=None*,
cookie_secure=False,
cookie_httponly=False,
cookie_on_exception=True,
signed_serialize=<function
signed_serialize>,
signed_deserialize=<function
signed_deserialize>)

Deprecated since version 1.5: Use `pyramid.session.SignedCookieSessionFactory()` instead. Caveat: Cookies generated using `SignedCookieSessionFactory` are not compatible with cookies generated using `UnencryptedCookieSessionFactory`, so existing user session data will be destroyed if you switch to it.

Configure a *session factory* which will provide unencrypted (but signed) cookie-based sessions. The return value of this function is a *session factory*, which may be provided as the `session_factory` argument of a `pyramid.config.Configurator` constructor, or used as the `session_factory` argument of the `pyramid.config.Configurator.set_session_factory()` method.

The session factory returned by this function will create sessions which are limited to storing fewer than 4000 bytes of data (as the payload must fit into a single cookie).

Parameters:

secret A string which is used to sign the cookie.

timeout A number of seconds of inactivity before a session times out.

cookie_name The name of the cookie used for sessioning.

cookie_max_age The maximum age of the cookie used for sessioning (in seconds). Default: None (browser scope).

cookie_path The path used for the session cookie.

cookie_domain The domain used for the session cookie. Default: `None` (no domain).

cookie_secure The ‘secure’ flag of the session cookie.

cookie_httponly The ‘httpOnly’ flag of the session cookie.

cookie_on_exception If `True`, set a session cookie even if an exception occurs while rendering a view.

signed_serialize A callable which takes more or less arbitrary Python data structure and a secret and returns a signed serialization in bytes. Default: `signed_serialize` (using `pickle`).

signed_deserialize A callable which takes a signed and serialized data structure in bytes and a secret and returns the original data structure if the signature is valid. Default: `signed_deserialize` (using `pickle`).

BaseCookieSessionFactory (*serializer*, *cookie_name*='session', *max_age*=None, *path*='/', *domain*=None, *secure*=False, *httponly*=False, *timeout*=1200, *reissue_time*=0, *set_on_exception*=True)

New in version 1.5.

Configure a *session factory* which will provide cookie-based sessions. The return value of this function is a *session factory*, which may be provided as the `session_factory` argument of a `pyramid.config.Configurator` constructor, or used as the `session_factory` argument of the `pyramid.config.Configurator.set_session_factory()` method.

The session factory returned by this function will create sessions which are limited to storing fewer than 4000 bytes of data (as the payload must fit into a single cookie).

Parameters:

serializer An object with two methods: `loads` and `dumps`. The `loads` method should accept bytes and return a Python object. The `dumps` method should accept a Python object and return bytes. A `ValueError` should be raised for malformed inputs.

cookie_name The name of the cookie used for sessioning. Default: `'session'`.

max_age The maximum age of the cookie used for sessioning (in seconds). Default: `None` (browser scope).

path The path used for the session cookie. Default: `'/'`.

domain The domain used for the session cookie. Default: `None` (no domain).

secure The ‘secure’ flag of the session cookie. Default: `False`.

httponly Hide the cookie from Javascript by setting the ‘HttpOnly’ flag of the session cookie. Default: `False`.

timeout A number of seconds of inactivity before a session times out. If `None` then the cookie never expires. This lifetime only applies to the *value* within the cookie. Meaning that if the cookie expires due to a lower `max_age`, then this setting has no effect. Default: `1200`.

reissue_time The number of seconds that must pass before the cookie is automatically reissued as the result of a request which accesses the session. The duration is measured as the number of seconds since the last session cookie was issued and ‘now’. If this value is `0`, a new cookie will be reissued on every request accessing the session. If `None` then the cookie’s lifetime will never be extended.

A good rule of thumb: if you want auto-expired cookies based on inactivity: set the `timeout` value to `1200` (20 mins) and set the `reissue_time` value to perhaps a tenth of the `timeout` value (`120` or `2` mins). It’s nonsensical to set the `timeout` value lower than the `reissue_time` value, as the ticket will never be reissued. However, such a configuration is not explicitly prevented.

Default: `0`.

set_on_exception If `True`, set a session cookie even if an exception occurs while rendering a view. Default: `True`.

class PickleSerializer (*protocol=4*)

A serializer that uses the pickle protocol to dump Python data to bytes.

This is the default serializer used by Pyramid.

`protocol` may be specified to control the version of pickle used. Defaults to `pickle.HIGHEST_PROTOCOL`.

pyramid.settings

asbool (*s*)

Return the boolean value `True` if the case-lowered value of string input *s* is a *truthy string*. If *s* is already one of the boolean values `True` or `False`, return it.

aslist (*value, flatten=True*)

Return a list of strings, separating the input based on newlines and, if `flatten=True` (the default), also split on spaces within each line.

pyramid.static

```
class static_view(root_dir,          cache_max_age=3600,          package_name=None,
                  use_subpath=False, index='index.html', cachebust_match=None)
```

An instance of this class is a callable which can act as a Pyramid *view callable*; this view will serve static files from a directory on disk based on the `root_dir` you provide to its constructor.

The directory may contain subdirectories (recursively); the static view implementation will descend into these directories as necessary based on the components of the URL in order to resolve a path into a response.

You may pass an absolute or relative filesystem path or a *asset specification* representing the directory containing static files as the `root_dir` argument to this class' constructor.

If the `root_dir` path is relative, and the `package_name` argument is `None`, `root_dir` will be considered relative to the directory in which the Python file which *calls* `static` resides. If the `package_name` name argument is provided, and a relative `root_dir` is provided, the `root_dir` will be considered relative to the Python *package* specified by `package_name` (a dotted path to a Python package).

`cache_max_age` influences the `Expires` and `Max-Age` response headers returned by the view (default is 3600 seconds or one hour).

`use_subpath` influences whether `request.subpath` will be used as `PATH_INFO` when calling the underlying WSGI application which actually serves the static files. If it is `True`, the static application will consider `request.subpath` as `PATH_INFO` input. If it is `False`, the static application will consider `request.environ[PATH_INFO]` as `PATH_INFO` input. By default, this is `False`.



If the `root_dir` is relative to a *package*, or is a *asset specification* the Pyramid `pyramid.config.Configurator` method can be used to override assets within the named `root_dir` package-relative directory. However, if the `root_dir` is absolute, configuration will not be able to override the assets it contains.

```
class ManifestCacheBuster(manifest_spec, reload=False)
```

An implementation of *ICacheBuster* which uses a supplied manifest file to map an asset path to a cache-busted version of the path.

The `manifest_spec` can be an absolute path or a *asset specification* pointing to a package-relative file.

The manifest file is expected to conform to the following simple JSON format:

```
{  
    "css/main.css": "css/main-678b7c80.css",  
    "images/background.png": "images/background-a8169106.png",  
}
```

By default, it is a JSON-serialized dictionary where the keys are the source asset paths used in calls to `static_url()`. For example:

```
>>> request.static_url('myapp:static/css/main.css')  
"http://www.example.com/static/css/main-678b7c80.css"
```

The file format and location can be changed by subclassing and overriding `parse_manifest()`.

If a path is not found in the manifest it will pass through unchanged.

If `reload` is `True` then the manifest file will be reloaded when changed. It is not recommended to leave this enabled in production.

If the manifest file cannot be found on disk it will be treated as an empty mapping unless `reload` is `False`.

New in version 1.6.

static_exists (*path*)

Test whether a path exists. Returns `False` for broken symbolic links

static_getmtime (*filename*)

Return the last modification time of a file, reported by `os.stat()`.

manifest

The current manifest dictionary.

parse_manifest (*content*)

Parse the `content` read from the `manifest_path` into a dictionary mapping.

Subclasses may override this method to use something other than `json.loads` to load any type of file format and return a conforming dictionary.

class `QueryStringCacheBuster` (*param='x'*)

An implementation of *ICacheBuster* which adds a token for cache busting in the query string of an asset URL.

The optional *param* argument determines the name of the parameter added to the query string and defaults to 'x'.

To use this class, subclass it and provide a *tokenize* method which accepts *request*, *pathspec*, *kw* and returns a token.

New in version 1.6.

class `QueryStringConstantCacheBuster` (*token, param='x'*)

An implementation of *ICacheBuster* which adds an arbitrary token for cache busting in the query string of an asset URL.

The *token* parameter is the token string to use for cache busting and will be the same for every request.

The optional *param* argument determines the name of the parameter added to the query string and defaults to 'x'.

New in version 1.6.

`pyramid.testing`

`setUp` (*registry=None, request=None, hook_zca=True, autocommit=True, settings=None, package=None*)

Set Pyramid registry and request thread locals for the duration of a single unit test.

Use this function in the *setUp* method of a unittest test case which directly or indirectly uses:

- any method of the *pyramid.config.Configurator* object returned by this function.
- the *pyramid.threadlocal.get_current_registry()* or *pyramid.threadlocal.get_current_request()* functions.

If you use the `get_current_*` functions (or call Pyramid code that uses these functions) without calling `setUp`, `pyramid.threadlocal.get_current_registry()` will return a *global application registry*, which may cause unit tests to not be isolated with respect to registrations they perform.

If the `registry` argument is `None`, a new empty *application registry* will be created (an instance of the `pyramid.registry.Registry` class). If the `registry` argument is not `None`, the value passed in should be an instance of the `pyramid.registry.Registry` class or a suitable testing analogue.

After `setUp` is finished, the registry returned by the `pyramid.threadlocal.get_current_registry()` function will be the passed (or constructed) registry until `pyramid.testing.tearDown()` is called (or `pyramid.testing.setUp()` is called again).

If the `hook_zca` argument is `True`, `setUp` will attempt to perform the operation `zope.component.getSiteManager.sethook(pyramid.threadlocal.get_current_registry())`, which will cause the *Zope Component Architecture* global API (e.g. `zope.component.getSiteManager()`, `zope.component.getAdapter()`, and so on) to use the registry constructed by `setUp` as the value it returns from `zope.component.getSiteManager()`. If the `zope.component` package cannot be imported, or if `hook_zca` is `False`, the hook will not be set.

If `settings` is not `None`, it must be a dictionary representing the values passed to a Configurator as its `settings=` argument.

If `package` is `None` it will be set to the caller's package. The `package` setting in the `pyramid.config.Configurator` will affect any relative imports made via `pyramid.config.Configurator.include()` or `pyramid.config.Configurator.maybe_dotted()`.

This function returns an instance of the `pyramid.config.Configurator` class, which can be used for further configuration to set up an environment suitable for a unit or integration test. The `registry` attribute attached to the Configurator instance represents the 'current' *application registry*; the same registry will be returned by `pyramid.threadlocal.get_current_registry()` during the execution of the test.

`tearDown(unhook_zca=True)`

Undo the effects of `pyramid.testing.setUp()`. Use this function in the `tearDown` method of a unit test that uses `pyramid.testing.setUp()` in its `setUp` method.

If the `unhook_zca` argument is `True` (the default), call `zope.component.getSiteManager.reset()`. This undoes the action of `pyramid.testing.setUp()` when called with the argument `hook_zca=True`. If `zope.component` cannot be imported, `unhook_zca` is set to `False`.

testConfig (*registry=None, request=None, hook_zca=True, autocommit=True, settings=None*)

Returns a context manager for test set up.

This context manager calls `pyramid.testing.setUp()` when entering and `pyramid.testing.tearDown()` when exiting.

All arguments are passed directly to `pyramid.testing.setUp()`. If the ZCA is hooked, it will always be un-hooked in `tearDown`.

This context manager allows you to write test code like this:

```
1 with testConfig() as config:
2     config.add_route('bar', '/bar/{id}')
3     req = DummyRequest()
4     resp = myview(req),
```

cleanUp (**arg, **kw*)

An alias for `pyramid.testing.setUp()`.

class DummyResource (*__name__=None, __parent__=None, __provides__=None, **kw*)

A dummy Pyramid *resource* object.

clone (*__name__=<object object>, __parent__=<object object>, **kw*)

Create a clone of the resource object. If `__name__` or `__parent__` arguments are passed, use these values to override the existing `__name__` or `__parent__` of the resource. If any extra keyword args are passed in via the `kw` argument, use these keywords to add to or override existing resource keywords (attributes).

items ()

Return the items set by `__setitem__`

keys ()

Return the keys set by `__setitem__`

values ()

Return the values set by `__setitem__`

class DummyRequest (*params=None, environ=None, headers=None, path='/', cookies=None, post=None, **kw*)

A `DummyRequest` object (incompletely) imitates a *request* object.

The `params`, `environ`, `headers`, `path`, and `cookies` arguments correspond to their *WebOb* equivalents.

The `post` argument, if passed, populates the request's `POST` attribute, but *not* `params`, in order to allow testing that the app accepts data for a given view only from `POST` requests. This argument also sets `self.method` to `"POST"`.

Extra keyword arguments are assigned as attributes of the request itself.

Note that `DummyRequest` does not have complete fidelity with a “real” request. For example, by default, the `DummyRequest` `GET` and `POST` attributes are of type `dict`, unlike a normal `Request`'s `GET` and `POST`, which are of type `MultiDict`. If your code uses the features of `MultiDict`, you should either use a real `pyramid.request.Request` or adapt your `DummyRequest` by replacing the attributes with `MultiDict` instances.

Other similar incompatibilities exist. If you need all the features of a `Request`, use the `pyramid.request.Request` class itself rather than this class while writing tests.

`request_iface = <InterfaceClass pyramid.interfaces.IRequest>`

`class DummyTemplateRenderer (string_response='')`

An instance of this class is returned from `pyramid.config.Configurator.testing_add_renderer()`. It has a helper function (`assert_`) that makes it possible to make an assertion which compares data passed to the renderer by the view function against expected key/value pairs.

`assert_ (kw)`**

Accept an arbitrary set of assertion key/value pairs. For each assertion key/value pair assert that the renderer (eg. `pyramid.renderers.render_to_response()`) received the key with a value that equals the asserted value. If the renderer did not receive the key at all, or the value received by the renderer doesn't match the assertion value, raise an `AssertionError`.

`pyramid.threadlocal`

`get_current_request ()`

Return the currently active request or `None` if no request is currently active.

This function should be used *extremely sparingly*, usually only in unit testing code. It's almost always usually a mistake to use `get_current_request` outside a testing context because its usage makes it possible to write code that can be neither easily tested nor scripted.

`get_current_registry ()`

Return the currently active *application registry* or the global application registry if no request is currently active.

This function should be used *extremely sparingly*, usually only in unit testing code. It's almost always usually a mistake to use `get_current_registry` outside a testing context because its usage makes it possible to write code that can be neither easily tested nor scripted.

pyramid.traversal**find_interface** (*resource*, *class_or_interface*)

Return the first resource found in the *lineage* of *resource* which, a) if *class_or_interface* is a Python class object, is an instance of the class or any subclass of that class or b) if *class_or_interface* is a *interface*, provides the specified interface. Return *None* if no resource providing *interface_or_class* can be found in the lineage. The *resource* passed in *must* be *location-aware*.

find_resource (*resource*, *path*)

Given a resource object and a string or tuple representing a path (such as the return value of `pyramid.traversal.resource_path()` or `pyramid.traversal.resource_path_tuple()`), return a resource in this application's resource tree at the specified path. The resource passed in *must* be *location-aware*. If the path cannot be resolved (if the respective node in the resource tree does not exist), a `KeyError` will be raised.

This function is the logical inverse of `pyramid.traversal.resource_path()` and `pyramid.traversal.resource_path_tuple()`; it can resolve any path string or tuple generated by either of those functions.

Rules for passing a *string* as the *path* argument: if the first character in the path string is the `/` character, the path is considered absolute and the resource tree traversal will start at the root resource. If the first character of the path string is *not* the `/` character, the path is considered relative and resource tree traversal will begin at the resource object supplied to the function as the *resource* argument. If an empty string is passed as *path*, the *resource* passed in will be returned. Resource path strings must be escaped in the following manner: each Unicode path segment must be encoded as UTF-8 and as each path segment must escaped via Python's `urllib.quote`. For example, `/path/to%20the/La%20Pe%C3%B1a` (absolute) or `to%20the/La%20Pe%C3%B1a` (relative). The `pyramid.traversal.resource_path()` function generates strings which follow these rules (albeit only absolute ones).

Rules for passing *text* (Unicode) as the *path* argument are the same as those for a string. In particular, the text may not have any nonascii characters in it.

Rules for passing a *tuple* as the *path* argument: if the first element in the path tuple is the empty string (for example `(' ', 'a', 'b', 'c')`), the path is considered absolute and the resource tree traversal will start at the resource tree root object. If the first element in the path tuple is not the empty string (for example `('a', 'b', 'c')`), the path is considered relative and resource tree traversal will begin at the resource object supplied to the function as the *resource* argument. If an empty sequence is passed as *path*, the *resource* passed in itself will be returned. No URL-quoting or UTF-8-encoding of individual path segments within the tuple is required (each segment may be any string or unicode object representing a resource name). Resource path tuples

generated by `pyramid.traversal.resource_path_tuple()` can always be resolved by `find_resource`.

i For backwards compatibility purposes, this function can also be imported as `pyramid.traversal.find_model()`, although doing so will emit a deprecation warning.

find_root (*resource*)

Find the root node in the resource tree to which `resource` belongs. Note that `resource` should be *location-aware*. Note that the root resource is available in the request object by accessing the `request.root` attribute.

resource_path (*resource*, **elements*)

Return a string object representing the absolute physical path of the resource object based on its position in the resource tree, e.g `/foo/bar`. Any positional arguments passed in as `elements` will be appended as path segments to the end of the resource path. For instance, if the resource's path is `/foo/bar` and `elements` equals `('a', 'b')`, the returned string will be `/foo/bar/a/b`. The first character in the string will always be the `/` character (a leading `/` character in a path string represents that the path is absolute).

Resource path strings returned will be escaped in the following manner: each unicode path segment will be encoded as UTF-8 and each path segment will be escaped via Python's `urllib.quote`. For example, `/path/to%20the/La%20Pe%C3%B1a`.

This function is a logical inverse of `pyramid.traversal.find_resource`: it can be used to generate path references that can later be resolved via that function.

The `resource` passed in *must* be *location-aware*.

i Each segment in the path string returned will use the `__name__` attribute of the resource it represents within the resource tree. Each of these segments *should* be a unicode or string object (as per the contract of *location-awareness*). However, no conversion or safety checking of resource names is performed. For instance, if one of the resources in your tree has a `__name__` which (by error) is a dictionary, the `pyramid.traversal.resource_path()` function will attempt to append it to a string and it will cause a `pyramid.exceptions.URLDecodeError`.

i The *root* resource *must* have a `__name__` attribute with a value of either `None` or the empty string for paths to be generated properly. If the root resource has a non-null `__name__` attribute, its name will be prepended to the generated path rather than a single leading `/` character.

i For backwards compatibility purposes, this function can also be imported as `model_path`, although doing so will cause a deprecation warning to be emitted.

resource_path_tuple (*resource*, **elements*)

Return a tuple representing the absolute physical path of the `resource` object based on its position in a resource tree, e.g. ('', 'foo', 'bar'). Any positional arguments passed in as `elements` will be appended as elements in the tuple representing the resource path. For instance, if the resource's path is ('', 'foo', 'bar') and `elements` equals ('a', 'b'), the returned tuple will be ('', 'foo', 'bar', 'a', 'b'). The first element of this tuple will always be the empty string (a leading empty string element in a path tuple represents that the path is absolute).

This function is a logical inverse of `pyramid.traversal.find_resource()`: it can be used to generate path references that can later be resolved by that function.

The `resource` passed in *must* be *location-aware*.

i Each segment in the path tuple returned will equal the `__name__` attribute of the resource it represents within the resource tree. Each of these segments *should* be a unicode or string object (as per the contract of *location-awareness*). However, no conversion or safety checking of resource names is performed. For instance, if one of the resources in your tree has a `__name__` which (by error) is a dictionary, that dictionary will be placed in the path tuple; no warning or error will be given.

i The *root* resource *must* have a `__name__` attribute with a value of either `None` or the empty string for path tuples to be generated properly. If the root resource has a non-null `__name__` attribute, its name will be the first element in the generated path tuple rather than the empty string.

i For backwards compatibility purposes, this function can also be imported as `model_path_tuple`, although doing so will cause a deprecation warning to be emitted.

quote_path_segment (*segment*, *safe*='')

virtual_root (*resource*, *request*)

Provided any *resource* and a *request* object, return the resource object representing the *virtual root* of the current *request*. Using a virtual root in a *traversal* -based Pyramid application permits rooting, for example, the resource at the traversal path `/cms` at `http://example.com/` instead of rooting it at `http://example.com/cms/`.

If the *resource* passed in is a context obtained via *traversal*, and if the `HTTP_X_VHM_ROOT` key is in the WSGI environment, the value of this key will be treated as a ‘virtual root path’: the `pyramid.traversal.find_resource()` API will be used to find the virtual root resource using this path; if the resource is found, it will be returned. If the `HTTP_X_VHM_ROOT` key is not present in the WSGI environment, the physical *root* of the resource tree will be returned instead.

Virtual roots are not useful at all in applications that use *URL dispatch*. Contexts obtained via URL dispatch don’t really support being virtually rooted (each URL dispatch context is both its own physical and virtual root). However if this API is called with a *resource* argument which is a context obtained via URL dispatch, the resource passed in will be returned unconditionally.

traverse (*resource*, *path*)

Given a resource object as *resource* and a string or tuple representing a path as *path* (such as the return value of `pyramid.traversal.resource_path()` or `pyramid.traversal.resource_path_tuple()` or the value of `request.environ['PATH_INFO']`), return a dictionary with the keys `context`, `root`, `view_name`, `subpath`, `traversed`, `virtual_root`, and `virtual_root_path`.

A definition of each value in the returned dictionary:

- context**: The *context* (a *resource* object) found via traversal or url dispatch. If the *path* passed in is the empty string, the value of the *resource* argument passed to this function is returned.
- root**: The resource object at which *traversal* begins. If the *resource* passed in was found via url dispatch or if the *path* passed in was relative (non-absolute), the value of the *resource* argument passed to this function is returned.
- view_name**: The *view name* found during *traversal* or *url dispatch*; if the *resource* was found via traversal, this is usually a representation of the path segment which directly follows the path to the context in the path. The *view_name* will be a Unicode object or the empty string. The *view_name* will be the empty string if there is no element which follows the context path. An example: if the path passed is `/foo/bar`, and a resource object is found at `/foo` (but not at `/foo/bar`), the ‘view name’ will be `u'bar'`. If the *resource* was found via *urldispatch*, the *view_name* will be the name the route found was registered with.

- `subpath`: For a resource found via *traversal*, this is a sequence of path segments found in the `path` that follow the `view_name` (if any). Each of these items is a Unicode object. If no path segments follow the `view_name`, the `subpath` will be the empty sequence. An example: if the path passed is `/foo/bar/baz/buz`, and a resource object is found at `/foo` (but not `/foo/bar`), the ‘view name’ will be `u'bar'` and the *subpath* will be `[u'baz', u'buz']`. For a resource found via *url dispatch*, the `subpath` will be a sequence of values discerned from `*subpath` in the route pattern matched or the empty sequence.
- `traversed`: The sequence of path elements traversed from the root to find the `context` object during *traversal*. Each of these items is a Unicode object. If no path segments were traversed to find the `context` object (e.g. if the path provided is the empty string), the `traversed` value will be the empty sequence. If the resource is a resource found via *url dispatch*, `traversed` will be `None`.
- `virtual_root`: A resource object representing the ‘virtual’ root of the resource tree being traversed during *traversal*. See *Virtual Hosting* for a definition of the virtual root object. If no virtual hosting is in effect, and the `path` passed in was absolute, the `virtual_root` will be the *physical* root resource object (the object at which *traversal* begins). If the resource passed in was found via *URL dispatch* or if the `path` passed in was relative, the `virtual_root` will always equal the `root` object (the resource passed in).
- `virtual_root_path` – If *traversal* was used to find the resource, this will be the sequence of path elements traversed to find the `virtual_root` resource. Each of these items is a Unicode object. If no path segments were traversed to find the `virtual_root` resource (e.g. if virtual hosting is not in effect), the `traversed` value will be the empty list. If *url dispatch* was used to find the resource, this will be `None`.

If the path cannot be resolved, a `KeyError` will be raised.

Rules for passing a *string* as the `path` argument: if the first character in the path string is the `/` character, the path will be considered absolute and the resource tree traversal will start at the root resource. If the first character of the path string is *not* the `/` character, the path is considered relative and resource tree traversal will begin at the resource object supplied to the function as the `resource` argument. If an empty string is passed as `path`, the `resource` passed in will be returned. Resource path strings must be escaped in the following manner: each Unicode path segment must be encoded as UTF-8 and each path segment must be escaped via Python’s `urllib.quote`. For example, `/path/to%20the/La%20Pe%C3%B1a` (absolute) or `to%20the/La%20Pe%C3%B1a` (relative). The `pyramid.traversal.resource_path()` function generates strings which follow these rules (albeit only absolute ones).

Rules for passing a *tuple* as the `path` argument: if the first element in the path tuple is the empty string (for example `('', 'a', 'b', 'c')`), the path is considered absolute and the resource tree traversal will start at the resource tree root object. If the first element in the path tuple is not the empty string (for example `('a', 'b', 'c')`), the path is considered relative and resource tree

traversal will begin at the resource object supplied to the function as the `resource` argument. If an empty sequence is passed as `path`, the `resource` passed in itself will be returned. No URL-quoting or UTF-8-encoding of individual path segments within the tuple is required (each segment may be any string or unicode object representing a resource name).

Explanation of the conversion of `path` segment values to Unicode during traversal: Each segment is URL-unquoted, and decoded into Unicode. Each segment is assumed to be encoded using the UTF-8 encoding (or a subset, such as ASCII); a `pyramid.exceptions.URLDecodeError` is raised if a segment cannot be decoded. If a segment name is empty or if it is `.`, it is ignored. If a segment name is `..`, the previous segment is deleted, and the `..` is ignored. As a result of this process, the return values `view_name`, each element in the `subpath`, each element in `traversed`, and each element in the `virtual_root_path` will be Unicode as opposed to a string, and will be URL-decoded.

traversal_path (*path*)

Variant of `pyramid.traversal.traversal_path_info()` suitable for decoding paths that are URL-encoded.

If this function is passed a Unicode object instead of a sequence of bytes as `path`, that Unicode object *must* directly encodeable to ASCII. For example, `u'foo'` will work but `u'<unprintable unicode>'` (a Unicode object with characters that cannot be encoded to ascii) will not. A `UnicodeEncodeError` will be raised if the Unicode cannot be encoded directly to ASCII.

pyramid.tweens

excview_tween_factory (*handler, registry*)

A *tween* factory which produces a tween that catches an exception raised by downstream tweens (or the main Pyramid request handler) and, if possible, converts it into a Response using an *exception view*.

MAIN

Constant representing the main Pyramid handling function, for use in `under` and `over` arguments to `pyramid.config.Configurator.add_tween()`.

INGRESS

Constant representing the request ingress, for use in `under` and `over` arguments to `pyramid.config.Configurator.add_tween()`.

EXCVIEW

Constant representing the exception view tween, for use in `under` and `over` arguments to `pyramid.config.Configurator.add_tween()`.

pyramid.url

Utility functions for dealing with URLs in pyramid

resource_url (*context, request, *elements, query=None, anchor=None*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.resource_url(resource, *elements, **kw)
```

See `pyramid.request.Request.resource_url()` for more information.

route_url (*route_name, request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.route_url(route_name, *elements, **kw)
```

See `pyramid.request.Request.route_url()` for more information.

current_route_url (*request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.current_route_url(*elements, **kw)
```

See `pyramid.request.Request.current_route_url()` for more information.

route_path (*route_name, request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.route_path(route_name, *elements, **kw)
```

See `pyramid.request.Request.route_path()` for more information.

current_route_path (*request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.current_route_path(*elements, **kw)
```

See `pyramid.request.Request.current_route_path()` for more information.

static_url (*path, request, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.static_url(path, **kw)
```

See `pyramid.request.Request.static_url()` for more information.

static_path (*path*, *request*, ***kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.static_path(path, **kw)
```

See `pyramid.request.Request.static_path()` for more information.

urlencode (*query*, *doseq=True*)

An alternate implementation of Python's `stdlib urllib.urlencode` function which accepts unicode keys and values within the `query` dict/sequence; all Unicode keys and values are first converted to UTF-8 before being used to compose the query string.

The value of `query` must be a sequence of two-tuples representing key/value pairs *or* an object (often a dictionary) with an `.items()` method that returns a sequence of two-tuples representing key/value pairs.

For minimal calling convention backwards compatibility, this version of `urlencode` accepts *but ignores* a second argument conventionally named `doseq`. The Python `stdlib` version behaves differently when `doseq` is `False` and when a sequence is presented as one of the values. This version always behaves in the `doseq=True` mode, no matter what the value of the second argument.

See the Python `stdlib` documentation for `urllib.urlencode` for more information.

Changed in version 1.5: In a key/value pair, if the value is `None` then it will be dropped from the resulting output.

pyramid.view

render_view_to_response (*context*, *request*, *name=''*, *secure=True*)

Call the *view callable* configured with a *view configuration* that matches the *view name* `name` registered against the specified *context* and *request* and return a *response* object. This function will return `None` if a corresponding *view callable* cannot be found (when no *view configuration* matches the combination of `name / context / and request`).

If `secure` is `True`, and the *view callable* found is protected by a permission, the permission will be checked before calling the view function. If the permission check disallows view execution (based on the current *authorization policy*), a `pyramid.httpexceptions.HTTPForbidden` exception will be raised. The exception's `args` attribute explains why the view access was disallowed.

If `secure` is `False`, no permission checking is done.

render_view_to_iterable (*context*, *request*, *name*='', *secure*=True)

Call the *view callable* configured with a *view configuration* that matches the *view name* *name* registered against the specified *context* and *request* and return an iterable object which represents the body of a response. This function will return *None* if a corresponding *view callable* cannot be found (when no *view configuration* matches the combination of *name* / *context* / and *request*). Additionally, this function will raise a *ValueError* if a view function is found and called but the view function's result does not have an *app_iter* attribute.

You can usually get the bytestring representation of the return value of this function by calling `b''.join(iterable)`, or just use `pyramid.view.render_view()` instead.

If *secure* is *True*, and the view is protected by a permission, the permission will be checked before the view function is invoked. If the permission check disallows view execution (based on the current *authentication policy*), a `pyramid.httpexceptions.HTTPForbidden` exception will be raised; its *args* attribute explains why the view access was disallowed.

If *secure* is *False*, no permission checking is done.

render_view (*context*, *request*, *name*='', *secure*=True)

Call the *view callable* configured with a *view configuration* that matches the *view name* *name* registered against the specified *context* and *request* and unwind the view response's *app_iter* (see *View Callable Responses*) into a single bytestring. This function will return *None* if a corresponding *view callable* cannot be found (when no *view configuration* matches the combination of *name* / *context* / and *request*). Additionally, this function will raise a *ValueError* if a view function is found and called but the view function's result does not have an *app_iter* attribute. This function will return *None* if a corresponding view cannot be found.

If *secure* is *True*, and the view is protected by a permission, the permission will be checked before the view is invoked. If the permission check disallows view execution (based on the current *authorization policy*), a `pyramid.httpexceptions.HTTPForbidden` exception will be raised; its *args* attribute explains why the view access was disallowed.

If *secure* is *False*, no permission checking is done.

class view_config (***settings*)

A function, class or method *decorator* which allows a developer to create view registrations nearer to a *view callable* definition than use *imperative configuration* to do the same.

For example, this code in a module `views.py`:

```
from resources import MyResource

@view_config(name='my_view', context=MyResource, permission='read',
             route_name='site1')
def my_view(context, request):
    return 'OK'
```

Might replace the following call to the `pyramid.config.Configurator.add_view()` method:

```
import views
from resources import MyResource
config.add_view(views.my_view, context=MyResource, name='my_view',
                permission='read', route_name='site1')
```

`pyramid.view.view_config` supports the following keyword arguments: `context`, `permission`, `name`, `request_type`, `route_name`, `request_method`, `request_param`, `containment`, `xhr`, `accept`, `header`, `path_info`, `custom_predicates`, `decorator`, `mapper`, `http_cache`, `require_csrf`, `match_param`, `check_csrf`, `physical_path`, and `view_options`.

The meanings of these arguments are the same as the arguments passed to `pyramid.config.Configurator.add_view()`. If any argument is left out, its default will be the equivalent `add_view` default.

An additional keyword argument named `_depth` is provided for people who wish to reuse this class from another decorator. The default value is 0 and should be specified relative to the `view_config` invocation. It will be passed in to the *venusian* `attach` function as the depth of the callstack when Venusian checks if the decorator is being used in a class or module context. It's not often used, but it can be useful in this circumstance. See the `attach` function in Venusian for more information.

See also:

See also *Adding View Configuration Using the @view_config Decorator* for details about using `pyramid.view.view_config`.



`view_config` will work ONLY on module top level members because of the limitation of *venusian.Scanner.scan*.

class `view_defaults` (***settings*)

A class *decorator* which, when applied to a class, will provide defaults for all view configurations that use the class. This decorator accepts all the arguments accepted by `pyramid.view.view_config()`, and each has the same meaning.

See *@view_defaults Class Decorator* for more information.

class notfound_view_config (**settings)

New in version 1.3.

An analogue of `pyramid.view.view_config` which registers a *Not Found View*.

The `notfound_view_config` constructor accepts most of the same arguments as the constructor of `pyramid.view.view_config`. It can be used in the same places, and behaves in largely the same way, except it always registers a not found exception view instead of a ‘normal’ view.

Example:

```
from pyramid.view import notfound_view_config
from pyramid.response import Response

@notfound_view_config()
def notfound(request):
    return Response('Not found!', status='404 Not Found')
```

All arguments except `append_slash` have the same meaning as `pyramid.view.view_config()` and each predicate argument restricts the set of circumstances under which this notfound view will be invoked.

If `append_slash` is `True`, when the Not Found View is invoked, and the current path info does not end in a slash, the notfound logic will attempt to find a *route* that matches the request’s path info suffixed with a slash. If such a route exists, Pyramid will issue a redirect to the URL implied by the route; if it does not, Pyramid will return the result of the view callable provided as `view`, as normal.

If the argument provided as `append_slash` is not a boolean but instead implements *IResponse*, the `append_slash` logic will behave as if `append_slash=True` was passed, but the provided class will be used as the response class instead of the default `HTTPFound` response class when a redirect is performed. For example:

```
from pyramid.httpexceptions import (
    HTTPMovedPermanently,
    HTTPNotFound
)

@notfound_view_config(append_slash=HTTPMovedPermanently)
def aview(request):
    return HTTPNotFound('not found')
```

The above means that a redirect to a slash-appended route will be attempted, but instead of *HTTPFound* being used, *HTTPMovedPermanently* will be used for the redirect response if a slash-appended route is found.

Changed in version 1.6.

See *Changing the Not Found View* for detailed usage information.

class forbidden_view_config (**settings)

New in version 1.3.

An analogue of *pyramid.view.view_config* which registers a *forbidden* view.

The *forbidden_view_config* constructor accepts most of the same arguments as the constructor of *pyramid.view.view_config*. It can be used in the same places, and behaves in largely the same way, except it always registers a forbidden exception view instead of a ‘normal’ view.

Example:

```
from pyramid.view import forbidden_view_config
from pyramid.response import Response

@forbidden_view_config()
def forbidden(request):
    return Response('You are not allowed', status='403 Forbidden')
```

All arguments passed to this function have the same meaning as *pyramid.view.view_config()* and each predicate argument restricts the set of circumstances under which this notfound view will be invoked.

See *Changing the Forbidden View* for detailed usage information.

pyramid.viewderivers

INGRESS

Constant representing the request ingress, for use in under arguments to *pyramid.config.Configurator.add_view_deriver()*.

VIEW

Constant representing the *view callable* at the end of the view pipeline, for use in over arguments to *pyramid.config.Configurator.add_view_deriver()*.

pyramid.wsgi**wsgiapp** (*wrapped*)

Decorator to turn a WSGI application into a Pyramid *view callable*. This decorator differs from the `pyramid.wsgi.wsgiapp2()` decorator inasmuch as fixups of `PATH_INFO` and `SCRIPT_NAME` within the WSGI environment *are not* performed before the application is invoked.

E.g., the following in a `views.py` module:

```
@wsgiapp
def hello_world(environ, start_response):
    body = 'Hello world'
    start_response('200 OK', [ ('Content-Type', 'text/plain'),
                               ('Content-Length', len(body)) ] )
    return [body]
```

Allows the following call to `pyramid.config.Configurator.add_view()`:

```
from views import hello_world
config.add_view(hello_world, name='hello_world.txt')
```

The `wsgiapp` decorator will convert the result of the WSGI application to a *Response* and return it to Pyramid as if the WSGI app were a Pyramid view.

wsgiapp2 (*wrapped*)

Decorator to turn a WSGI application into a Pyramid view callable. This decorator differs from the `pyramid.wsgi.wsgiapp()` decorator inasmuch as fixups of `PATH_INFO` and `SCRIPT_NAME` within the WSGI environment *are* performed before the application is invoked.

E.g. the following in a `views.py` module:

```
@wsgiapp2
def hello_world(environ, start_response):
    body = 'Hello world'
    start_response('200 OK', [ ('Content-Type', 'text/plain'),
                               ('Content-Length', len(body)) ] )
    return [body]
```

Allows the following call to `pyramid.config.Configurator.add_view()`:

```
from views import hello_world
config.add_view(hello_world, name='hello_world.txt')
```

The `wsgiapp2` decorator will convert the result of the WSGI application to a Response and return it to Pyramid as if the WSGI app were a Pyramid view. The `SCRIPT_NAME` and `PATH_INFO` values present in the WSGI environment are fixed up before the application is invoked. In particular, a new WSGI environment is generated, and the *subpath* of the request passed to `wsgiapp2` is used as the new request's `PATH_INFO` and everything preceding the subpath is used as the `SCRIPT_NAME`. The new environment is passed to the downstream WSGI application.

p* Scripts Documentation

p* Scripts Documentation

Command line programs (p* scripts) included with Pyramid.

pcreate

```
$ pcreate --help
Usage: pcreate [options] -s <scaffold> output_directory

Render Pyramid scaffolding to an output directory

Options:
  -h, --help                show this help message and exit
  -s SCAFFOLD_NAME, --scaffold=SCAFFOLD_NAME
                           Add a scaffold to the create process (multiple -s
↳args
                           accepted)
  -t SCAFFOLD_NAME, --template=SCAFFOLD_NAME
                           A backwards compatibility alias for -s/--scaffold.
                           Add a scaffold to the create process (multiple -t
↳args
                           accepted)
  -l, --list                List all available scaffold names
  --list-templates          A backwards compatibility alias for -l/--list.
↳List
                           all available scaffold names.
```

<code>--simulate</code>	Simulate but do no work
<code>--overwrite</code>	Always overwrite
<code>--interactive</code>	When a file would be overwritten, interrogate
<code>--ignore-conflicting-name</code>	Do create a project even if the chosen name is the name of an already existing / importable package.

See also:

Creating the Project

pdistreport

```
$ pdistreport --help
Usage: pdistreport

Show Python distribution versions and locations in use

Options:
  -h, --help  show this help message and exit
```

See also:

Showing All Installed Distributions and Their Versions

prequest

```
$ prequest --help
Usage: prequest config_uri path_info [args/options]

Submit a HTTP request to a web application. This command makes an
↳artificial
request to a web application that uses a PasteDeploy (.ini) configuration
↳file
for the server and application. Use "prequest config.ini /path" to request
"/path". Use "prequest --method=POST config.ini /path < data" to do a POST
with the given request body. Use "prequest --method=PUT config.ini /path <
data" to do a PUT with the given request body. Use "prequest --
↳method=PATCH
config.ini /path < data" to do a PATCH with the given request body. Use
```

```
"prequest --method=OPTIONS config.ini /path" to do an OPTIONS request. Use
"prequest --method=PROPFIND config.ini /path" to do a PROPFIND request. If
the path is relative (doesn't begin with "/" ) it is interpreted as
↳relative to
"/". The path passed to this script should be URL-quoted. The path can be
succeeded with a query string (e.g. `/path?a=1&b=2'). The variable
"environ['paste.command_request']" will be set to "True" in the request's
↳WSGI
environment, so your application can distinguish these calls from normal
requests.

Options:
  -h, --help                show this help message and exit
  -n NAME, --app-name=NAME  Load the named application from the config file
                           (default 'main')
  --header=NAME:VALUE       Header to add to request (you can use this option
                           multiple times)
  -d, --display-headers     Display status and headers before the response body
  -m METHOD, --method=METHOD
                           Request method type (GET, POST, PUT, PATCH, DELETE,
                           PROPFIND, OPTIONS)
  -l LOGIN, --login=LOGIN   HTTP basic auth username:password pair
```

See also:*Invoking a Request***proutes**

```
$ proutes --help
Usage: proutes config_uri
```

Print all URL dispatch routes used by a Pyramid application in the order in which they are evaluated. Each route includes the name of the route, the pattern of the route, and the view callable which will be invoked when the route is matched. This command accepts one positional argument named 'config_uri'. It specifies the PasteDeploy config file to use for the interactive shell. The format is 'inifile#name'. If the name is left off, 'main' will be assumed. Example: 'proutes myapp.ini'.


```
Options:
-h, --help            show this help message and exit
-g GLOB, --glob=GLOB  Display routes matching glob pattern
-f FORMAT, --format=FORMAT
                        Choose which columns to display, this will override
                        the format key in the [proutes] ini section
```

See also:

Displaying All Application Routes

pserve

```
$ pserve --help
Usage: pserve config_uri [start|stop|restart|status] [var=value]

This command serves a web application that uses a PasteDeploy configuration
file for the server and application. If start/stop/restart is given, then
--daemon is implied, and it will start (normal operation), stop (--stop-
daemon), or do both. Note: Daemonization features are deprecated. You can
also include variable assignments like 'http_port=8080' and then use
%(http_port)s in your config files.

Options:
-h, --help            show this help message and exit
-n NAME, --app-name=NAME
                        Load the named application (default main)
-s SERVER_TYPE, --server=SERVER_TYPE
                        Use the named server.
--server-name=SECTION_NAME
                        Use the named server as defined in the
↳ configuration
                        file (default: main)
--daemon              Run in daemon (background) mode [DEPRECATED]
--pid-file=FILENAME  Save PID to file (default to pyramid.pid if
↳ running in
                        daemon mode) [DEPRECATED]
--log-file=LOG_FILE  Save output to the given log file (redirects
↳ stdout)
                        [DEPRECATED]
--reload              Use auto-restart file monitor
--reload-interval=RELOAD_INTERVAL
                        Seconds between checking files (low number can
↳ cause
```

	significant CPU usage)
--monitor-restart	Auto-restart server if it dies [DEPRECATED]
-b, --browser	Open a web browser to server url
--status	Show the status of the (presumably daemonized) <u></u>
→server	
	[DEPRECATED]
-v, --verbose	Set verbose level (default 1)
-q, --quiet	Suppress verbose output
--user=USERNAME	Set the user (usually only possible when run as <u></u>
→root)	
--group=GROUP	Set the group (usually only possible when run as <u></u>
→root)	
--stop-daemon	Stop a daemonized server (given a PID file, or <u></u>
→default	
	pyramid.pid file) [DEPRECATED]

See also:*Running the Project Application***pshell**

```
$ pshell --help
Usage: pshell config_uri

Open an interactive shell with a Pyramid app loaded.  This command accepts 
→one
positional argument named "config_uri" which specifies the PasteDeploy 
→config
file to use for the interactive shell. The format is "inifile#name". If the
name is left off, the Pyramid default application will be assumed. 
→Example:
"pshell myapp.ini#main"  If you do not point the loader directly at the
section of the ini file containing your Pyramid application, the command 
→will
attempt to find the app for you. If you are loading a pipeline that 
→contains
more than one Pyramid application within it, the loader will use the last 
→one.

Options:
  -h, --help            show this help message and exit
  -p PYTHON_SHELL, --python-shell=PYTHON_SHELL
```

	Select the shell to use. A list of possible shells.
→ is	
	available using the --list-shells option.
-l, --list-shells	List all available shells.
--setup=SETUP	A callable that will be passed the environment.
→ before	
	it is made available to the shell. This option will
	override the 'setup' key in the [pshell] ini.
→ section.	

See also:

The Interactive Shell

ptweens

```
$ ptweens --help
Usage: ptweens config_uri

Print all implicit and explicit tween objects used by a Pyramid
→ application.
The handler output includes whether the system is using an explicit tweens
ordering (will be true when the "pyramid.tweens" deployment setting is
→ used)
or an implicit tweens ordering (will be true when the "pyramid.tweens"
deployment setting is *not* used). This command accepts one positional
argument named "config_uri" which specifies the PasteDeploy config file to
→ use
for the interactive shell. The format is "infile#name". If the name is
→ left
off, "main" will be assumed. Example: "ptweens myapp.ini#main".

Options:
-h, --help show this help message and exit
```

See also:

Displaying “Tweens”

pviews

```
$ pviews --help
Usage: pviews config_uri url

Print, for a given URL, the views that might match. Underneath each
potentially matching route, list the predicates required. Underneath each
route+predicate set, print each view that might match and its predicates.
This command accepts two positional arguments: 'config_uri' specifies the
PasteDeploy config file to use for the interactive shell. The format is
'inifile#name'. If the name is left off, 'main' will be assumed. 'url'
specifies the path info portion of a URL that will be used to find matching
views. Example: 'proutes myapp.ini#main /url'

Options:
  -h, --help  show this help message and exit
```

See also:

Displaying Matching Views for a Given URL

Change History

What's New in Pyramid 1.7

This article explains the new features in Pyramid version 1.7 as compared to its predecessor, Pyramid 1.6. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.7, as well as software dependency changes and notable documentation additions.

Bug Fix Releases

Pyramid 1.7 was released on 2016-05-19.

The following bug fix releases were made since then. Bug fix releases also include documentation improvements and other minor feature changes.

- *1.7.1 (2016-08-16)*
- *1.7.2 (2016-08-16)*
- *1.7.3 (2016-08-17)*
- *1.7.4 (2017-01-24)*
- *1.7.5 (2017-03-12)*
- *1.7.6 (2017-06-11)*

Backwards Incompatibilities

- The default hash algorithm for `pyramid.authentication.AuthTktAuthenticationPolicy` has changed from `md5` to `sha512`. If you are using the authentication policy and need to continue using `md5`, please explicitly set `hashalg='md5'`.

If you are not currently specifying the `hashalg` option in your apps, then this change means any existing auth tickets (and associated cookies) will no longer be valid, users will be logged out, and have to login to their accounts again.

This change has been issuing a `DeprecationWarning` since Pyramid 1.4.

See <https://github.com/Pylons/pyramid/pull/2496>

- Python 2.6 and 3.2 are no longer supported by Pyramid. See <https://github.com/Pylons/pyramid/issues/2368> and <https://github.com/Pylons/pyramid/pull/2256>
- The `pyramid.session.check_csrf_token()` function no longer validates a csrf token in the query string of a request. Only headers and request bodies are supported. See <https://github.com/Pylons/pyramid/pull/2500>
- A global permission set via `pyramid.config.Configurator.set_default_permission()` will no longer affect exception views. A permission must be set explicitly on the view for it to be enforced. See <https://github.com/Pylons/pyramid/pull/2534>

Feature Additions

- A new *View Derivers* concept has been added to Pyramid to allow framework authors to inject elements into the standard Pyramid view pipeline and affect all views in an application. This is similar to a decorator except that it has access to options passed to `config.add_view` and can affect other stages of the pipeline such as the raw response from a view or prior to security checks. See <https://github.com/Pylons/pyramid/pull/2021>
- Added a `require_csrf` view option which will enforce CSRF checks on requests with an unsafe method as defined by RFC2616. If the CSRF check fails a `BadCSRFToken` exception will be raised and may be caught by exception views (the default response is a 400 Bad Request). This option should be used in place of the deprecated `check_csrf` view predicate which would normally result in unexpected 404 Not Found response to the client instead of a catchable exception. See *Checking CSRF Tokens Automatically*, <https://github.com/Pylons/pyramid/pull/2413> and <https://github.com/Pylons/pyramid/pull/2500>

- Added a new method, `pyramid.config.Configurator.set_csrf_default_options()`, for configuring CSRF checks used by the `require_csrf=True` view option. This method can be used to turn on CSRF checks globally for every view in the application. This should be considered a good default for websites built on Pyramid. It is possible to opt-out of CSRF checks on a per-view basis by setting `require_csrf=False` on those views. See *Checking CSRF Tokens Automatically* and <https://github.com/Pylons/pyramid/pull/2413> and <https://github.com/Pylons/pyramid/pull/2518>
- Added an additional CSRF validation that checks the origin/referrer of a request and makes sure it matches the current `request.domain`. This particular check is only active when accessing a site over HTTPS as otherwise browsers don't always send the required information. If this additional CSRF validation fails a `BadCSRFOrigin` exception will be raised and may be caught by exception views (the default response is 400 Bad Request). Additional allowed origins may be configured by setting `pyramid.csrf_trusted_origins` to a list of domain names (with ports if on a non standard port) to allow. Subdomains are not allowed unless the domain name has been prefixed with a `..` See <https://github.com/Pylons/pyramid/pull/2501>
- Added a new `pyramid.session.check_csrf_origin()` API for validating the origin or referrer headers against the request's domain. See <https://github.com/Pylons/pyramid/pull/2501>
- Subclasses of `pyramid.httpexceptions.HTTPException` will now take into account the best match for the clients Accept header, and depending on what is requested will return `text/html`, `application/json` or `text/plain`. The default for `*/*` is still `text/html`, but if `application/json` is explicitly mentioned it will now receive a valid JSON response. See <https://github.com/Pylons/pyramid/pull/2489>
- A new event, `pyramid.events.BeforeTraversal`, and interface `pyramid.interfaces.IBeforeTraversal` have been introduced that will notify listeners before traversal starts in the router. See *Request Processing* as well as <https://github.com/Pylons/pyramid/pull/2469> and <https://github.com/Pylons/pyramid/pull/1876>
- A new method, `pyramid.request.Request.invoke_exception_view()`, which can be used to invoke an exception view and get back a response. This is useful for rendering an exception view outside of the context of the `EXCVIEW` tween where you may need more control over the request. See <https://github.com/Pylons/pyramid/pull/2393>
- A global permission set via `pyramid.config.Configurator.set_default_permission()` will no longer affect exception views. A permission must be set explicitly on the view for it to be enforced. See <https://github.com/Pylons/pyramid/pull/2534>
- Allow a leading `=` on the key of the request param predicate. For example, `'=abc=1'` is equivalent down to `request.params['=abc'] == '1'`. See <https://github.com/Pylons/pyramid/pull/1370>

- Allow using variable substitutions like `% (LOGGING_LOGGER_ROOT_LEVEL) s` for logging sections of the `.ini` file and populate these variables from the `pserve` command line – e.g.:

```
pserve development.ini LOGGING_LOGGER_ROOT_LEVEL=DEBUG
```

This support is thanks to the new `global_conf` option on `pyramid.paster.setup_logging()`. See <https://github.com/Pylons/pyramid/pull/2399>

- The `pyramid.tweens.EXCVIEW` tween will now re-raise the original exception if no exception view could be found to handle it. This allows the exception to be handled upstream by another tween or middleware. See <https://github.com/Pylons/pyramid/pull/2567>

Deprecations

- The `check_csrf` view predicate has been deprecated. Use the new `require_csrf` option or the `pyramid.require_default_csrf` setting to ensure that the `pyramid.exceptions.BadCSRFToken` exception is raised. See <https://github.com/Pylons/pyramid/pull/2413>
- Support for Python 3.3 will be removed in Pyramid 1.8. <https://github.com/Pylons/pyramid/issues/2477>

Scaffolding Enhancements

- A complete overhaul of the `alchemy` scaffold to show more modern best practices with regards to SQLAlchemy session management, as well as a more modular approach to configuration, separating routes into a separate module to illustrate uses of `pyramid.config.Configurator.include()`. See <https://github.com/Pylons/pyramid/pull/2024>

Documentation Enhancements

A massive overhaul of the packaging and tools used in the documentation was completed in <https://github.com/Pylons/pyramid/pull/2468>. A summary follows:

- All docs now recommend using `pip` instead of `easy_install`.
- The installation docs now expect the user to be using Python 3.4 or greater with access to the `python3 -m venv` tool to create virtual environments.

- Tutorials now use `py.test` and `pytest-cov` instead of `nose` and `coverage`.
- Further updates to the scaffolds as well as tutorials and their `src` files.

Along with the overhaul of the `alchemy` scaffold came a total overhaul of the *SQLAlchemy + URL dispatch wiki tutorial* to introduce more modern features into the usage of SQLAlchemy with Pyramid and provide a better starting point for new projects. See <https://github.com/Pylons/pyramid/pull/2024> for more. Highlights were:

- New SQLAlchemy session management without any global `DBSession`. Replaced by a per-request `request.dbsession` property.
- A new authentication chapter demonstrating how to get simple authentication bootstrapped quickly in an application.
- Authorization was overhauled to show the use of per-route context factories which demonstrate object-level authorization on top of simple group-level authorization. Did you want to restrict page edits to only the owner but couldn't figure it out before? Here you go!
- The users and groups are stored in the database now instead of within tutorial-specific global variables.
- User passwords are stored using `bcrypt`.

What's New in Pyramid 1.6

This article explains the new features in Pyramid version 1.6 as compared to its predecessor, Pyramid 1.5. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.6, as well as software dependency changes and notable documentation additions.

Backwards Incompatibilities

- IPython and BPython support have been removed from `pshell` in the core. To continue using them on Pyramid 1.6+, you must install the binding packages explicitly. One way to do this is by adding `pyramid_ipython` (or `pyramid_bpython`) to the `install_requires` section of your package's `setup.py` file, then re-running `setup.py develop`:


```
setup(
    #...
    install_requires=[
        'pyramid_ipython',          # new dependency
        'pyramid',
        #...
    ],
)
```

- `request.response` will no longer be mutated when using the `render_to_response()` API. It is now necessary to pass in a `response=` argument to `render_to_response()` if you wish to supply the renderer with a custom response object. If you do not pass one, then a response object will be created using the current response factory. Almost all renderers mutate the `request.response` response object (for example, the JSON renderer sets `request.response.content_type` to `application/json`). However, when invoking `render_to_response`, it is not expected that the response object being returned would be the same one used later in the request. The response object returned from `render_to_response` is now explicitly different from `request.response`. This does not change the API of a renderer. See <https://github.com/Pylons/pyramid/pull/1563>
- In an effort to combat a common issue it is now a `ConfigurationError` to register a view callable that is actually an unbound method when using the default view mapper. As unbound methods do not exist in PY3+ possible errors are detected by checking if the first parameter is named `self`. For example, `config.add_view(ViewClass.some_method, ...)` should actually be `config.add_view(ViewClass, attr='some_method')`. This was always an issue in Pyramid on PY2 but the backward incompatibility is on PY3+ where you may not use a function with the first parameter named `self`. In this case it looks too much like a common error and the exception will be raised. See <https://github.com/Pylons/pyramid/pull/1498>

Feature Additions

- Python 3.5 and pypy3 compatibility.
- `pserve --reload` will no longer crash on syntax errors. See <https://github.com/Pylons/pyramid/pull/2044>
- Cache busting for static resources has been added and is available via a new `pyramid.config.Configurator.add_cache_buster()` API. Core APIs are shipped for both cache busting via query strings and via asset manifests for integrating into custom asset pipelines. See <https://github.com/Pylons/pyramid/pull/1380> and <https://github.com/Pylons/pyramid/pull/1583> and <https://github.com/Pylons/pyramid/pull/2171>

- Assets can now be overridden by an absolute path on the filesystem when using the `override_asset()` API. This makes it possible to fully support serving up static content from a mutable directory while still being able to use the `static_url()` API and `add_static_view()`. Previously it was not possible to use `add_static_view()` with an absolute path **and** generate urls to the content. This change replaces the call, `config.add_static_view('/abs/path', 'static')`, with `config.add_static_view('myapp:static', 'static')` and `config.override_asset(to_override='myapp:static/', override_with='/abs/path/')`. The `myapp:static` asset spec is completely made up and does not need to exist—it is used for generating URLs via `request.static_url('myapp:static/foo.png')`. See <https://github.com/Pylons/pyramid/issues/1252>
- Added `set_response_factory()` and the `response_factory` keyword argument to the constructor of *Configurator* for defining a factory that will return a custom Response class. See <https://github.com/Pylons/pyramid/pull/1499>
- Added `pyramid.config.Configurator.root_package` attribute and `init` parameter to assist with includible packages that wish to resolve resources relative to the package in which the configurator was created. This is especially useful for add-ons that need to load asset specs from settings, in which case it may be natural for a developer to define imports or assets relative to the top-level package. See <https://github.com/Pylons/pyramid/pull/1337>
- Overall improvements for the `proutes` command. Added `--format` and `--glob` arguments to the command, introduced the `method` column for displaying available request methods, and improved the `view` output by showing the module instead of just `__repr__`. See <https://github.com/Pylons/pyramid/pull/1488>
- `pserve` can now take a `-b` or `--browser` option to open the server URL in a web browser. See <https://github.com/Pylons/pyramid/pull/1533>
- Support keyword-only arguments and function annotations in views in Python 3. See <https://github.com/Pylons/pyramid/pull/1556>
- The `append_slash` argument of `add_notfound_view()` will now accept anything that implements the *IResponse* interface and will use that as the response class instead of the default *HTTPFound*. See <https://github.com/Pylons/pyramid/pull/1610>
- The *Configurator* has grown the ability to allow actions to call other actions during a commit cycle. This enables much more logic to be placed into actions, such as the ability to invoke other actions or group them for improved conflict detection. We have also exposed and documented the configuration phases that Pyramid uses in order to further assist in building conforming add-ons. See <https://github.com/Pylons/pyramid/pull/1513>

- Allow an iterator to be returned from a renderer. Previously it was only possible to return bytes or unicode. See <https://github.com/Pylons/pyramid/pull/1417>
- Improve robustness to timing attacks in the *AuthTktCookieHelper* and the *SignedCookieSessionFactory* classes by using the `stdlib`'s `hmac.compare_digest` if it is available (such as Python 2.7.7+ and 3.3+). See <https://github.com/Pylons/pyramid/pull/1457>
- Improve the readability of the `pcreate` shell script output. See <https://github.com/Pylons/pyramid/pull/1453>
- Make it simple to define `notfound` and `forbidden` views that wish to use the default exception-response view, but with altered predicates and other configuration options. The `view` argument is now optional in `add_notfound_view()` and `add_forbidden_view()` See <https://github.com/Pylons/pyramid/issues/494>
- The `pshell` script will now load a `PYTHONSTARTUP` file if one is defined in the environment prior to launching the interpreter. See <https://github.com/Pylons/pyramid/pull/1448>
- Add new HTTP exception objects for status codes 428 Precondition Required, 429 Too Many Requests and 431 Request Header Fields Too Large in `pyramid.httpexceptions`. See <https://github.com/Pylons/pyramid/pull/1372/files>
- `pcreate` when run without a `scaffold` argument will now print information on the missing flag, as well as a list of available scaffolds. See <https://github.com/Pylons/pyramid/pull/1566> and <https://github.com/Pylons/pyramid/issues/1297>
- `pcreate` will now ask for confirmation if invoked with an argument for a project name that already exists or is importable in the current environment. See <https://github.com/Pylons/pyramid/issues/1357> and <https://github.com/Pylons/pyramid/pull/1837>
- Add `pyramid.request.apply_request_extensions()` function which can be used in testing to apply any request extensions configured via `config.add_request_method`. Previously it was only possible to test the extensions by going through Pyramid's router. See <https://github.com/Pylons/pyramid/pull/1581>
- Make it possible to subclass `pyramid.request.Request` and also use `pyramid.request.Request.add_request_method`. See <https://github.com/Pylons/pyramid/issues/1529>
- Additional shells for `pshell` can now be registered as entry points. See <https://github.com/Pylons/pyramid/pull/1891> and <https://github.com/Pylons/pyramid/pull/2012>
- The variables injected into `pshell` are now displayed with their docstrings instead of the default `str(obj)` when possible. See <https://github.com/Pylons/pyramid/pull/1929>

Deprecations

- The `pserve` command's daemonization features, as well as `--monitor-restart`, have been deprecated. This includes the `[start, stop, restart, status]` subcommands, as well as the `--daemon`, `--stop-daemon`, `--pid-file`, `--status`, `--user`, and `--group` flags. See <https://github.com/Pylons/pyramid/pull/2120> and <https://github.com/Pylons/pyramid/pull/2189> and <https://github.com/Pylons/pyramid/pull/1641>

Please use a real process manager in the future instead of relying on `pserve` to daemonize itself. Many options exist, including your operating system's services, such as Systemd or Upstart, as well as Python-based solutions like Circus and Supervisor.

See <https://github.com/Pylons/pyramid/pull/1641> and <https://github.com/Pylons/pyramid/pull/2120>

- The `principal` argument to `pyramid.security.remember()` was renamed to `userid`. Using `principal` as the argument name still works and will continue to work for the next few releases, but a deprecation warning is printed.

Scaffolding Enhancements

- Added line numbers to the log formatters in the scaffolds to assist with debugging. See <https://github.com/Pylons/pyramid/pull/1326>
- Updated scaffold generating machinery to return the version of Pyramid and its documentation for use in scaffolds. Updated `starter`, `alchemy` and `zodb` templates to have links to correctly versioned documentation, and to reflect which Pyramid was used to generate the scaffold.
- Removed non-ASCII copyright symbol from templates, as this was causing the scaffolds to fail for project generation.

Documentation Enhancements

- Removed logging configuration from Quick Tutorial `ini` files, except for scaffolding- and logging-related chapters, to avoid needing to explain it too early.
- Improve and clarify the documentation on what Pyramid defines as a `principal` and a `userid` in its security APIs. See <https://github.com/Pylons/pyramid/pull/1399>

- Moved the documentation for `accept` on `pyramid.config.Configurator.add_view()` to no longer be part of the predicate list. See <https://github.com/Pylons/pyramid/issues/1391> for a bug report stating `not_` was failing on `accept`. Discussion with @mcdonc led to the conclusion that it should not be documented as a predicate. See <https://github.com/Pylons/pyramid/pull/1487> for this PR.
- Clarify a previously-implied detail of the `ISession.invalidate` API documentation.
- Add documentation of command line programs (p* scripts). See <https://github.com/Pylons/pyramid/pull/2191>

What's New in Pyramid 1.5

This article explains the new features in Pyramid version 1.5 as compared to its predecessor, Pyramid 1.4. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.5, as well as software dependency changes and notable documentation additions.

Major Backwards Incompatibilities

- Pyramid no longer depends on or configures the Mako and Chameleon templating system renderers by default. Disincluding these templating systems by default means that the Pyramid core has fewer dependencies and can run on future platforms without immediate concern for the compatibility of its templating add-ons. It also makes maintenance slightly more effective, as different people can maintain the templating system add-ons that they understand and care about without needing commit access to the Pyramid core, and it allows users who just don't want to see any packages they don't use come along for the ride when they install Pyramid.

This means that upon upgrading to Pyramid 1.5a2+, projects that use either of these templating systems will see a traceback that ends something like this when their application attempts to render a Chameleon or Mako template:

```
ValueError: No such renderer factory .pt
```

Or:

```
ValueError: No such renderer factory .mako
```

Or:

```
ValueError: No such renderer factory .mak
```

Support for Mako templating has been moved into an add-on package named `pyramid_mako`, and support for Chameleon templating has been moved into an add-on package named `pyramid_chameleon`. These packages are drop-in replacements for the old built-in support for these templating languages. All you have to do is install them and make them active in your configuration to register renderer factories for `.pt` and/or `.mako` (or `.mak`) to make your application work again.

To re-add support for Chameleon and/or Mako template renderers into your existing projects, follow the below steps.

If you depend on Mako templates:

- Make sure the `pyramid_mako` package is installed. One way to do this is by adding `pyramid_mako` to the `install_requires` section of your package's `setup.py` file and afterwards rerunning `setup.py develop`:

```
setup(
    #...
    install_requires=[
        'pyramid_mako',          # new dependency
        'pyramid',
        #...
    ],
)
```

- Within the portion of your application which instantiates a *Pyramid Configurator* (often the `main()` function in your project's `__init__.py` file), tell Pyramid to include the `pyramid_mako` includeme:

```
config = Configurator(....)
config.include('pyramid_mako')
```

If you depend on Chameleon templates:

- Make sure the `pyramid_chameleon` package is installed. One way to do this is by adding `pyramid_chameleon` to the `install_requires` section of your package's `setup.py` file and afterwards rerunning `setup.py develop`:

```
setup(
    #...
    install_requires=[
        'pyramid_chameleon',          # new dependency
        'pyramid',
        #...
    ],
)
```

- Within the portion of your application which instantiates a Pyramid *Configurator* (often the `main()` function in your project's `__init__.py` file), tell Pyramid to include the `pyramid_chameleon` includeme:

```
config = Configurator(...)
config.include('pyramid_chameleon')
```

Note that it's also fine to install these packages into *older* Pyramids for forward compatibility purposes. Even if you don't upgrade to Pyramid 1.5 immediately, performing the above steps in a Pyramid 1.4 installation is perfectly fine, won't cause any difference, and will give you forward compatibility when you eventually do upgrade to Pyramid 1.5.

With the removal of Mako and Chameleon support from the core, some unit tests that use the `pyramid.renderers.render*` methods may begin to fail. If any of your unit tests are invoking either `pyramid.renderers.render()` or `pyramid.renderers.render_to_response()` with either Mako or Chameleon templates then the `pyramid.config.Configurator` instance in effect during the unit test should be also be updated to include the addons, as shown above. For example:

```
class ATest(unittest.TestCase):
    def setUp(self):
        self.config = pyramid.testing.setUp()
        self.config.include('pyramid_mako')

    def test_it(self):
        result = pyramid.renderers.render('mypkg:templates/home.mako',
        ↪ {})
```

Or:

```
class ATest(unittest.TestCase):
    def setUp(self):
```

```

        self.config = pyramid.testing.setUp()
        self.config.include('pyramid_chameleon')

    def test_it(self):
        result = pyramid.renderers.render('mypkg:templates/home.pt', {})
        ↪

```

- If you're using the Pyramid debug toolbar, when you upgrade Pyramid to 1.5a2+, you'll also need to upgrade the `pyramid_debugtoolbar` package to at least version 1.0.8, as older toolbar versions are not compatible with Pyramid 1.5a2+ due to the removal of Mako support from the core. It's fine to use this newer version of the toolbar code with older Pyramids too.

Feature Additions

The feature additions in Pyramid 1.5 follow.

- Python 3.4 compatibility.
- Add `pdistreport` script, which prints the Python version in use, the Pyramid version in use, and the version number and location of all Python distributions currently installed.
- Add the ability to invert the result of any view, route, or subscriber predicate value using the `not_` class. For example:

```

from pyramid.config import not_

@view_config(route_name='myroute', request_method=not_('POST'))
def myview(request): ...

```

The above example will ensure that the view is called if the request method is not POST, at least if no other view is more specific.

The `pyramid.config.not_` class can be used against any value that is a predicate value passed in any of these contexts:

```

- pyramid.config.Configurator.add_view()

- pyramid.config.Configurator.add_route()

- pyramid.config.Configurator.add_subscriber()

```



```
- pyramid.view.view_config()

- pyramid.events.subscriber()
```

- View lookup will now search for valid views based on the inheritance hierarchy of the context. It tries to find views based on the most specific context first, and upon predicate failure, will move up the inheritance chain to test views found by the super-type of the context. In the past, only the most specific type containing views would be checked and if no matching view could be found then a `PredicateMismatch` would be raised. Now predicate mismatches don't hide valid views registered on super-types. Here's an example that now works:

```
class IResource(Interface):

    ...

@view_config(context=IResource)
def get(context, request):

    ...

@view_config(context=IResource, request_method='POST')
def post(context, request):

    ...

@view_config(context=IResource, request_method='DELETE')
def delete(context, request):

    ...

@implementer(IResource)
class MyResource:

    ...

@view_config(context=MyResource, request_method='POST')
def override_post(context, request):

    ...
```

Previously the `override_post` view registration would hide the `get` and `delete` views in the context of `MyResource` – leading to a predicate mismatch error when trying to use `GET` or `DELETE` methods. Now the views are found and no predicate mismatch is raised. See <https://github.com/Pylons/pyramid/pull/786> and <https://github.com/Pylons/pyramid/pull/1004> and <https://github.com/Pylons/pyramid/pull/1046>

- `scripts/prequest.py` (aka the prequest console script): added support for submitting PUT and PATCH requests. See <https://github.com/Pylons/pyramid/pull/1033>. add support for submitting OPTIONS and PROPFIND requests, and allow users to specify basic authentication credentials in the request via a `--login` argument to the script. See <https://github.com/Pylons/pyramid/pull/1039>.
 - The `pyramid.config.Configurator.add_route()` method now supports being called with an external URL as pattern. See <https://github.com/Pylons/pyramid/issues/611> and the documentation section *External Routes*.
 - `pyramid.authorization.ACLAuthorizationPolicy` supports `__acl__` as a callable. This removes the ambiguity between the potential `AttributeError` that would be raised on the context when the property was not defined and the `AttributeError` that could be raised from any user-defined code within a dynamic property. It is recommended to define a dynamic ACL as a callable to avoid this ambiguity. See <https://github.com/Pylons/pyramid/issues/735>.
 - Allow a protocol-relative URL (e.g. `//example.com/images`) to be passed to `pyramid.config.Configurator.add_static_view()`. This allows externally-hosted static URLs to be generated based on the current protocol.
 - The `pyramid.authentication.AuthTktAuthenticationPolicy` class has two new options to configure its domain usage:
 - `parent_domain`: if set the authentication cookie is set on the parent domain. This is useful if you have multiple sites sharing the same domain.
 - `domain`: if provided the cookie is always set for this domain, bypassing all usual logic.
- See <https://github.com/Pylons/pyramid/pull/1028>, <https://github.com/Pylons/pyramid/pull/1072> and <https://github.com/Pylons/pyramid/pull/1078>.
- The `pyramid.authentication.AuthTktPolicy` now supports IPv6 addresses when using the `include_ip=True` option. This is possibly incompatible with alternative `auth_tkt` implementations, as the specification does not define how to properly handle IPv6. See <https://github.com/Pylons/pyramid/issues/831>.
 - Make it possible to use variable arguments via `pyramid.paster.get_appsettings()`. This also allowed the generated `initialize_db` script from the alchemy scaffold to grow support for options in the form `a=1 b=2` so you can fill in values in a parameterized `.ini` file, e.g. `initialize_myapp_db etc/development.ini a=1 b=2`. See <https://github.com/Pylons/pyramid/pull/911>

- The `request.session.check_csrf_token()` method and the `check_csrf` view predicate now take into account the value of the HTTP header named `X-CSRF-Token` (as well as the `csrf_token` form parameter, which they always did). The header is tried when the form parameter does not exist.
- You can now generate “hybrid” `urldispatch/traversal` URLs more easily by using the new `route_name`, `route_kw` and `route_remainder_name` arguments to `resource_url()` and `resource_path()`. See *Generating Hybrid URLs*.
- A new http exception superclass named `HTTPSuccessful` was added. You can use this class as the context of an exception view to catch all 200-series “exceptions” (e.g. “raise `HTTPOk`”). This also allows you to catch *only* the `HTTPOk` exception itself; previously this was impossible because a number of other exceptions (such as `HTTPNoContent`) inherited from `HTTPOk`, but now they do not.
- It is now possible to escape double braces in Pyramid scaffolds (unescaped, these represent replacement values). You can use `\{\{a\}\}` to represent a “bare” `{{a}}`. See <https://github.com/Pylons/pyramid/pull/862>
- Add `localizer` and `locale_name` properties (reified) to `pyramid.request.Request`. See <https://github.com/Pylons/pyramid/issues/508>. Note that the `pyramid.i18n.get_localizer()` and `pyramid.i18n.get_locale_name()` functions now simply look up these properties on the request.
- The `pserve` command now takes a `-v` (or `--verbose`) flag and a `-q` (or `--quiet`) flag. Output from running `pserve` can be controlled using these flags. `-v` can be specified multiple times to increase verbosity. `-q` sets verbosity to 0 unconditionally. The default verbosity level is 1.
- The `alchemy` scaffold tests now provide better coverage. See <https://github.com/Pylons/pyramid/pull/1029>
- Users can now provide dotted Python names to as the `factory` argument the Configurator methods named `add_view_predicate()`, `add_route_predicate()` and `add_subscriber_predicate()`. Instead of passing the predicate factory directly, you can pass a dotted name which refers to the factory.
- `pyramid.path.package_name()` no longer throws an exception when resolving the package name for namespace packages that have no `__file__` attribute.
- An authorization API has been added as a method of the request: `pyramid.request.Request.has_permission()`. It is a method-based alternative to the `pyramid.security.has_permission()` API and works exactly the same. The older API is now deprecated.

- Property API attributes have been added to the request for easier access to authentication data: `pyramid.request.Request.authenticated_userid`, `pyramid.request.Request.unauthenticated_userid`, and `pyramid.request.Request.effective_principals`. These are analogues, respectively, of `pyramid.security.authenticated_userid()`, `pyramid.security.unauthenticated_userid()`, and `pyramid.security.effective_principals()`. They operate exactly the same, except they are attributes of the request instead of functions accepting a request. They are properties, so they cannot be assigned to. The older function-based APIs are now deprecated.
- Pyramid's console scripts (`pserve`, `pviews`, etc) can now be run directly, allowing custom arguments to be sent to the python interpreter at runtime. For example:

```
python -3 -m pyramid.scripts.pserve development.ini
```

- Added a specific subclass of `pyramid.httpexceptions.HTTPBadRequest` named `pyramid.exceptions.BadCSRFToken` which will now be raised in response to failures in the `check_csrf_token` view predicate. See <https://github.com/Pylons/pyramid/pull/1149>
- Added a new `SignedCookieSessionFactory` which is very similar to the `UnencryptedCookieSessionFactoryConfig` but with a clearer focus on signing content. The custom serializer arguments to this function should only focus on serializing, unlike its predecessor which required the serializer to also perform signing. See <https://github.com/Pylons/pyramid/pull/1142> . Note that cookies generated using `SignedCookieSessionFactory` are not compatible with cookies generated using `UnencryptedCookieSessionFactory`, so existing user session data will be destroyed if you switch to it.
- Added a new `BaseCookieSessionFactory` which acts as a generic cookie factory that can be used by framework implementors to create their own session implementations. It provides a reusable API which focuses strictly on providing a dictionary-like object that properly handles renewals, timeouts, and conformance with the `ISession` API. See <https://github.com/Pylons/pyramid/pull/1142>
- We no longer eagerly clear `request.exception` and `request.exc_info` in the exception view tween. This makes it possible to inspect exception information within a finished callback. See <https://github.com/Pylons/pyramid/issues/1223>.

Other Backwards Incompatibilities

- Modified the `current_route_url()` method. The method previously returned the URL without the query string by default, it now does attach the query string unless it is overridden.

- The `route_url()` and `route_path()` APIs no longer quote `/` to `%2F` when a replacement value contains a `/`. This was pointless, as WSGI servers always unquote the slash anyway, and Pyramid never sees the quoted value.
- It is no longer possible to set a `locale_name` attribute of the request, nor is it possible to set a `localizer` attribute of the request. These are now “reified” properties that look up a locale name and localizer respectively using the machinery described in *Internationalization and Localization*.
- If you send an `X-Vhm-Root` header with a value that ends with any number of slashes, the trailing slashes will be removed before the URL is generated when you use `resource_url()` or `resource_path()`. Previously the virtual root path would not have trailing slashes stripped, which would influence URL generation.
- The `pyramid.interfaces.IResourceURL` interface has now grown two new attributes: `virtual_path_tuple` and `physical_path_tuple`. These should be the tuple form of the resource’s path (physical and virtual).
- Removed the `request.response_*` varying attributes (such as “`request.response_headers`”). These attributes had been deprecated since Pyramid 1.1, and as per the deprecation policy, have now been removed.
- `request.response` will no longer be mutated when using the `pyramid.renderers.render()` API. Almost all renderers mutate the `request.response` response object (for example, the JSON renderer sets `request.response.content_type` to `application/json`), but this is only necessary when the renderer is generating a response; it was a bug when it was done as a side effect of calling `pyramid.renderers.render()`.
- Removed the `bfg2pyramid` fixer script.
- The `pyramid.events.NewResponse` event is now sent **after** response callbacks are executed. It previously executed before response callbacks were executed. Rationale: it’s more useful to be able to inspect the response after response callbacks have done their jobs instead of before.
- Removed the class named `pyramid.view.static` that had been deprecated since Pyramid 1.1. Instead use `pyramid.static.static_view` with the `use_subpath=True` argument.
- Removed the `pyramid.view.is_response` function that had been deprecated since Pyramid 1.1. Use the `pyramid.request.Request.is_response()` method instead.
- Removed the ability to pass the following arguments to `pyramid.config.Configurator.add_route()`: `view`, `view_context`, `view_for`, `view_permission`, `view_renderer`, and `view_attr`. Using these arguments had been deprecated since Pyramid 1.1. Instead of passing view-related arguments to `add_route`, use a separate call to `pyramid.config.Configurator.add_view()` to associate a view with a route using its `route_name` argument. Note that this impacts the `pyramid.config.Configurator.add_static_view()` function too, because it delegates to “`add_route`”.

- Removed the ability to influence and query a `pyramid.request.Request` object as if it were a dictionary. Previously it was possible to use methods like `__getitem__`, `get`, `items`, and other dictlike methods to access values in the WSGI environment. This behavior had been deprecated since Pyramid 1.1. Use methods of `request.environ` (a real dictionary) instead.
- Removed ancient backwards compatibility hack in `pyramid.traversal.DefaultRootFactory` which populated the `__dict__` of the factory with the `matchdict` values for compatibility with BFG 0.9.
- The `renderer_globals_factory` argument to the `pyramid.config.Configurator` constructor and the corresponding argument to `setup_registry()` has been removed. The `set_renderer_globals_factory` method of `Configurator` has also been removed. The (internal) `pyramid.interfaces.IRendererGlobals` interface was also removed. These arguments, methods and interfaces had been deprecated since 1.1. Use a `BeforeRender` event subscriber as documented in the “Hooks” chapter of the Pyramid narrative documentation instead of providing renderer globals values to the configurator.
- The key/values in the `_query` parameter of `pyramid.request.Request.route_url()` and the `query` parameter of `pyramid.request.Request.resource_url()` (and their variants), used to encode a value of `None` as the string `'None'`, leaving the resulting query string to be `a=b&key=None`. The value is now dropped in this situation, leaving a query string of `a=b&key=`. See <https://github.com/Pylons/pyramid/issues/1119>

Deprecations

- Returning a `("defname", dict)` tuple from a view which has a Mako renderer is now deprecated. Instead you should use the renderer spelling `foo#defname.mak` in the view configuration definition and return a dict only.
- The `pyramid.config.Configurator.set_request_property()` method now issues a deprecation warning when used. It had been docs-deprecated in 1.4 but did not issue a deprecation warning when used.
- `pyramid.security.has_permission()` is now deprecated in favor of using `pyramid.request.Request.has_permission()`.
- The `pyramid.security.authenticated_userid()`, `pyramid.security.unauthenticated_userid()`, and `pyramid.security.effective_principals()` functions have been deprecated. Use `pyramid.request.Request.authenticated_userid`, `pyramid.request.Request.unauthenticated_userid` and `pyramid.request.Request.effective_principals` instead.

- Deprecate the `pyramid.interfaces.ITemplateRenderer` interface. It was ill-defined and became unused when Mako and Chameleon template bindings were split into their own packages.
- The `pyramid.session.UnencryptedCookieSessionFactoryConfig` API has been deprecated and is superseded by the `pyramid.session.SignedCookieSessionFactory`. Note that while the cookies generated by the `UnencryptedCookieSessionFactoryConfig` are compatible with cookies generated by old releases, cookies generated by the `SignedCookieSessionFactory` are not. See <https://github.com/Pylons/pyramid/pull/1142>

Documentation Enhancements

- A new documentation chapter named *Quick Tour of Pyramid* was added. It describes starting out with Pyramid from a high level.
- Added a *Quick Tutorial for Pyramid* to go with the Quick Tour
- Many other enhancements.

Scaffolding Enhancements

- All scaffolds have a new HTML + CSS theme.
- Updated docs and scaffolds to keep in step with new 2.0 release of *Lingua*. This included removing all `setup.cfg` files from scaffolds and documentation environments.

Dependency Changes

- Pyramid no longer depends upon Mako or Chameleon.
- Pyramid now depends on `WebOb>=1.3` (it uses `webob.cookies.CookieProfile` from 1.3+).

What's New in Pyramid 1.4

This article explains the new features in Pyramid version 1.4 as compared to its predecessor, Pyramid 1.3. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.4, as well as software dependency changes and notable documentation additions.

Major Feature Additions

The major feature additions in Pyramid 1.4 follow.

Third-Party Predicates

- Third-party custom view, route, and subscriber predicates can now be added for use by view authors via `pyramid.config.Configurator.add_view_predicate()`, `pyramid.config.Configurator.add_route_predicate()` and `pyramid.config.Configurator.add_subscriber_predicate()`. So, for example, doing this:

```
config.add_view_predicate('abc', my.package.ABCPredicate)
```

Might allow a view author to do this in an application that configured that predicate:

```
@view_config(abc=1)
```

Similar features exist for `pyramid.config.Configurator.add_route()`, and `pyramid.config.Configurator.add_subscriber()`. See *Adding a Third Party View, Route, or Subscriber Predicate* for more information.

Easy Custom JSON Serialization

- Views can now return custom objects which will be serialized to JSON by a JSON renderer by defining a `__json__` method on the object's class. This method should return values natively serializable by `json.dumps` (such as ints, lists, dictionaries, strings, and so forth). See *Serializing Custom Objects* for more information. The JSON renderer now also allows for the definition of custom type adapters to convert unknown objects to JSON serializations, in case you can't add a `__json__` method to returned objects.

Partial Mako and Chameleon Template Renderings

- The Mako renderer now supports using a def name in an asset spec. When the def name is present in the asset spec, the system will render the template named def within the template instead of rendering the entire template. An example asset spec which names a def is `package:path/to/template#defname.mako`. This will render the def named `defname` inside the template. `mako` template instead of rendering the entire template. The old way of returning a tuple in the form `('defname', {})` from the view is supported for backward compatibility.
- The Chameleon ZPT renderer now supports using a macro name in an asset spec. When the macro name is present in the asset spec, the system will render the macro listed as a `define-macro` and return the result instead of rendering the entire template. An example asset spec: `package:path/to/template#macroname.pt`. This will render the macro defined as `macroname` within the `template.pt` template instead of the entire template.

Subrequest Support

- Developers may invoke a subrequest by using the `pyramid.request.Request.invoke_subrequest()` API. This allows a developer to obtain a response from one view callable by issuing a subrequest from within a different view callable. See *Invoking a Subrequest* for more information.

Minor Feature Additions

- `pyramid.authentication.AuthTktAuthenticationPolicy` has been updated to support newer hashing algorithms such as `sha512`. Existing applications should consider updating if possible for improved security over the default `md5` hashing.
- `pyramid.config.Configurator.add_directive()` now accepts arbitrary callables like partials or objects implementing `__call__` which don't have `__name__` and `__doc__` attributes. See <https://github.com/Pylons/pyramid/issues/621> and <https://github.com/Pylons/pyramid/pull/647>.
- As of this release, the `request_method` view/route predicate, when used, will also imply that `HEAD` is implied when you use `GET`. For example, using `@view_config(request_method='GET')` is equivalent to using `@view_config(request_method=('GET', 'HEAD'))`. Using `@view_config(request_method=('GET', 'POST'))` is equivalent to using `@view_config(request_method=('GET', 'HEAD', 'POST'))`. This is because `HEAD` is a variant of `GET` that omits the body, and `WebOb` has special support to return an empty body when a `HEAD` is used.

- `pyramid.config.Configurator.add_request_method()` has been introduced to support extending request objects with arbitrary callables. This method expands on the now documentation-deprecated `pyramid.config.Configurator.set_request_property()` by supporting methods as well as properties. This method also causes less code to be executed at request construction time than `set_request_property()`.
- The static view machinery now raises rather than returns `pyramid.httpexceptions.HTTPNotFound` and `pyramid.httpexceptions.HTTPMovedPermanently` exceptions, so these can be caught by the Not Found View (and other exception views).
- When there is a predicate mismatch exception (seen when no view matches for a given request due to predicates not working), the exception now contains a textual description of the predicate which didn't match.
- An `pyramid.config.Configurator.add_permission()` directive method was added to the Configurator. This directive registers a free-standing permission introspectable into the Pyramid introspection system. Frameworks built atop Pyramid can thus use the permissions introspectable category data to build a comprehensive list of permissions supported by a running system. Before this method was added, permissions were already registered in this introspectable category as a side effect of naming them in an `pyramid.config.Configurator.add_view()` call, this method just makes it possible to arrange for a permission to be put into the permissions introspectable category without naming it along with an associated view. Here's an example of usage of `add_permission`:

```
config = Configurator()
config.add_permission('view')
```

- The `pyramid.session.UnencryptedCookieSessionFactoryConfig()` function now accepts `signed_serialize` and `signed_deserialize` hooks which may be used to influence how the sessions are marshalled (by default this is done with HMAC+pickle).
- `pyramid.testing.DummyRequest` now supports methods supplied by the `pyramid.util.InstancePropertyMixin` class such as `set_property`.
- Request properties and methods added via `pyramid.config.Configurator.add_request_method()` or `pyramid.config.Configurator.set_request_property()` are now available to tweens.
- Request properties and methods added via `pyramid.config.Configurator.add_request_method()` or `pyramid.config.Configurator.set_request_property()` are now available in the request object returned from `pyramid.paster.bootstrap()`.

- `request.context` of environment request during `pyramid.paster.bootstrap()` is now the root object if a context isn't already set on a provided request.
- `pyramid.decorator.reify` is now an API, and was added to the API documentation.
- Added the `pyramid.testing.testConfig()` context manager, which can be used to generate a configurator in a test, e.g. with `testing.testConfig(...):`.
- A new `pyramid.session.check_csrf_token()` convenience API function was added.
- A `check_csrf` view predicate was added. For example, you can now do `config.add_view(someview, check_csrf=True)`. When the predicate is checked, if the `csrf_token` value in `request.params` matches the csrf token in the request's session, the view will be permitted to execute. Otherwise, it will not be permitted to execute.
- Add `Base.metadata.bind = engine` to alchemy scaffold, so that tables defined imperatively will work.
- Comments with references to documentation sections placed in scaffold `.ini` files.
- Allow multiple values to be specified to the `request_param` view/route predicate as a sequence. Previously only a single string value was allowed. See <https://github.com/Pylons/pyramid/pull/705>
- Added an HTTP Basic authentication policy at `pyramid.authentication.BasicAuthAuthenticationPolicy`.
- The `pyramid.config.Configurator.testing_securitypolicy()` method now returns the policy object it creates.
- The `DummySecurityPolicy` created by `pyramid.config.Configurator.testing_securitypolicy()` now sets a forgotten value on the policy (the value `True`) when its `forget` method is called.
- The `DummySecurityPolicy` created by `pyramid.config.Configurator.testing_securitypolicy()` now sets a remembered value on the policy, which is the value of the principal argument it's called with when its `remember` method is called.
- New `physical_path` view predicate. If specified, this value should be a string or a tuple representing the physical traversal path of the context found via traversal for this predicate to match as true. For example: `physical_path= '/'` or `physical_path= '/a/b/c'` or `physical_path= ('', 'a', 'b', 'c')`. It's useful when you want to always potentially show a view when some object is traversed to, but you can't be sure about what kind of object it will be, so you can't use the `context` predicate.

- Added an `effective_principals` route and view predicate.
- Do not allow the `userid` returned from the `pyramid.security.authenticated_userid()` or the `userid` that is one of the list of principals returned by `pyramid.security.effective_principals()` to be either of the strings `system`, `Everyone` or `system.Authenticated` when any of the built-in authorization policies that live in `pyramid.authentication` are in use. These two strings are reserved for internal usage by Pyramid and they will no longer be accepted as valid `userid`s.
- Allow a `_depth` argument to `pyramid.view.view_config`, which will permit limited composition reuse of the decorator by other software that wants to provide custom decorators that are much like `view_config`.
- Allow an iterable of decorators to be passed to `pyramid.config.Configurator.add_view()`. This allows views to be wrapped by more than one decorator without requiring combining the decorators yourself.
- `pyramid.security.view_execution_permitted()` used to return `True` if no view could be found. It now raises a `TypeError` exception in that case, as it doesn't make sense to assert that a nonexistent view is execution-permitted. See <https://github.com/Pylons/pyramid/issues/299>.
- Small microspeed enhancement which anticipates that a `pyramid.response.Response` object is likely to be returned from a view. Some code is shortcut if the class of the object returned by a view is this class. A similar microoptimization was done to `pyramid.request.Request.is_response()`.
- Make it possible to use variable arguments on all `p*` commands (`pserve`, `pshell`, `pviews`, etc) in the form `a=1 b=2` so you can fill in values in parameterized `.ini` file, e.g. `pshell etc/development.ini http_port=8080`.
- In order to allow people to ignore unused arguments to subscriber callables and to normalize the relationship between event subscribers and subscriber predicates, we now allow both subscribers and subscriber predicates to accept only a single `event` argument even if they've been subscribed for notifications that involve multiple interfaces.

Backwards Incompatibilities

- The Pyramid router no longer adds the values `bfg.routes.route` or `bfg.routes.matchdict` to the request's WSGI environment dictionary. These values were docs-deprecated in `repoze.bfg 1.0` (effectively seven minor releases ago). If your code depended on these values, use `request.matched_route` and `request.matchdict` instead.

- It is no longer possible to pass an environ dictionary directly to `pyramid.traversal.ResourceTreeTraverser.__call__` (aka `ModelGraphTraverser.__call__`). Instead, you must pass a request object. Passing an environment instead of a request has generated a deprecation warning since Pyramid 1.1.
- Pyramid will no longer work properly if you use the `webob.request.LegacyRequest` as a request factory. Instances of the `LegacyRequest` class have a `request.path_info` which return a string. This Pyramid release assumes that `request.path_info` will unconditionally be Unicode.
- The functions from `pyramid.chameleon_zpt` and `pyramid.chameleon_text` named `get_renderer`, `get_template`, `render_template`, and `render_template_to_response` have been removed. These have issued a deprecation warning upon import since Pyramid 1.0. Use `pyramid.renderers.get_renderer()`, `pyramid.renderers.get_renderer().implementation()`, `pyramid.renderers.render()` or `pyramid.renderers.render_to_response()` respectively instead of these functions.
- The `pyramid.configuration` module was removed. It had been deprecated since Pyramid 1.0 and printed a deprecation warning upon its use. Use `pyramid.config` instead.
- The `pyramid.paster.PyramidTemplate` API was removed. It had been deprecated since Pyramid 1.1 and issued a warning on import. If your code depended on this, adjust your code to import `pyramid.scaffolds.PyramidTemplate` instead.
- The `pyramid.settings.get_settings()` API was removed. It had been printing a deprecation warning since Pyramid 1.0. If your code depended on this API, use `pyramid.threadlocal.get_current_registry().settings` instead or use the `settings` attribute of the registry available from the request (`request.registry.settings`).
- These APIs from the `pyramid.testing` module were removed. They have been printing deprecation warnings since Pyramid 1.0:
 - `registerDummySecurityPolicy`, use `pyramid.config.Configurator.testing_securitypolicy()` instead.
 - `registerResources` (aka `registerModels`), use `pyramid.config.Configurator.testing_resources()` instead.
 - `registerEventListener`, use `pyramid.config.Configurator.testing_add_subscriber()` instead.
 - `registerTemplateRenderer` (aka `registerDummyRenderer`), use `pyramid.config.Configurator.testing_add_renderer()` instead.

- `registerView`, use `pyramid.config.Configurator.add_view()` instead.
 - `registerUtility`, use `pyramid.config.Configurator.registry.registerUtility()` instead.
 - `registerAdapter`, use `pyramid.config.Configurator.registry.registerAdapter()` instead.
 - `registerSubscriber`, use `pyramid.config.Configurator.add_subscriber()` instead.
 - `registerRoute`, use `pyramid.config.Configurator.add_route()` instead.
 - `registerSettings`, use `pyramid.config.Configurator.add_settings()` instead.
- In Pyramid 1.3 and previous, the `__call__` method of a `Response` object returned by a view was invoked before any finished callbacks were executed. As of this release, the `__call__` method of a `Response` object is invoked *after* finished callbacks are executed. This is in support of the `pyramid.request.Request.invoke_subrequest()` feature.

Deprecations

- The `pyramid.config.Configurator.set_request_property()` directive has been documentation-deprecated. The method remains usable but the more featureful `pyramid.config.Configurator.add_request_method()` should be used in its place (it has all of the same capabilities but can also extend the request object with methods).
- `pyramid.authentication.AuthTktAuthenticationPolicy` will emit a deprecation warning if an application is using the policy without explicitly passing a `hashalg` argument. This is because the default is “md5” which is considered theoretically subject to collision attacks. If you really want “md5” then you must specify it explicitly to get rid of the warning.

Documentation Enhancements

- Added an *Upgrading Pyramid* chapter to the narrative documentation. It describes how to cope with deprecations and removals of Pyramid APIs and how to show Pyramid-generated deprecation warnings while running tests and while running a server.
- Added a *Invoking a Subrequest* chapter to the narrative documentation.
- All of the tutorials that use `pyramid.authentication.AuthTktAuthenticationPolicy` now explicitly pass `sha512` as a `hashalg` argument.
- Many cleanups and improvements to narrative and API docs.

Dependency Changes

- Pyramid now requires WebOb 1.2b3+ (the prior Pyramid release only relied on 1.2dev+). This is to ensure that we obtain a version of WebOb that returns `request.path_info` as text.

What's New in Pyramid 1.3

This article explains the new features in Pyramid version 1.3 as compared to its predecessor, Pyramid 1.2. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.3, as well as software dependency changes and notable documentation additions.

Major Feature Additions

The major feature additions in Pyramid 1.3 follow.

Python 3 Compatibility



Pyramid continues to run on Python 2, but Pyramid is now also Python 3 compatible. To use Pyramid under Python 3, Python 3.3 or better is required.

Many Pyramid add-ons are already Python 3 compatible. For example, `pyramid_debugtoolbar`, `pyramid_jinja2`, `pyramid_exclog`, `pyramid_tm`, `pyramid_mailer`, and `pyramid_handlers` are all Python 3-ready. But other add-ons are known to work only under Python 2. Also, some scaffolding dependencies (particularly ZODB) do not yet work under Python 3.

Please be patient as we gain full ecosystem support for Python 3. You can see more details about ongoing porting efforts at <https://github.com/Pylons/pyramid/wiki/Python-3-Porting>.

Python 3 compatibility required dropping some package dependencies and support for older Python versions and platforms. See the “Backwards Incompatibilities” section below for more information.

The `paster` Command Has Been Replaced

We've replaced the `paster` command with Pyramid-specific analogues. Why? The libraries that supported the `paster` command named `Paste` and `PasteScript` do not run under Python 3, and we were unwilling to port and maintain them ourselves. As a result, we've had to make some changes.

Previously (in Pyramid 1.0, 1.1 and 1.2), you created a Pyramid application using `paster create`, like so:

```
$ $VENV/bin/paster create -t pyramid_starter foo
```

In 1.3, you're now instead required to create an application using `pcreate` like so:

```
$ $VENV/bin/pcreate -s starter foo
```

`pcreate` is required to be used for internal Pyramid scaffolding; externally distributed scaffolding may allow for both `pcreate` and/or `paster create`.

In previous Pyramid versions, you ran a Pyramid application like so:

```
$ $VENV/bin/paster serve development.ini
```

Instead, you now must use the `pserve` command in 1.3:

```
$ $VENV/bin/pserve development.ini
```

The `ini` configuration file format supported by Pyramid has not changed. As a result, Python 2-only users can install `PasteScript` manually and use `paster serve` instead if they like. However, using `pserve` will work under both Python 2 and Python 3.

Analogues of `paster pshell`, `paster pviews`, `paster request` and `paster ptweens` also exist under the respective console script names `pshell`, `pviews`, `prequest` and `ptweens`.

`paste.httpserver` replaced by `waitress` in Scaffolds


Because the `paste.httpserver` server we used previously in scaffolds is not Python 3 compatible, we've made the default WSGI server used by Pyramid scaffolding the `waitress` server. The `waitress` server is both Python 2 and Python 3 compatible.

Once you create a project from a scaffold, its `development.ini` and `production.ini` will have the following line:


```
use = egg:waitress#main
```

Instead of this (which was the default in older versions):

```
use = egg:Paste#http
```

 `paste.httpserver` “helped” by converting header values that were Unicode into strings, which was a feature that subverted the *WSGI* specification. The `waitress` server, on the other hand implements the WSGI spec more fully. This specifically may affect you if you are modifying headers on your responses. The following error might be an indicator of this problem: **AssertionError: Header values must be strings, please check the type of the header being returned.** A common case would be returning Unicode headers instead of string headers.

Compatibility Helper Library

A new `pyramid.compat` module was added which provides Python 2/3 straddling support for Pyramid add-ons and development environments.

Introspection

A configuration introspection system was added; see *Pyramid Configuration Introspection* and *Adding Configuration Introspection* for more information on using the introspection system as a developer.

The latest release of the pyramid debug toolbar (0.9.7+) provides an “Introspection” panel that exposes introspection information to a Pyramid application developer.

New APIs were added to support introspection `pyramid.registry.Introspectable`, `pyramid.config.Configurator.introspector`, `pyramid.config.Configurator.introspectable`, `pyramid.registry.Registry.introspector`.

@view_defaults Decorator

If you use a class as a view, you can use the new `pyramid.view.view_defaults` class decorator on the class to provide defaults to the view configuration information used by every `@view_config` decorator that decorates a method of that class.

For instance, if you’ve got a class that has methods that represent “REST actions”, all which are mapped to the same route, but different request methods, instead of this:

```

1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 class RESTView(object):
5     def __init__(self, request):
6         self.request = request
7
8     @view_config(route_name='rest', request_method='GET')
9     def get(self):
10         return Response('get')
11
12     @view_config(route_name='rest', request_method='POST')
13     def post(self):
14         return Response('post')
15
16     @view_config(route_name='rest', request_method='DELETE')
17     def delete(self):
18         return Response('delete')

```

You can do this:

```

1 from pyramid.view import view_defaults
2 from pyramid.view import view_config
3 from pyramid.response import Response
4
5 @view_defaults(route_name='rest')
6 class RESTView(object):
7     def __init__(self, request):
8         self.request = request
9
10    @view_config(request_method='GET')
11    def get(self):
12        return Response('get')
13
14    @view_config(request_method='POST')

```

```
15     def post(self):
16         return Response('post')
17
18     @view_config(request_method='DELETE')
19     def delete(self):
20         return Response('delete')
```

This also works for imperative view configurations that involve a class.

See *@view_defaults Class Decorator* for more information.

Extending a Request without Subclassing

It is now possible to extend a `pyramid.request.Request` object with property descriptors without having to create a custom request factory. The new method `pyramid.config.Configurator.set_request_property()` provides an entry point for addons to register properties which will be added to each request. New properties may be reified, effectively caching the return value for the lifetime of the instance. Common use-cases for this would be to get a database connection for the request or identify the current user. The new method `pyramid.request.Request.set_property()` has been added, as well, but the configurator method should be preferred as it provides conflict detection and consistency in the lifetime of the properties.

Not Found and Forbidden View Helpers

Not Found helpers:

- New API: `pyramid.config.Configurator.add_notfound_view()`. This is a wrapper for `pyramid.config.Configurator.add_view()` which provides support for an “append_slash” feature as well as doing the right thing when it comes to permissions (a Not Found View should always be public). It should be preferred over calling `add_view` directly with `context=HTTPNotFound` as was previously recommended.
- New API: `pyramid.view.nofound_view_config`. This is a decorator constructor like `pyramid.view.view_config` that calls `pyramid.config.Configurator.add_notfound_view()` when scanned. It should be preferred over using `pyramid.view.view_config` with `context=HTTPNotFound` as was previously recommended.

Forbidden helpers:

- New API: `pyramid.config.Configurator.add_forbidden_view()`. This is a wrapper for `pyramid.config.Configurator.add_view()` which does the right thing about permissions. It should be preferred over calling `add_view` directly with `context=HTTPForbidden` as was previously recommended.
- New API: `pyramid.view.forbidden_view_config`. This is a decorator constructor like `pyramid.view.view_config` that calls `pyramid.config.Configurator.add_forbidden_view()` when scanned. It should be preferred over using `pyramid.view.view_config` with `context=HTTPForbidden` as was previously recommended.

Minor Feature Additions

- New APIs: `pyramid.path.AssetResolver` and `pyramid.path.DottedNameResolver`. The former can be used to resolve an *asset specification* to an API that can be used to read the asset's data, the latter can be used to resolve a *dotted Python name* to a module or a package.
- A `mako.directories` setting is no longer required to use Mako templates Rationale: Mako template renderers can be specified using an absolute asset spec. An entire application can be written with such asset specs, requiring no ordered lookup path.
- bpython interpreter compatibility in `pshell`. See *Alternative Shells* for more information.
- Added `pyramid.paster.get_appsettings()` API function. This function returns the settings defined within an `[app:...]` section in a `PasteDeploy` `ini` file.
- Added `pyramid.paster.setup_logging()` API function. This function sets up Python logging according to the logging configuration in a `PasteDeploy` `ini` file.
- Configuration conflict reporting is reported in a more understandable way ("Line 11 in file..." vs. a repr of a tuple of similar info).
- We allow extra keyword arguments to be passed to the `pyramid.config.Configurator.action()` method.
- Responses generated by Pyramid's `pyramid.static.static_view` now use a `wsgi.file_wrapper` (see <http://www.python.org/dev/peps/pep-0333/#optional-platform-specific-file-handling>) when one is provided by the web server.
- The `pyramid.config.Configurator.scan()` method can be passed an `ignore` argument, which can be a string, a callable, or a list consisting of strings and/or callables. This feature allows submodules, subpackages, and global objects from being scanned. See <http://readthedocs.org/docs/venusian/en/latest/#ignore-scan-argument> for more information about how to use the `ignore` argument to `scan`.

- Add `pyramid.config.Configurator.add_traverser()` API method. See *Changing the Traverser* for more information. This is not a new feature, it just provides an API for adding a traverser without needing to use the ZCA API.
- Add `pyramid.config.Configurator.add_resource_url_adapter()` API method. See *Changing How `pyramid.request.Request.resource_url()` Generates a URL* for more information. This is not a new feature, it just provides an API for adding a resource url adapter without needing to use the ZCA API.
- Better error messages when a view callable returns a value that cannot be converted to a response (for example, when a view callable returns a dictionary without a renderer defined, or doesn't return any value at all). The error message now contains information about the view callable itself as well as the result of calling it.
- Better error message when a .pyc-only module is `config.include`-ed. This is not permitted due to error reporting requirements, and a better error message is shown when it is attempted. Previously it would fail with something like "AttributeError: 'NoneType' object has no attribute 'rfind'".
- The system value `req` is now supplied to renderers as an alias for `request`. This means that you can now, for example, in a template, do `req.route_url(...)` instead of `request.route_url(...)`. This is purely a change to reduce the amount of typing required to use request methods and attributes from within templates. The value `request` is still available too, this is just an alternative.
- A new interface was added: `pyramid.interfaces.IResourceURL`. An adapter implementing its interface can be used to override resource URL generation when `pyramid.request.Request.resource_url()` is called. This interface replaces the now-deprecated `pyramid.interfaces.IContextURL` interface.
- The dictionary passed to a resource's `__resource_url__` method (see *Overriding Resource URL Generation*) now contains an `app_url` key, representing the application URL generated during `pyramid.request.Request.resource_url()`. It represents a potentially customized URL prefix, containing potentially custom scheme, host and port information passed by the user to `request.resource_url`. It should be used instead of `request.application_url` where necessary.
- The `pyramid.request.Request.resource_url()` API now accepts these arguments: `app_url`, `scheme`, `host`, and `port`. The `app_url` argument can be used to replace the URL prefix wholesale during url generation. The `scheme`, `host`, and `port` arguments can be used to replace the respective default values of `request.application_url` partially.
- A new API named `pyramid.request.Request.resource_path()` now exists. It works like `pyramid.request.Request.resource_url()` but produces a relative URL rather than an absolute one.

- The `pyramid.request.Request.route_url()` API now accepts these arguments: `_app_url`, `_scheme`, `_host`, and `_port`. The `_app_url` argument can be used to replace the URL prefix wholesale during url generation. The `_scheme`, `_host`, and `_port` arguments can be used to replace the respective default values of `request.application_url` partially.
- New APIs: `pyramid.response.FileResponse` and `pyramid.response.FileIter`, for usage in views that must serve files “manually”.

Backwards Incompatibilities

- Pyramid no longer runs on Python 2.5. This includes the most recent release of Jython and the Python 2.5 version of Google App Engine.

The reason? We could not easily “straddle” Python 2 and 3 versions and support Python 2 versions older than Python 2.6. You will need Python 2.6 or better to run this version of Pyramid. If you need to use Python 2.5, you should use the most recent 1.2.X release of Pyramid.

- The names of available scaffolds have changed and the flags supported by `pcreate` are different than those that were supported by `paster create`. For example, `pyramid_alchemy` is now just `alchemy`.
- The `paster` command is no longer the documented way to create projects, start the server, or run debugging commands. To create projects from scaffolds, `paster create` is replaced by the `pcreate` console script. To serve up a project, `paster serve` is replaced by the `pserve` console script. New console scripts named `pshell`, `pviews`, `proutes`, and `ptweens` do what their `paster <commandname>` equivalents used to do. All relevant narrative documentation has been updated. Rationale: the Paste and PasteScript packages do not run under Python 3.
- The default WSGI server run as the result of `pserve` from newly rendered scaffolding is now the `waitress` WSGI server instead of the `paste.httpserver` server. Rationale: the Paste and PasteScript packages do not run under Python 3.
- The `pshell` command (see “`paster pshell`”) no longer accepts a `--disable-ipython` command-line argument. Instead, it accepts a `-p` or `--python-shell` argument, which can be any of the values `python`, `ipython` or `bpython`.
- Removed the `pyramid.renderers.renderer_from_name` function. It has been deprecated since Pyramid 1.0, and was never an API.
- To use ZCML with versions of Pyramid `>= 1.3`, you will need `pyramid_zcml` version `>= 0.8` and `zope.configuration` version `>= 3.8.0`. The `pyramid_zcml` package version 0.8 is backwards compatible all the way to Pyramid 1.0, so you won’t be warned if you have older versions installed and upgrade Pyramid itself “in-place”; it may simply break instead (particularly if you use ZCML’s `includeOverrides` directive).

- String values passed to `pyramid.request.Request.route_url()` or `pyramid.request.Request.route_path()` that are meant to replace “remainder” matches will now be URL-quoted except for embedded slashes. For example:

```
config.add_route('remain', '/foo*remainder')
request.route_path('remain', remainder='abc / def')
# -> '/foo/abc%20/%20def'
```

Previously string values passed as remainder replacements were tacked on untouched, without any URL-quoting. But this doesn’t really work logically if the value passed is Unicode (raw unicode cannot be placed in a URL or in a path) and it is inconsistent with the rest of the URL generation machinery if the value is a string (it won’t be quoted unless by the caller).

Some folks will have been relying on the older behavior to tack on query string elements and anchor portions of the URL; sorry, you’ll need to change your code to use the `_query` and/or `_anchor` arguments to `route_path` or `route_url` to do this now.

- If you pass a bytestring that contains non-ASCII characters to `pyramid.config.Configurator.add_route()` as a pattern, it will now fail at startup time. Use Unicode instead.
- The `path_info` route and view predicates now match against `request.upath_info` (Unicode) rather than `request.path_info` (indeterminate value based on Python 3 vs. Python 2). This has to be done to normalize matching on Python 2 and Python 3.
- The `match_param` view predicate no longer accepts a dict. This will have no negative affect because the implementation was broken for dict-based arguments.
- The `pyramid.interfaces.IContextURL` interface has been deprecated. People have been instructed to use this to register a resource url adapter in the “Hooks” chapter to use to influence `pyramid.request.Request.resource_url()` URL generation for resources found via custom traversers since Pyramid 1.0.

The interface still exists and registering an adapter using it as documented in older versions still works, but this interface will be removed from the software after a few major Pyramid releases. You should replace it with an equivalent `pyramid.interfaces.IResourceURL` adapter, registered using the new `pyramid.config.Configurator.add_resource_url_adapter()` API. A deprecation warning is now emitted when a `pyramid.interfaces.IContextURL` adapter is found when `pyramid.request.Request.resource_url()` is called.

- Remove `pyramid.config.Configurator.with_context` class method. It was never an API, it is only used by `pyramid_zcml` and its functionality has been moved to that package’s latest release. This means that you’ll need to use the 0.9.2 or later release of `pyramid_zcml` with this release of Pyramid.

- The older deprecated `set_notfound_view` Configurator method is now an alias for the new `add_notfound_view` Configurator method. Likewise, the older deprecated `set_forbidden_view` is now an alias for the new `add_forbidden_view` Configurator method. This has the following impact: the context sent to views with a `(context, request)` call signature registered via the `set_notfound_view` or `set_forbidden_view` will now be an exception object instead of the actual resource context found. Use `request.context` to get the actual resource context. It's also recommended to disuse `set_notfound_view` in favor of `add_notfound_view`, and disuse `set_forbidden_view` in favor of `add_forbidden_view` despite the aliasing.

Deprecations

- The API documentation for `pyramid.view.append_slash_notfound_view` and `pyramid.view.AppendSlashNotFoundViewFactory` was removed. These names still exist and are still importable, but they are no longer APIs. Use `pyramid.config.Configurator.add_notfound_view(append_slash=True)` or `pyramid.view.notfound_view_config(append_slash=True)` to get the same behavior.
- The `set_forbidden_view` and `set_notfound_view` methods of the Configurator were removed from the documentation. They have been deprecated since Pyramid 1.1.
- All references to the `tmpl_context` request variable were removed from the docs. Its existence in Pyramid is confusing for people who were never Pylons users. It was added as a porting convenience for Pylons users in Pyramid 1.0, but it never caught on because the Pyramid rendering system is a lot different than Pylons' was, and alternate ways exist to do what it was designed to offer in Pylons. It will continue to exist "forever" but it will not be recommended or mentioned in the docs.
- Remove references to do-nothing `pyramid.debug_templates` setting in all Pyramid-provided `.ini` files. This setting previously told Chameleon to render better exceptions; now Chameleon always renders nice exceptions regardless of the value of this setting.

Known Issues

- As of this writing (the release of Pyramid 1.3b2), if you attempt to install a Pyramid project that used the alchemy scaffold via `setup.py develop` on Python 3.2, it will quit with an installation error while trying to install Pygments. If this happens, please just rerun the `setup.py develop` command again, and it will complete successfully. This is due to a minor bug in SQLAlchemy 0.7.5 under Python 3, and has been fixed in a later SQLAlchemy release. Keep an eye on <http://www.sqlalchemy.org/trac/ticket/2421>

Documentation Enhancements

- The *SQLAlchemy + URL dispatch wiki tutorial* has been updated. It now uses `@view_config` decorators and an explicit database population script.
- Minor updates to the *ZODB + Traversal Wiki Tutorial*.
- A narrative documentation chapter named *Extending Pyramid Configuration* was added; it describes how to add a custom *configuration directive*, and how use the `pyramid.config.Configurator.action()` method within custom directives. It also describes how to add *introspectable* objects.
- A narrative documentation chapter named *Pyramid Configuration Introspection* was added. It describes how to query the introspection system.
- Added an API docs chapter for `pyramid.scaffolds`.
- Added a narrative docs chapter named *Creating Pyramid Scaffolds*.
- Added a description of the `prequest` command-line script at *Invoking a Request*.
- Added a section to the “Command-Line Pyramid” chapter named *Making Your Script into a Console Script*.
- Removed the “Running Pyramid on Google App Engine” tutorial from the main docs. It survives on in the Pyramid Community Cookbook as Pyramid on Google’s App Engine (using `appengine-monkey`). Rationale: it provides the correct info for the Python 2.5 version of GAE only, and this version of Pyramid does not support Python 2.5.
- Updated the *Changing the Forbidden View* section, replacing explanations of registering a view using `add_view` or `view_config` with ones using `add_forbidden_view` or `forbidden_view_config`.
- Updated the *Changing the Not Found View* section, replacing explanations of registering a view using `add_view` or `view_config` with ones using `add_notfound_view` or `notfound_view_config`.
- Updated the *Redirecting to Slash-Appended Routes* section, replacing explanations of registering a view using `add_view` or `view_config` with ones using `add_notfound_view` or `notfound_view_config`.
- Updated all tutorials to use `pyramid.view.forbidden_view_config` rather than `pyramid.view.view_config` with an `HTTPForbidden` context.

Dependency Changes

- Pyramid no longer depends on the `zope.component` package, except as a testing dependency.
- Pyramid now depends on the following package versions: `zope.interface>=3.8.0`, `WebOb>=1.2dev`, `repoze.lru>=0.4`, `zope.deprecation>=3.5.0`, `translationstring>=0.4` for Python 3 compatibility purposes. It also, as a testing dependency, depends on `WebTest>=1.3.1` for the same reason.
- Pyramid no longer depends on the `Paste` or `PasteScript` packages. These packages are not Python 3 compatible.
- Depend on `venusian >= 1.0a3` to provide `scan ignore` support.

Scaffolding Changes

- Rendered scaffolds have now been changed to be more relocatable (fewer mentions of the package name within files in the package).
- The `routesalchemy` scaffold has been renamed `alchemy`, replacing the older (traversal-based) `alchemy` scaffold (which has been retired).
- The `alchemy` and `starter` scaffolds are Python 3 compatible.
- The `starter` scaffold now uses URL dispatch by default.

What's New in Pyramid 1.2

This article explains the new features in Pyramid version 1.2 as compared to its predecessor, Pyramid 1.1. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.2, as well as software dependency changes and notable documentation additions.

Major Feature Additions

The major feature additions in Pyramid 1.2 follow.

Debug Toolbar

The scaffolding packages that come with Pyramid now include a debug toolbar component which can be used to interactively debug an application. See *The Debug Toolbar* for more information.

`route_prefix` Argument to include

The `pyramid.config.Configurator.include()` method now accepts a `route_prefix` argument. This argument allows you to compose URL dispatch applications together from disparate packages. See *Using a Route Prefix to Compose Applications* for more information.

Tweens

A *tween* is used to wrap the Pyramid router's primary request handling function. This is a feature that can be used by Pyramid framework extensions, to provide, for example, view timing support and can provide a convenient place to hang bookkeeping code. Tweens are a little like *WSGI middleware*, but have access to Pyramid functionality such as renderers and a full-featured request object.

To support this feature, a new configurator directive exists named `pyramid.config.Configurator.add_tween()`. This directive adds a "tween".

Tweens are further described in *Registering Tweens*.

A new paster command now exists: `paster ptweens`. This command prints the current tween configuration for an application. See the section entitled *Displaying "Tweens"* for more info.

Scaffolding Changes

- All scaffolds now use the `pyramid_tm` package rather than the `repoze.tm2 middleware` to manage transaction management.
- The ZODB scaffold now uses the `pyramid_zodbconn` package rather than the `repoze.zodbconn` package to provide ZODB integration.
- All scaffolds now use the `pyramid_debugtoolbar` package rather than the `WebError` package to provide interactive debugging features.
- Projects created via a scaffold no longer depend on the `WebError` package at all; configuration in the `production.ini` file which used to require its `error_catcher middleware` has been removed. Configuring error catching / email sending is now the domain of the `pyramid_exclog` package (see http://docs.pylonsproject.org/projects/pyramid_exclog/dev/).
- All scaffolds now send the `cache_max_age` parameter to the `add_static_view` method.

Minor Feature Additions

- The `[pshell]` section in an ini configuration file now treats a `setup` key as a dotted name that points to a callable that is passed the bootstrap environment. It can mutate the environment as necessary during a `paster pshell` session. This feature is described in *Writing a Script*.
- A new configuration setting named `pyramid.includes` is now available. It is described in *Including Packages*.
- Added a `pyramid.security.NO_PERMISSION_REQUIRED` constant for use in `permission=` statements to view configuration. This constant has a value of the string `__no_permission_required__`. This string value was previously referred to in documentation; now the documentation uses the constant.
- Added a decorator-based way to configure a response adapter: `pyramid.response.response_adapter`. This decorator has the same use as `pyramid.config.Configurator.add_response_adapter()` but it's declarative.
- The `pyramid.events.BeforeRender` event now has an attribute named `rendering_val`. This can be used to introspect the value returned by a view in a `BeforeRender` subscriber.
- The Pyramid debug logger now uses the standard logging configuration (usually set up by Paste as part of startup). This means that output from e.g. `debug_notfound`, `debug_authorization`, etc. will go to the normal logging channels. The logger name of the debug logger will be the package name of the *caller* of the `Configurator`'s constructor.
- A new attribute is available on request objects: `exc_info`. Its value will be `None` until an exception is caught by the Pyramid router, after which it will be the result of `sys.exc_info()`.
- `pyramid.testing.DummyRequest` now implements the `add_finished_callback` and `add_response_callback` methods implemented by `pyramid.request.Request`.
- New methods of the `pyramid.config.Configurator` class: `set_authentication_policy()` and `set_authorization_policy()`. These are meant to be consumed mostly by add-on authors who wish to offer packages which register security policies.
- New `Configurator` method: `pyramid.config.Configurator.set_root_factory()`, which can set the root factory after the `Configurator` has been constructed.
- Pyramid no longer eagerly commits some default configuration statements at `Configurator` construction time, which permits values passed in as constructor arguments (e.g. `authentication_policy` and `authorization_policy`) to override the same settings obtained via the `pyramid.config.Configurator.include()` method.

- Better Mako rendering exceptions; the template line which caused the error is now shown when a Mako rendering raises an exception.
- New request methods: `current_route_url()`, `current_route_path()`, and `static_path()`.
- New functions in the `pyramid.url` module: `current_route_path()` and `static_path()`.
- The `pyramid.request.Request.static_url()` API (and its brethren `pyramid.request.Request.static_path()`, `pyramid.url.static_url()`, and `pyramid.url.static_path()`) now accept an absolute filename as a “path” argument. This will generate a URL to an asset as long as the filename is in a directory which was previously registered as a static view. Previously, trying to generate a URL to an asset using an absolute file path would raise a `ValueError`.
- The `RemoteUserAuthenticationPolicy`, `AuthTktAuthenticationPolicy`, and `SessionAuthenticationPolicy` constructors now accept an additional keyword argument named `debug`. By default, this keyword argument is `False`. When it is `True`, debug information will be sent to the Pyramid debug logger (usually on `stderr`) when the `authenticated_userid` or `effective_principals` method is called on any of these policies. The output produced can be useful when trying to diagnose authentication-related problems.
- New view predicate: `match_param`. Example: a view added via `config.add_view(aview, match_param='action=edit')` will be called only when the `request.matchdict` has a value inside it named `action` with a value of `edit`.
- Support an `onerror` keyword argument to `pyramid.config.Configurator.scan()`. This argument is passed to `venusian.Scanner.scan()` to influence error behavior when an exception is raised during scanning.
- The `request_method` predicate argument to `pyramid.config.Configurator.add_view()` and `pyramid.config.Configurator.add_route()` is now permitted to be a tuple of HTTP method names. Previously it was restricted to being a string representing a single HTTP method name.
- Undeprecated `pyramid.traversal.find_model`, `pyramid.traversal.model_path`, `pyramid.traversal.model_path_tuple`, and `pyramid.url.model_url`, which were all deprecated in Pyramid 1.0. There’s just not much cost to keeping them around forever as aliases to their renamed `resource_*` prefixed functions.
- Undeprecated `pyramid.view.bfg_view`, which was deprecated in Pyramid 1.0. This is a low-cost alias to `pyramid.view.view_config` which we’ll just keep around forever.
- Route pattern replacement marker names can now begin with an underscore. See <https://github.com/Pylons/pyramid/issues/276>.

Deprecations

- All Pyramid-related *deployment settings* (e.g. `debug_all`, `debug_notfound`) are now meant to be prefixed with the prefix `pyramid..` For example: `debug_all` -> `pyramid.debug_all`. The old non-prefixed settings will continue to work indefinitely but supplying them may print a deprecation warning. All scaffolds and tutorials have been changed to use prefixed settings.
- The *deployment settings* dictionary now raises a deprecation warning when you attempt to access its values via `__getattr__` instead of via `__getitem__`.

Backwards Incompatibilities

- If a string is passed as the `debug_logger` parameter to a *Configurator*, that string is considered to be the name of a global Python logger rather than a dotted name to an instance of a logger.
- The `pyramid.config.Configurator.include()` method now accepts only a single callable argument. A *sequence* of callables used to be permitted. If you are passing more than one callable to `pyramid.config.Configurator.include()`, it will break. You now must now instead make a separate call to the method for each callable.
- It may be necessary to more strictly order configuration route and view statements when using an “autocommitting” *Configurator*. In the past, it was possible to add a view which named a route name before adding a route with that name when you used an autocommitting configurator. For example:

```
config = Configurator(autocommit=True)
config.add_view('my.pkg.someview', route_name='foo')
config.add_route('foo', '/foo')
```

The above will raise an exception when the view attempts to add itself. Now you must add the route before adding the view:

```
config = Configurator(autocommit=True)
config.add_route('foo', '/foo')
config.add_view('my.pkg.someview', route_name='foo')
```

This won't effect “normal” users, only people who have legacy BFG codebases that used an automitting configurator and possibly tests that use the configurator API (the configurator returned by `pyramid.testing.setUp()` is an autocommitting configurator). The right way to get around this is to use a default non-autocommitting configurator, which does not have these directive ordering requirements:

```
config = Configurator()
config.add_view('my.pkg.someview', route_name='foo')
config.add_route('foo', '/foo')
```

The above will work fine.

- The `pyramid.config.Configurator.add_route()` directive no longer returns a route object. This change was required to make route vs. view configuration processing work properly.

Behavior Differences

- An ETag header is no longer set when serving a static file. A Last-Modified header is set instead.
- Static file serving no longer supports the `wsgi.file_wrapper` extension.
- Instead of returning a 403 Forbidden error when a static file is served that cannot be accessed by the Pyramid process' user due to file permissions, an IOError (or similar) will be raised.

Documentation Enhancements

- Narrative and API documentation which used the `route_url`, `route_path`, `resource_url`, `static_url`, and `current_route_url` functions in the `pyramid.url` package have now been changed to use eponymous methods of the request instead.
- Added a section entitled *Using a Route Prefix to Compose Applications* to the “URL Dispatch” narrative documentation chapter.
- Added a new module to the API docs: `pyramid.tweens`.
- Added a *Registering Tweens* section to the “Hooks” narrative chapter.
- Added a *Displaying “Tweens”* section to the “Command-Line Pyramid” narrative chapter.
- Added documentation for *Explicit Tween Configuration* and *Including Packages* to the “Environment Variables and .ini Files Settings” chapter.
- Added a *Logging* chapter to the narrative docs.
- All tutorials now use - The `route_url`, `route_path`, `resource_url`, `static_url`, and `current_route_url` methods of the `pyramid.request.Request` rather than the function variants imported from `pyramid.url`.
- The ZODB wiki tutorial now uses the `pyramid_zodbconn` package rather than the `repoze.zodbconn` package to provide ZODB integration.
- Added *What makes Pyramid unique* to the Introduction narrative chapter.

Dependency Changes

- Pyramid now relies on PasteScript `>= 1.7.4`. This version contains a feature important for allowing flexible logging configuration.
- Pyramid now requires Venusian 1.0a1 or better to support the `onerror` keyword argument to `pyramid.config.Configurator.scan()`.
- The `zope.configuration` package is no longer a dependency.

What's New in Pyramid 1.1

This article explains the new features in Pyramid version 1.1 as compared to its predecessor, Pyramid 1.0. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.1, as well as software dependency changes and notable documentation additions.

Terminology Changes

The term “template” used by the Pyramid documentation used to refer to both “paster templates” and “rendered templates” (templates created by a rendering engine. i.e. Mako, Chameleon, Jinja, etc.). “Paster templates” will now be referred to as “scaffolds”, whereas the name for “rendered templates” will remain as “templates.”

Major Feature Additions

The major feature additions in Pyramid 1.1 are:

- Support for the `request.response` attribute.
- New views introspection feature: `paster pviews`.
- Support for “static” routes.
- Default HTTP exception view.
- `http_cache` view configuration parameter causes Pyramid to set HTTP caching headers.
- Features that make it easier to write scripts that work in a Pyramid environment.

request.response

- Instances of the `pyramid.request.Request` class now have a `response` attribute.

The object passed to a view callable as `request` is an instance of `pyramid.request.Request`. `request.response` is an instance of the class `pyramid.response.Response`. View callables that are configured with a *renderer* will return this response object to the Pyramid router. Therefore, code in a renderer-using view callable can set response attributes such as `request.response.content_type` (before they return, e.g. a dictionary to the renderer) and this will influence the HTTP return value of the view callable.

`request.response` can also be used in view callable code that is not configured to use a renderer. For example, a view callable might do `request.response.body = '123'; return request.response`. However, the response object that is produced by `request.response` must be *returned* when a renderer is not in play in order to have any effect on the HTTP response (it is not a “global” response, and modifications to it are not somehow merged into a separately returned response object).

The `request.response` object is lazily created, so its introduction does not negatively impact performance.

paster pviews

- A new paster command named `paster pviews` was added. This command prints a summary of potentially matching views for a given path. See the section entitled *Displaying Matching Views for a Given URL* for more information.

Static Routes

- The `add_route` method of the Configurator now accepts a `static` argument. If this argument is `True`, the added route will never be considered for matching when a request is handled. Instead, it will only be useful for URL generation via `route_url` and `route_path`. See the section entitled *Static Routes* for more information.

Default HTTP Exception View

- A default exception view for the interface `pyramid.interfaces.IExceptionResponse` is now registered by default. This means that an instance of any exception class imported from `pyramid.httpexceptions` (such as `HTTPFound`) can now be raised from within view code; when raised, this exception view will render the exception to a response.

To allow for configuration of this feature, the *Configurator* now accepts an additional keyword argument named `exceptionresponse_view`. By default, this argument is populated with a default exception view function that will be used when an HTTP exception is raised. When `None` is passed for this value, an exception view for HTTP exceptions will not be registered. Passing `None` returns the behavior of raising an HTTP exception to that of Pyramid 1.0 (the exception will propagate to *middleware* and to the WSGI server).

`http_cache`

A new value `http_cache` can be used as a *view configuration* parameter.

When you supply an `http_cache` value to a view configuration, the `Expires` and `Cache-Control` headers of a response generated by the associated view callable are modified. The value for `http_cache` may be one of the following:

- A nonzero integer. If it's a nonzero integer, it's treated as a number of seconds. This number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=3600` instructs the requesting browser to 'cache this response for an hour, please'.
- A `datetime.timedelta` instance. If it's a `datetime.timedelta` instance, it will be converted into a number of seconds, and that number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=datetime.timedelta(days=1)` instructs the requesting browser to 'cache this response for a day, please'.
- Zero (0). If the value is zero, the `Cache-Control` and `Expires` headers present in all responses from this view will be composed such that client browser cache (and any intermediate caches) are instructed to never cache the response.

- A two-tuple. If it's a two tuple (e.g. `http_cache=(1, {'public':True})`), the first value in the tuple may be a nonzero integer or a `datetime.timedelta` instance; in either case this value will be used as the number of seconds to cache the response. The second value in the tuple must be a dictionary. The values present in the dictionary will be used as input to the Cache-Control response header. For example: `http_cache=(3600, {'public':True})` means 'cache for an hour, and add public to the Cache-Control header of the response'. All keys and values supported by the `webob.cachecontrol.CacheControl` interface may be added to the dictionary. Supplying `{ 'public':True }` is equivalent to calling `response.cache_control.public = True`.

Providing a non-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value)` within your view's body.

Providing a two-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value[0], **value[1])` within your view's body.

If you wish to avoid influencing, the Expires header, and instead wish to only influence Cache-Control headers, pass a tuple as `http_cache` with the first element of None, e.g.: `(None, {'public':True})`.

The environment setting `PYRAMID_PREVENT_HTTP_CACHE` and configuration file value `prevent_http_cache` are synonymous and allow you to prevent HTTP cache headers from being set by Pyramid's `http_cache` machinery globally in a process. see *Influencing HTTP Caching* and *Preventing HTTP Caching*.

Easier Scripting Writing

A new API function `pyramid.paster.bootstrap()` has been added to make writing scripts that need to work under Pyramid environment easier, e.g.:

```
from pyramid.paster import bootstrap
info = bootstrap('/path/to/my/development.ini')
request = info['request']
print request.route_url('myroute')
```

See *Writing a Script* for more details.

Minor Feature Additions

- It is now possible to invoke `paster pshell` even if the `paste.ini` file section name pointed to in its argument is not actually a Pyramid WSGI application. The shell will work in a degraded mode, and will warn the user. See “The Interactive Shell” in the “Creating a Pyramid Project” narrative documentation section.
- The `paster pshell`, `paster pviews`, and `paster proutes` commands each now under the hood uses `pyramid.paster.bootstrap()`, which makes it possible to supply an `.ini` file without naming the “right” section in the file that points at the actual Pyramid application. Instead, you can generally just run `paster {pshell|proutes|pviews} development.ini` and it will do mostly the right thing.
- It is now possible to add a `[pshell]` section to your application’s `.ini` configuration file, which influences the global names available to a `pshell` session. See *Extending the Shell*.
- The `pyramid.config.Configurator.scan()` method has grown a `**kw` argument. `kw` argument represents a set of keyword arguments to pass to the `Venusian Scanner` object created by Pyramid. (See the *Venusian* documentation for more information about `Scanner`).
- New request property: `json_body`. This property will return the JSON-decoded variant of the request body. If the request body is not well-formed JSON, this property will raise an exception.
- A JSONP render. See *JSONP Renderer* for more details.
- New authentication policy: `pyramid.authentication.SessionAuthenticationPolicy`, which uses a session to store credentials.
- A function named `pyramid.httpexceptions.exception_response()` is a shortcut that can be used to create HTTP exception response objects using an HTTP integer status code.
- Integers and longs passed as elements to `pyramid.url.resource_url()` or `pyramid.request.Request.resource_url()` e.g. `resource_url(context, request, 1, 2)` (1 and 2 are the elements) will now be converted implicitly to strings in the result. Previously passing integers or longs as elements would cause a `TypeError`.
- `pyramid_alchemy` scaffold now uses `query.get` rather than `query.filter_by` to take better advantage of identity map caching.
- `pyramid_alchemy` scaffold now has unit tests.
- Added a `pyramid.i18n.make_localizer()` API.

- An exception raised by a `pyramid.events.NewRequest` event subscriber can now be caught by an exception view.
- It is now possible to get information about why Pyramid raised a Forbidden exception from within an exception view. The `ACLDenied` object returned by the `permits` method of each stock authorization policy (`pyramid.interfaces.IAuthorizationPolicy.permits()`) is now attached to the Forbidden exception as its `result` attribute. Therefore, if you've created a Forbidden exception view, you can see the ACE, ACL, permission, and principals involved in the request as eg. `context.result.permission`, `context.result.acl`, etc within the logic of the Forbidden exception view.
- Don't explicitly prevent the `timeout` from being lower than the `reissue_time` when setting up an `pyramid.authentication.AuthTktAuthenticationPolicy` (previously such a configuration would raise a `ValueError`, now it's allowed, although typically nonsensical). Allowing the nonsensical configuration made the code more understandable and required fewer tests.
- The `pyramid.request.Request` class now has a `ResponseClass` attribute which points at `pyramid.response.Response`.
- The `pyramid.response.Response` class now has a `RequestClass` interface which points at `pyramid.request.Request`.
- It is now possible to return an arbitrary object from a Pyramid view callable even if a renderer is not used, as long as a suitable adapter to `pyramid.interfaces.IResponse` is registered for the type of the returned object by using the new `pyramid.config.Configurator.add_response_adapter()` API. See the section in the Hooks chapter of the documentation entitled *Changing How Pyramid Treats View Responses*.
- The Pyramid router will now, by default, call the `__call__` method of response objects when returning a WSGI response. This means that, among other things, the `conditional_response` feature response objects inherited from `WebOb` will now behave properly.
- New method named `pyramid.request.Request.is_response()`. This method should be used instead of the `pyramid.view.is_response()` function, which has been deprecated.
- `pyramid.exceptions.NotFound` is now just an alias for `pyramid.httpexceptions.HTTPNotFound`.
- `pyramid.exceptions.Forbidden` is now just an alias for `pyramid.httpexceptions.HTTPForbidden`.
- Added `mako.preprocessor` config file parameter; allows for a Mako preprocessor to be specified as a Python callable or Python dotted name. See <https://github.com/Pylons/pyramid/pull/183> for rationale.

- New API class: `pyramid.static.static_view`. This supersedes the (now deprecated) `pyramid.view.static` class. `pyramid.static.static_view`, by default, serves up documents as the result of the request's `path_info` attribute rather than its `subpath` attribute (the inverse was true of `pyramid.view.static`, and still is). `pyramid.static.static_view` exposes a `use_subpath` flag for use when you want the static view to behave like the older deprecated version.
- A new api function `pyramid.scripting.prepare()` has been added. It is a lower-level analogue of `pyramid.paster.bootstrap()` that accepts a request and a registry instead of a config file argument, and is used for the same purpose:

```
from pyramid.scripting import prepare
info = prepare(registry=myregistry)
request = info['request']
print request.route_url('myroute')
```

- A new API function `pyramid.scripting.make_request()` has been added. The resulting request will have a `registry` attribute. It is meant to be used in conjunction with `pyramid.scripting.prepare()` and/or `pyramid.paster.bootstrap()` (both of which accept a request as an argument):

```
from pyramid.scripting import make_request
request = make_request('/')
```

- New API attribute `pyramid.config.global_registries` is an iterable object that contains references to every Pyramid registry loaded into the current process via `pyramid.config.Configurator.make_wsgi_app()`. It also has a `last` attribute containing the last registry loaded. This is used by the scripting machinery, and is available for introspection.
- Added the `pyramid.renderers.null_renderer` object as an API. The null renderer is an object that can be used in advanced integration cases as input to the view configuration `renderer=` argument. When the null renderer is used as a view renderer argument, Pyramid avoids converting the view callable result into a `Response` object. This is useful if you want to reuse the view configuration and lookup machinery outside the context of its use by the Pyramid router. (This feature was added for consumption by the `pyramid_rpc` package, which uses view configuration and lookup outside the context of a router in exactly this way.)

Backwards Incompatibilities

- Pyramid no longer supports Python 2.4. Python 2.5 or better is required to run Pyramid 1.1+. Pyramid, however, does not work under any version of Python 3 yet.

- The Pyramid router now, by default, expects response objects returned from view callables to implement the `pyramid.interfaces.IResponse` interface. Unlike the Pyramid 1.0 version of this interface, objects which implement `IResponse` now must define a `__call__` method that accepts `environ` and `start_response`, and which returns an `app_iter` iterable, among other things. Previously, it was possible to return any object which had the three WebOb `app_iter`, `headerlist`, and `status` attributes as a response, so this is a backwards incompatibility. It is possible to get backwards compatibility back by registering an adapter to `IResponse` from the type of object you're now returning from view callables. See the section in the Hooks chapter of the documentation entitled *Changing How Pyramid Treats View Responses*.
- The `pyramid.interfaces.IResponse` interface is now much more extensive. Previously it defined only `app_iter`, `status` and `headerlist`; now it is basically intended to directly mirror the `webob.Response` API, which has many methods and attributes.
- The `pyramid.httpexceptions` classes named `HTTPFound`, `HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPSeeOther`, `HTTPUseProxy`, and `HTTPTemporaryRedirect` now accept `location` as their first positional argument rather than `detail`. This means that you can do, e.g. `return pyramid.httpexceptions.HTTPFound('http://foo')` rather than `return pyramid.httpexceptions.HTTPFound(location='http://foo')` (the latter will of course continue to work).
- The pyramid Router attempted to set a value into the key `environ['repoze.bfg.message']` when it caught a view-related exception for backwards compatibility with applications written for `repoze.bfg` during error handling. It did this by using code that looked like so:

```
# "why" is an exception object
try:
    msg = why[0]
except:
    msg = ''

environ['repoze.bfg.message'] = msg
```

Use of the value `environ['repoze.bfg.message']` was docs-deprecated in Pyramid 1.0. Our standing policy is to not remove features after a deprecation for two full major releases, so this code was originally slated to be removed in Pyramid 1.2. However, computing the `repoze.bfg.message` value was the source of at least one bug found in the wild (<https://github.com/Pylons/pyramid/issues/199>), and there isn't a foolproof way to both preserve backwards compatibility and to fix the bug. Therefore, the code which sets the value has been removed in this release. Code in exception views which relies on this value's presence in the environment should now use the `exception` attribute of the request (e.g. `request.exception[0]`) to retrieve the message instead of relying on `request.environ['repoze.bfg.message']`.

Deprecations and Behavior Differences

i Under Python 2.7+, it's necessary to pass the Python interpreter the correct warning flags to see deprecation warnings emitted by Pyramid when porting your application from an older version of Pyramid. Use the `PYTHONWARNINGS` environment variable with the value `all` in the shell you use to invoke `paster serve` to see these warnings, e.g. on UNIX, `PYTHONWARNINGS=all $VENV/bin/paster serve development.ini`. Python 2.5 and 2.6 show deprecation warnings by default, so this is unnecessary there. All deprecation warnings are emitted to the console.

- The `pyramid.view.static` class has been deprecated in favor of the newer `pyramid.static.static_view` class. A deprecation warning is raised when it is used. You should replace it with a reference to `pyramid.static.static_view` with the `use_subpath=True` argument.
- The `paster pshell`, `paster proutes`, and `paster pviews` commands now take a single argument in the form `/path/to/config.ini#sectionname` rather than the previous 2-argument spelling `/path/to/config.ini sectionname`. `#sectionname` may be omitted, in which case `#main` is assumed.
- The default Mako renderer is now configured to escape all HTML in expression tags. This is intended to help prevent XSS attacks caused by rendering unsanitized input from users. To revert this behavior in user's templates, they need to filter the expression through the 'n' filter:

```
${ myhtml | n }.
```

See <https://github.com/Pylons/pyramid/issues/193>.

- Deprecated all assignments to `request.response_*` attributes (for example `request.response_content_type = 'foo'` is now deprecated). Assignments and mutations of assignable request attributes that were considered by the framework for response influence are now deprecated: `response_content_type`, `response_headerlist`, `response_status`, `response_charset`, and `response_cache_for`. Instead of assigning these to the request object for later detection by the rendering machinery, users should use the appropriate API of the Response object created by accessing `request.response` (e.g. code which does `request.response_content_type = 'abc'` should be changed to `request.response.content_type = 'abc'`).
- Passing view-related parameters to `pyramid.config.Configurator.add_route()` is now deprecated. Previously, a view was permitted to be connected to a route using a set of `view*` parameters passed to the `add_route` method of the Configurator. This was a shorthand which replaced the need to perform a subsequent call to `add_view`. For example, it was valid (and often recommended) to do:


```
config.add_route('home', '/', view='mypackage.views.myview',
                 view_renderer='some/renderer.pt')
```

Passing `view*` arguments to `add_route` is now deprecated in favor of connecting a view to a predefined route via `pyramid.config.Configurator.add_view()` using the route's `route_name` parameter. As a result, the above example should now be spelled:

```
config.add_route('home', '/')
config.add_view('mypackage.views.myview', route_name='home',
               renderer='some/renderer.pt')
```

This deprecation was done to reduce confusion observed in IRC, as well as to (eventually) reduce documentation burden. A deprecation warning is now issued when any view-related parameter is passed to `add_route`.

See also:

See also issue #164 on GitHub.

- Passing an `environ` dictionary to the `__call__` method of a “traverser” (e.g. an object that implements `pyramid.interfaces.ITraverser` such as an instance of `pyramid.traversal.ResourceTreeTraverser`) as its `request` argument now causes a deprecation warning to be emitted. Consumer code should pass a `request` object instead. The fact that passing an `environ` dict is permitted has been documentation-deprecated since `repoze.bfg` 1.1, and this capability will be removed entirely in a future version.
- The following (undocumented, dictionary-like) methods of the `pyramid.request.Request` object have been deprecated: `__contains__`, `__delitem__`, `__getitem__`, `__iter__`, `__setitem__`, `get`, `has_key`, `items`, `iteritems`, `itervalues`, `keys`, `pop`, `popitem`, `setdefault`, `update`, and `values`. Usage of any of these methods will cause a deprecation warning to be emitted. These methods were added for internal compatibility in `repoze.bfg` 1.1 (code that currently expects a request object expected an `environ` object in `BFG` 1.0 and before). In a future version, these methods will be removed entirely.
- A custom request factory is now required to return a request object that has a `response` attribute (or “reified”/lazy property) if the request is meant to be used in a view that uses a renderer. This `response` attribute should be an instance of the class `pyramid.response.Response`.
- The JSON and string renderer factories now assign to `request.response.content_type` rather than `request.response_content_type`.

- Each built-in renderer factory now determines whether it should change the content type of the response by comparing the response's content type against the response's default content type; if the content type is the default content type (usually `text/html`), the renderer changes the content type (to `application/json` or `text/plain` for JSON and string renderers respectively).
- The `pyramid.wsgi.wsgiapp2()` now uses a slightly different method of figuring out how to “fix” `SCRIPT_NAME` and `PATH_INFO` for the downstream application. As a result, those values may differ slightly from the perspective of the downstream application (for example, `SCRIPT_NAME` will now never possess a trailing slash).
- Previously, `pyramid.request.Request` inherited from `webob.request.Request` and implemented `__getattr__`, `__setattr__` and `__delattr__` itself in order to override “ad hoc attr” WebOb behavior where attributes of the request are stored in the environ. Now, `pyramid.request.Request` inherits from (the more recent) `webob.request.BaseRequest` instead of `webob.request.Request`, which provides the same behavior. `pyramid.request.Request` no longer implements its own `__getattr__`, `__setattr__` or `__delattr__` as a result.
- Deprecated `pyramid.view.is_response()` function in favor of (newly-added) `pyramid.request.Request.is_response()` method. Determining if an object is truly a valid response object now requires access to the registry, which is only easily available as a request attribute. The `pyramid.view.is_response()` function will still work until it is removed, but now may return an incorrect answer under some (very uncommon) circumstances.
- `pyramid.response.Response` is now a subclass of `webob.response.Response` (in order to directly implement the `pyramid.interfaces.IResponse` interface, to speed up response generation).
- The “exception response” objects importable from `pyramid.httpexceptions` (e.g. `HTTPNotFound`) are no longer just import aliases for classes that actually live in `webob.exc`. Instead, we’ve defined our own exception classes within the module that mirror and emulate the `webob.exc` exception response objects almost entirely. See *Pyramid uses its own HTTP exception class hierarchy rather than webob.exc* in the Design Defense chapter for more information.
- When visiting a URL that represented a static view which resolved to a subdirectory, the `index.html` of that subdirectory would not be served properly. Instead, a redirect to `/subdir` would be issued. This has been fixed, and now visiting a subdirectory that contains an `index.html` within a static view returns the `index.html` properly.

See also:

See also issue #67 on GitHub.

- Deprecated the `pyramid.config.Configurator.set_renderer_globals_factory` method and the `renderer_globals` Configurator constructor parameter. Users should convert code using this feature to use a `BeforeRender` event. See the section *Using the Before Render Event* in the Hooks chapter.
- In Pyramid 1.0, the `pyramid.events.subscriber` directive behaved contrary to the documentation when passed more than one interface object to its constructor. For example, when the following listener was registered:

```
@subscriber(IFoo, IBar)
def expects_ifoo_events_and_ibar_events(event):
    print event
```

The Events chapter docs claimed that the listener would be registered and listening for both `IFoo` and `IBar` events. Instead, it registered an “object event” subscriber which would only be called if an `IOBJECTEvent` was emitted where the object interface was `IFoo` and the event interface was `IBar`.

The behavior now matches the documentation. If you were relying on the buggy behavior of the 1.0 `subscriber` directive in order to register an object event subscriber, you must now pass a sequence to indicate you’d like to register a subscriber for an object event. e.g.:

```
@subscriber([IFoo, IBar])
def expects_object_event(object, event):
    print object, event
```

- In 1.0, if a `pyramid.events.BeforeRender` event subscriber added a value via the `__setitem__` or `update` methods of the event object with a key that already existed in the `renderer_globals` dictionary, a `KeyError` was raised. With the deprecation of the “`add_renderer_globals`” feature of the configurator, there was no way to override an existing value in the `renderer_globals` dictionary that already existed. Now, the event object will overwrite an older value that is already in the `globals` dictionary when its `__setitem__` or `update` is called (as well as the new `setdefault` method), just like a plain old dictionary. As a result, for maximum interoperability with other third-party subscribers, if you write an event subscriber meant to be used as a `BeforeRender` subscriber, your subscriber code will now need to (using `.get` or `__contains__` of the event object) ensure no value already exists in the `renderer_globals` dictionary before setting an overriding value.
- The `pyramid.config.Configurator.add_route()` method allowed two routes with the same route to be added without an intermediate call to `pyramid.config.Configurator.commit()`. If you now receive a `ConfigurationError` at startup time that appears to be `add_route` related, you’ll need to either a) ensure that all of your route names are unique or b) call `config.commit()` before adding a second route with the name of a previously added name or c) use a Configurator that works in `autocommit` mode.

Dependency Changes

- Pyramid now depends on *WebOb* \geq 1.0.2 as tests depend on the bugfix in that release: “Fix handling of WSGI environs with missing `SCRIPT_NAME`”. (Note that in reality, everyone should probably be using 1.0.4 or better though, as WebOb 1.0.2 and 1.0.3 were effectively brownbag releases.)

Documentation Enhancements

- Added a section entitled *Writing a Script* to the “Command-Line Pyramid” chapter.
- The *ZODB + Traversal Wiki Tutorial* was updated slightly.
- The *SQLAlchemy + URL dispatch wiki tutorial* was updated slightly.
- Made `pyramid.interfaces.IAuthenticationPolicy` and `pyramid.interfaces.IAuthorizationPolicy` public interfaces, and they are now referred to within the `pyramid.authentication` and `pyramid.authorization` API docs.
- Render the function definitions for each exposed interface in `pyramid.interfaces`.
- Add missing docs reference to `pyramid.config.Configurator.set_view_mapper()` and refer to it within the documentation section entitled *Using a View Mapper*.
- Added section to the “Environment Variables and .ini File Settings” chapter in the narrative documentation section entitled *Adding a Custom Setting*.
- Added documentation for a *multidict* as `pyramid.interfaces.IMultiDict`.
- Added a section to the “URL Dispatch” narrative chapter regarding the new “static” route feature entitled *Static Routes*.
- Added API docs for `pyramid.httpexceptions.exception_response()`.
- Added *HTTP Exceptions* section to Views narrative chapter including a description of `pyramid.httpexceptions.exception_response()`.
- Added API docs for `pyramid.authentication.SessionAuthenticationPolicy`.

What's New in Pyramid 1.0

This article explains the new features in Pyramid version 1.0 as compared to its predecessor, `repoze.bfg` 1.3. It also documents backwards incompatibilities between the two versions and deprecations added to Pyramid 1.0, as well as software dependency changes and notable documentation additions.

Major Feature Additions

The major feature additions in Pyramid 1.0 are:

- New name and branding association with the Pylons Project.
- BFG conversion script
- Scaffold improvements
- Terminology changes
- Better platform compatibility and support
- Direct built-in support for the Mako templating language.
- Built-in support for sessions.
- Updated URL dispatch features
- Better imperative extensibility
- ZCML externalized
- Better support for global template variables during rendering
- View mappers
- Testing system improvements
- Authentication support improvements
- Documentation improvements

New Name and Branding

The name of `repoze.bfg` has been changed to Pyramid. The project is also now a subproject of a new entity, “The Pylons Project”. The Pylons Project is the project name for a collection of web-framework-related technologies. Pyramid was the first package in the Pylons Project. Other packages to the collection have been added over time, such as support packages useful for Pylons 1 users as well as ex-Zope users. Pyramid is the successor to both `repoze.bfg` and *Pylons* version 1.

The Pyramid codebase is derived almost entirely from `repoze.bfg` with some changes made for the sake of Pylons 1 compatibility.

Pyramid is technically backwards incompatible with `repoze.bfg`, as it has a new package name, so older imports from the `repoze.bfg` module will fail if you do nothing to your existing `repoze.bfg` application. However, you won’t have to do much to use your existing BFG applications on Pyramid. There’s automation which will change most of your import statements and ZCML declarations. See <http://docs.pylonsproject.org/projects/pyramid/current/tutorials/bfg/index.html> for upgrade instructions.

Pylons 1 users will need to do more work to use Pyramid, as Pyramid shares no “DNA” with Pylons. It is hoped that over time documentation and upgrade code will be developed to help Pylons 1 users transition to Pyramid more easily.

`repoze.bfg` version 1.3 will be its last major release. Minor updates will be made for critical bug fixes. Pylons version 1 will continue to see maintenance releases, as well.

The Repoze project will continue to exist. Repoze will be able to regain its original focus: bringing Zope technologies to WSGI. The popularity of `repoze.bfg` as its own web framework hindered this goal.

We hope that people are attracted at first by the spirit of cooperation demonstrated by the Pylons Project and the merging of development communities. It takes humility to sacrifice a little sovereignty and work together. The opposite, forking or splintering of projects, is much more common in the open source world. We feel there is a limited amount of oxygen in the space of “top-tier” Python web frameworks and we don’t do the Python community a service by over-crowding. By merging the `repoze.bfg` and the philosophically-similar Pylons communities, both gain an expanded audience and a stronger chance of future success.

BFG Conversion Script

The `bfg2pyramid` conversion script performs a mostly automated conversion of an existing `repoze.bfg` application to Pyramid. The process is described in “Converting a BFG Application to Pyramid”.

Scaffold Improvements

- The scaffolds now have much nicer CSS and graphics.
- The `development.ini`, generated by all scaffolds, is now configured to use the `WebError` interactive exception debugger by default.
- All scaffolds have been normalized: each now uses the name `main` to represent the function that returns a WSGI application, and each now has roughly the same shape of `development.ini` style.
- All preexisting scaffolds now use “imperative” configuration (`starter`, `routesalchemy`, `alchemy`, `zodb`) instead of ZCML configuration.
- The `pyramid_zodb`, `routesalchemy` and `pyramid_alchemy` scaffolds now use a default “commit veto” hook when configuring the `repoze.tm2` transaction manager in `development.ini`. This prevents a transaction from being committed when the response status code is within the 400 or 500 ranges.

See also:

See also <http://docs.repoze.org/tm2/#using-a-commit-veto>.

Terminology Changes

- The Pyramid concept previously known as “model” is now known as “resource”. As a result, the following API renames have been made. Backwards compatibility shims for the old names have been left in place in all cases:

```
pyramid.url.model_url ->
    pyramid.url.resource_url

pyramid.traversal.find_model ->
    pyramid.url.find_resource

pyramid.traversal.model_path ->
    pyramid.traversal.resource_path

pyramid.traversal.model_path_tuple ->
    pyramid.traversal.resource_path_tuple

pyramid.traversal.ModelGraphTraverser ->
    pyramid.traversal.ResourceTreeTraverser
```

```
pyramid.config.Configurator.testing_models ->
    pyramid.config.Configurator.testing_resources

pyramid.testing.registerModels ->
    pyramid.testing.registerResources

pyramid.testing.DummyModel ->
    pyramid.testing.DummyResource
```

- All documentation which previously referred to “model” now refers to “resource”.
- The starter scaffold now has a `resources.py` module instead of a `models.py` module.
- Positional argument names of various APIs have been changed from `model` to `resource`.
- The Pyramid concept previously known as “resource” is now known as “asset”. As a result, the following API changes were made. Backwards compatibility shims have been left in place as necessary:

```
pyramid.config.Configurator.absolute_resource_spec ->
    pyramid.config.Configurator.absolute_asset_spec

pyramid.config.Configurator.override_resource ->
    pyramid.config.Configurator.override_asset
```

- The (non-API) module previously known as `pyramid.resource` is now known as `pyramid.asset`.
- All docs that previously referred to “resource specification” now refer to “asset specification”.
- The setting previously known as `BFG_RELOAD_RESOURCES` (envvar) or `reload_resources` (config file) is now known, respectively, as `PYRAMID_RELOAD_ASSETS` and `reload_assets`.

Better Platform Compatibility and Support

We’ve made Pyramid’s test suite pass on both Jython and PyPy. However, Chameleon doesn’t work on either, so you’ll need to use Mako or Jinja2 templates on these platforms.

Sessions

Pyramid now has built-in sessioning support, documented in *Sessions*. The sessioning implementation is pluggable. It also provides flash messaging and cross-site-scripting prevention features.

Using `request.session` now returns a (dictionary-like) session object if a *session factory* has been configured.

A new argument to the Configurator constructor exists: `session_factory` and a new method on the configurator exists: `pyramid.config.Configurator.set_session_factory()`.

Mako

In addition to Chameleon templating, Pyramid now also provides built-in support for *Mako* templating. See *Available Add-On Template System Bindings* for more information.

URL Dispatch

- URL Dispatch now allows for replacement markers to be located anywhere in the pattern, instead of immediately following a `/`.
- URL Dispatch now uses the form `{marker}` to denote a replace marker in the route pattern instead of `:marker`. The old colon-style marker syntax is still accepted for backwards compatibility. The new format allows a regular expression for that marker location to be used instead of the default `[^/]+`, for example `{marker:\d+}` is now valid to require the marker to be digits.
- Added a new API `pyramid.url.current_route_url()`, which computes a URL based on the “current” route (if any) and its matchdict values.
- Added a `paster proute` command which displays a summary of the routing table. See the narrative documentation section entitled *Displaying All Application Routes*.
- Added `debug_routematch` configuration setting (settable in your `.ini` file) that logs matched routes including the matchdict and predicates.
- Add a `pyramid.url.route_path()` API, allowing folks to generate relative URLs. Calling `route_path` is the same as calling `pyramid.url.route_url()` with the argument `_app_url` equal to the empty string.
- Add a `pyramid.request.Request.route_path()` API. This is a convenience method of the request which calls `pyramid.url.route_url()`.
- Added class vars `matchdict` and `matched_route` to `pyramid.request.Request`. Each is set to `None` when a route isn’t matched during a request.

ZCML Externalized

- The `load_zcml` method of a `Configurator` has been removed from the Pyramid core. Loading ZCML is now a feature of the `pyramid_zcml` package, which can be downloaded from PyPI. Documentation for the package should be available via http://docs.pylonsproject.org/projects/pyramid_zcml/en/latest/, which describes how to add a configuration statement to your `main` block to re-obtain this method. You will also need to add an `install_requires` dependency upon the `pyramid_zcml` distribution to your `setup.py` file.
- The “Declarative Configuration” narrative chapter has been removed (it was moved to the `pyramid_zcml` package).
- Most references to ZCML in narrative chapters have been removed or redirected to `pyramid_zcml` locations.
- The `starter_zcml` paster scaffold has been moved to the `pyramid_zcml` package.

Imperative Two-Phase Configuration

To support application extensibility, the *Pyramid Configurator*, by default, now detects configuration conflicts and allows you to include configuration imperatively from other packages or modules. It also, by default, performs configuration in two separate phases. This allows you to ignore relative configuration statement ordering in some circumstances. See *Advanced Configuration* for more information.

The `pyramid.config.Configurator.add_directive()` allows framework extenders to add methods to the configurator, which allows extenders to avoid subclassing a `Configurator` just to add methods. See *Adding Methods to the Configurator via add_directive* for more info.

Surrounding application configuration with `config.begin()` and `config.end()` is no longer necessary. All scaffolds have been changed to no longer call these functions.

Better Support for Global Template Variables During Rendering

A new event type named `pyramid.interfaces.IBeforeRender` is now sent as an event before a renderer is invoked. Applications may now subscribe to the `IBeforeRender` event type in order to introspect the and modify the set of renderer globals before they are passed to a renderer. The event object itself has a dictionary-like interface that can be used for this purpose. For example:

```
from repoze.events import subscriber
from pyramid.interfaces import IRendererGlobalsEvent

@subscriber(IRendererGlobalsEvent)
def add_global(event):
    event['mykey'] = 'foo'
```

View Mappers

A “view mapper” subsystem has been extracted, which allows framework extenders to control how view callables are constructed and called. This feature is not useful for “civilians”, only for extension writers. See *Using a View Mapper* for more information.

Testing Support Improvements

The `pyramid.testing.setUp()` and `pyramid.testing.tearDown()` APIs have been un-deprecated. They are now the canonical setup and teardown APIs for test configuration, replacing “direct” creation of a Configurator. This is a change designed to provide a facade that will protect against any future Configurator deprecations.

Authentication Support Improvements

- The `pyramid.interfaces.IAuthenticationPolicy` interface now specifies an `unauthenticated_userid` method. This method supports an important optimization required by people who are using persistent storages which do not support object caching and whom want to create a “user object” as a request attribute.
- A new API has been added to the `pyramid.security` module named `unauthenticated_userid`. This API function calls the `unauthenticated_userid` method of the effective security policy.
- The class `pyramid.authentication.AuthTktCookieHelper` is now an API. This class can be used by third-party authentication policy developers to help in the mechanics of authentication cookie-setting.
- The `pyramid.authentication.AuthTktAuthenticationPolicy` now accepts a `tokens` parameter via `pyramid.security.remember()`. The value must be a sequence of strings. Tokens are placed into the `auth_tkt` “tokens” field and returned in the `auth_tkt` cookie.
- Added a `wild_domain` argument to `pyramid.authentication.AuthTktAuthenticationPolicy`, which defaults to `True`. If it is set to `False`, the feature of the policy which sets a cookie with a wildcard domain will be turned off.

Documentation Improvements

- Casey Duncan, a good friend, and an excellent technical writer has given us the gift of professionally editing the entire Pyramid documentation set. Any faults in the documentation are the development team's, and all improvements are his.
- The “Resource Location and View Lookup” chapter has been replaced with a variant of Rob Miller's “Much Ado About Traversal” (originally published at <http://blog.nonsequitarian.org/2010/much-ado-about-traversal/>).
- Many users have contributed documentation fixes and improvements including Ben Bangert, Blaise Laflamme, Rob Miller, Mike Orr, Carlos de la Guardia, Paul Everitt, Tres Seaver, John Shipman, Marius Gedminas, Chris Rossi, Joachim Krebs, Xavier Spriet, Reed O'Brien, William Chambers, Charlie Choiniere, and Jamaludin Ahmad.

Minor Feature Additions

- The `settings` dictionary passed to the Configurator is now available as `config.registry.settings` in configuration code and `request.registry.settings` in view code).
- `pyramid.config.Configurator.add_view()` now accepts a `decorator` keyword argument, a callable which will decorate the view callable before it is added to the registry.
- Allow static renderer provided during view registration to be overridden at request time via a request attribute named `override_renderer`, which should be the name of a previously registered renderer. Useful to provide “omnipresent” RPC using existing rendered views.
- If a resource implements a `__resource_url__` method, it will be called as the result of invoking the `pyramid.url.resource_url()` function to generate a URL, overriding the default logic. See *Generating the URL of a Resource* for more information.
- The name `registry` is now available in a `pshell` environment by default. It is the application registry object.
- Added support for json on Google App Engine by catching `NotImplementedError` and importing `simplejson` from `django.utils`.
- Added the `pyramid.httpexceptions` module, which is a facade for the `webob.exc` module.
- New class: `pyramid.response.Response`. This is a pure facade for `webob.Response` (old code need not change to use this facade, it's existence is mostly for vanity and documentation-generation purposes).
- The request now has a new attribute: `tmpl_context` for benefit of Pylons users.
- New API methods for `pyramid.request.Request`: `model_url`, `route_url`, and `static_url`. These are simple passthroughs for their respective functions in `pyramid.url`.

Backwards Incompatibilities

- When a `pyramid.exceptions.Forbidden` error is raised, its status code now 403 Forbidden. It was previously 401 Unauthorized, for backwards compatibility purposes with `repoze.bfg`. This change will cause problems for users of Pyramid with `repoze.who`, which intercepts 401 Unauthorized by default, but allows 403 Forbidden to pass through. Those deployments will need to configure `repoze.who` to also react to 403 Forbidden. To do so, use a `repoze.who` `challenge_decider` that looks like this:

```
import zope.interface
from repoze.who.interfaces import IChallengeDecider

def challenge_decider(envron, status, headers):
    return status.startswith('403') or status.startswith('401')
zope.interface.directlyProvides(challenge_decider, IChallengeDecider)
```

- The `paster bfgshell` command is now known as `paster pshell`.
- There is no longer an `IDebugLogger` object registered as a named utility with the name `repoze.bfg.debug`.
- These deprecated APIs have been removed: `pyramid.testing.registerViewPermission`, `pyramid.testing.registerRoutesMapper`, `pyramid.request.get_request`, `pyramid.security.Unauthorized`, `pyramid.view.view_execution_permitted`, `pyramid.view.NotFound`
- The Venusian “category” for all built-in Venusian decorators (e.g. `subscriber` and `view_config/bfg_view`) is now `pyramid` instead of `bfg`.
- The `pyramid.renderers.rendered_response` function removed; use `pyramid.renderers.render_to_response()` instead.
- Renderer factories now accept a *renderer info object* rather than an absolute resource specification or an absolute path. The object has the following attributes: `name` (the `renderer=` value), `package` (the ‘current package’ when the renderer configuration statement was found), `type`: the renderer type, `registry`: the current registry, and `settings`: the deployment settings dictionary. Third-party `repoze.bfg` renderer implementations that must be ported to Pyramid will need to account for this. This change was made primarily to support more flexible Mako template rendering.
- The presence of the key `repoze.bfg.message` in the WSGI environment when an exception occurs is now deprecated. Instead, code which relies on this `environ` value should use the `exception` attribute of the request (e.g. `request.exception[0]`) to retrieve the message.

- The values `bfg_localizer` and `bfg_locale_name` kept on the request during internationalization for caching purposes were never APIs. These however have changed to `localizer` and `locale_name`, respectively.
- The default `cookie_name` value of the `pyramid.authentication.AuthTktAuthenticationPolicy` now defaults to `auth_tkt` (it used to default to `repoze.bfg.auth_tkt`).
- The `pyramid.testing.zcml_configure()` API has been removed. It had been advertised as removed since `repoze.bfg 1.2a1`, but hadn't actually been.
- All environment variables which used to be prefixed with `BFG_` are now prefixed with `PYRAMID_` (e.g. `BFG_DEBUG_NOTFOUND` is now `PYRAMID_DEBUG_NOTFOUND`).
- Since the `pyramid.interfaces.IAuthenticationPolicy` interface now specifies that a policy implementation must implement an `unauthenticated_userid` method, all third-party custom authentication policies now must implement this method. It, however, will only be called when the global function named `pyramid.security.unauthenticated_userid()` is invoked, so if you're not invoking that, you will not notice any issues.
- The `configure_zcml` setting within the deployment settings (within `**settings` passed to a Pyramid main function) has ceased to have any meaning.
- The `make_app` function has been removed from the `pyramid.router` module. It continues life within the `pyramid_zcml` package. This leaves the `pyramid.router` module without any API functions.

Deprecations and Behavior Differences

- `pyramid.configuration.Configurator` is now deprecated. Use `pyramid.config.Configurator`, passing its constructor `autocommit=True` instead. The `pyramid.configuration.Configurator` alias will live for a long time, as every application uses it, but its import now issues a deprecation warning. The `pyramid.config.Configurator` class has the same API as the `pyramid.configuration.Configurator` class, which it means to replace, except by default it is a *non-autocommitting* configurator. The now-deprecated `pyramid.configuration.Configurator` will *autocommit* every time a configuration method is called. The `pyramid.configuration` module remains, but it is deprecated. Use `pyramid.config` instead.
- The `pyramid.settings.get_settings()` API is now deprecated. Use `pyramid.threadlocals.get_current_registry().settings` instead or use the `settings` attribute of the registry available from the request (`request.registry.settings`).

- The decorator previously known as `pyramid.view.bfg_view` is now known most formally as `pyramid.view.view_config` in docs and scaffolds.
- Obtaining the settings object via `registry.{get|query}Utility(ISettings)` is now deprecated. Instead, obtain the settings object via the `registry.settings` attribute. A backwards compatibility shim was added to the registry object to register the settings object as an `ISettings` utility when `setattr(registry, 'settings', foo)` is called, but it will be removed in a later release.
- Obtaining the settings object via `pyramid.settings.get_settings()` is now deprecated. Obtain it instead as the `settings` attribute of the registry now (obtain the registry via `pyramid.threadlocal.get_registry()` or as `request.registry`).

Dependency Changes

- Depend on Venusian ≥ 0.5 (for scanning conflict exception decoration).

Documentation Enhancements

- Added a `pyramid.httpexceptions` API documentation chapter.
- Added a `pyramid.session` API documentation chapter.
- Added an API chapter for the `pyramid.response` module.
- Added a *Sessions* narrative documentation chapter.
- All documentation which previously referred to `webob.Response` now uses `pyramid.response.Response` instead.
- The documentation has been overhauled to use imperative configuration, moving declarative configuration (ZCML) explanations to an external package, `pyramid_zcml`.
- Removed `zodbsessions` tutorial chapter. It's still useful, but we now have a `SessionFactory` abstraction which competes with it, and maintaining documentation on both ways to do it is a distraction.
- Added an example of `WebTest` functional testing to the testing narrative chapter at *Creating Functional Tests*.
- Extended the Resources chapter with examples of calls to resource-related APIs.

- Add “Pyramid Provides More Than One Way to Do It” to Design Defense documentation.
- The (weak) “Converting a CMF Application to Pyramid” tutorial has been removed from the tutorials section. It was moved to the `pyramid_tutorials` Github repository at http://docs.pylonsproject.org/projects/pyramid_tutorials/dev/.
- Moved “Using ZODB With ZEO” and “Using repoze.catalog Within Pyramid” tutorials out of core documentation and into the Pyramid Tutorials site (http://docs.pylonsproject.org/projects/pyramid_tutorials/dev/).
- Removed API documentation for deprecated `pyramid.testing` APIs named `registerDummySecurityPolicy`, `registerResources`, `registerModels`, `registerEventListener`, `registerTemplateRenderer`, `registerDummyRenderer`, `registerView`, `registerUtility`, `registerAdapter`, `registerSubscriber`, `registerRoute`, and `registerSettings`.

Pyramid Change History

1.7.6 (2017-06-11)

- Fix a bug in which `pyramid.security.ALL_PERMISSIONS` failed to return a valid iterator in its `__iter__` implementation. See <https://github.com/Pylons/pyramid/pull/3077>

1.7.5 (2017-03-12)

- `HTTPException`’s accepts a detail kwarg that may be used to pass additional details to the exception. You may now pass objects so long as they have a valid `__str__` method. See <https://github.com/Pylons/pyramid/pull/2951>
- Fix a reference cycle causing memory leaks in which the registry would keep a `Configurator` instance alive even after the configurator was discarded. Another fix was also added for the `global_registries` object in which the registry was stored in a closure preventing it from being deallocated. See <https://github.com/Pylons/pyramid/pull/2972>

1.7.4 (2017-01-24)

- Update HACKING.txt from stale branch that was never merged to master. See <https://github.com/Pylons/pyramid/pull/2785>
- Fix an inconsistency in the documentation between view predicates and route predicates and highlight the differences in their APIs. See <https://github.com/Pylons/pyramid/pull/2765>
- Fix incompatibilities in the `prequest` test suite caused by changes in WebOb 1.7. See <https://github.com/Pylons/pyramid/pull/2788>
- Fix bug in `i18n` where the default domain would always use the Germanic plural style, even if a different plural function is defined in the relevant messages file. See <https://github.com/Pylons/pyramid/pull/2865>

1.7.3 (2016-08-17)

Bug Fixes

- Oops, Apparently wheels do not build cleanly every time, so build artifacts from 1.6.3 crept into the wheel for 1.7.2. Note to self: `rm -rf build`.

1.7.2 (2016-08-16)

- Revert changes from #2706 released in Pyramid 1.7.1. JSON renderers will continue to return unicode data instead of UTF-8 encoded bytes. This means that WebOb responses are still expected to handle unicode data even though JSON does not have a charset. See <https://github.com/Pylons/pyramid/issues/2744>

1.7.1 (2016-08-16)

- Change flake8 noqa directive to ignore only a single line instead of the entire file in scaffold and documentation. See <https://github.com/Pylons/pyramid/pull/2646>
- Add option to build docs as PDF only via tox. See: <https://github.com/Pylons/pyramid/issues/2575>

- Correct the column type used in the SQLAlchemy + URL Dispatch tutorial by changing it from Integer to Text. See <https://github.com/Pylons/pyramid/pull/2591>
- Fix a bug in which the `password_hash` in the Wiki2 tutorial was sometimes being treated as bytes instead of unicode. See <https://github.com/Pylons/pyramid/pull/2705>
- Properly emit a `DeprecationWarning` for using `pyramid.config.Configurator.set_request_property` instead of `pyramid.config.Configurator.add_request_method`.
- Updated Windows installation instructions and related bits. See: <https://github.com/Pylons/pyramid/issues/2661>
- Fixed bug in *proutes* such that it now shows the correct view when a class and *attr* is involved. See: <https://github.com/Pylons/pyramid/pull/2687>
- The JSON renderers now encode their result as UTF-8. The renderer helper will now warn the user and encode the result as UTF-8 if a renderer returns a text type and the response does not have a valid character set. See <https://github.com/Pylons/pyramid/pull/2706>

1.7 (2016-05-19)

- Fix a bug in the wiki2 tutorial where `bcrypt` is always expecting byte strings. See <https://github.com/Pylons/pyramid/pull/2576>
- Simplify windows detection code and remove some duplicated data. See <https://github.com/Pylons/pyramid/pull/2585> and <https://github.com/Pylons/pyramid/pull/2586>

1.7b4 (2016-05-12)

- Fixed the exception view tween to re-raise the original exception if no exception view could be found to handle the exception. This better allows tweens further up the chain to handle exceptions that were left unhandled. Previously they would be converted into a `PredicateMismatch` exception if predicates failed to allow the view to handle the exception. See <https://github.com/Pylons/pyramid/pull/2567>
- Exposed the `pyramid.interfaces.IRequestFactory` interface to mirror the public `pyramid.interfaces.IResponseFactory` interface.

1.7b3 (2016-05-10)

- Fix `request.invoke_exception_view` to raise an `HTTPNotFound` exception if no view is matched. Previously `None` would be returned if no views were matched and a `PredicateMismatch` would be raised if a view “almost” matched (a view was found matching the context). See <https://github.com/Pylons/pyramid/pull/2564>
- Add defaults for `py.test` configuration and coverage to all three scaffolds, and update documentation accordingly. See <https://github.com/Pylons/pyramid/pull/2550>
- Add `linkcheck` to `Makefile` for `Sphinx`. To check the documentation for broken links, use the command `make linkcheck SPHINXBUILD=$VENV/bin/sphinx-build`. Also removed and fixed dozens of broken external links.
- Fix the internal runner for scaffold tests to ensure they work with `pip` and `py.test`. See <https://github.com/Pylons/pyramid/pull/2565>

1.7b2 (2016-05-01)

- Removed inclusion of `pyramid_tm` in `development.ini` for `alchemy` scaffold See <https://github.com/Pylons/pyramid/issues/2538>
- A default permission set via `config.set_default_permission` will no longer be enforced on an exception view. This has been the case for a while with the default exception views (`config.add_notfound_view` and `config.add_forbidden_view`), however for any other exception view a developer had to remember to set `permission=NO_PERMISSION_REQUIRED` or be surprised when things didn’t work. It is still possible to force a permission check on an exception view by setting the `permission` argument manually to `config.add_view`. This behavior is consistent with the new `CSRF` features added in the 1.7 series. See <https://github.com/Pylons/pyramid/pull/2534>

1.7b1 (2016-04-25)

- This release announces the beta period for 1.7.
- Fix an issue where some files were being included in the `alchemy` scaffold which had been removed from the 1.7 series. See <https://github.com/Pylons/pyramid/issues/2525>

1.7a2 (2016-04-19)

Features

- Automatic CSRF checks are now disabled by default on exception views. They can be turned back on by setting the appropriate *require_csrf* option on the view. See <https://github.com/Pylons/pyramid/pull/2517>
- The automatic CSRF API was reworked to use a config directive for setting the options. The `pyramid.require_default_csrf` setting is no longer supported. Instead, a new `config.set_default_csrf_options` directive has been introduced that allows the developer to specify the default value for `require_csrf` as well as change the CSRF token, header and safe request methods. The `pyramid.csrf_trusted_origins` setting is still supported. See <https://github.com/Pylons/pyramid/pull/2518>

Bug fixes

- CSRF origin checks had a bug causing the checks to always fail. See <https://github.com/Pylons/pyramid/pull/2512>
- Fix the test suite to pass on windows. See <https://github.com/Pylons/pyramid/pull/2520>

1.7a1 (2016-04-16)

Backward Incompatibilities

- Following the Pyramid deprecation period (1.4 -> 1.6), `AuthTktAuthenticationPolicy`'s default hashing algorithm is changing from md5 to sha512. If you are using the authentication policy and need to continue using md5, please explicitly set `hashalg` to 'md5'.

This change does mean that any existing auth tickets (and associated cookies) will no longer be valid, and users will no longer be logged in, and have to login to their accounts again.

See <https://github.com/Pylons/pyramid/pull/2496>

- The `check_csrf_token` function no longer validates a csrf token in the query string of a request. Only headers and request bodies are supported. See <https://github.com/Pylons/pyramid/pull/2500>

Features

- Added a new setting, `pyramid.require_default_csrf` which may be used to turn on CSRF checks globally for every POST request in the application. This should be considered a good default for websites built on Pyramid. It is possible to opt-out of CSRF checks on a per-view basis by setting `require_csrf=False` on those views. See <https://github.com/Pylons/pyramid/pull/2413>
- Added a `require_csrf` view option which will enforce CSRF checks on any request with an unsafe method as defined by RFC2616. If the CSRF check fails a `BadCSRFToken` exception will be raised and may be caught by exception views (the default response is a 400 Bad Request). This option should be used in place of the deprecated `check_csrf` view predicate which would normally result in unexpected 404 Not Found response to the client instead of a catchable exception. See <https://github.com/Pylons/pyramid/pull/2413> and <https://github.com/Pylons/pyramid/pull/2500>
- Added an additional CSRF validation that checks the origin/referrer of a request and makes sure it matches the current `request.domain`. This particular check is only active when accessing a site over HTTPS as otherwise browsers don't always send the required information. If this additional CSRF validation fails a `BadCSRFOrigin` exception will be raised and may be caught by exception views (the default response is 400 Bad Request). Additional allowed origins may be configured by setting `pyramid.csrf_trusted_origins` to a list of domain names (with ports if on a non standard port) to allow. Subdomains are not allowed unless the domain name has been prefixed with a `..` See <https://github.com/Pylons/pyramid/pull/2501>
- Added a new `pyramid.session.check_csrf_origin` API for validating the origin or referrer headers against the request's domain. See <https://github.com/Pylons/pyramid/pull/2501>
- Pyramid HTTPExceptions will now take into account the best match for the clients Accept header, and depending on what is requested will return text/html, application/json or text/plain. The default for `/` is still text/html, but if application/json is explicitly mentioned it will now receive a valid JSON response. See <https://github.com/Pylons/pyramid/pull/2489>
- A new event and interface (`BeforeTraversal`) has been introduced that will notify listeners before traversal starts in the router. See <https://github.com/Pylons/pyramid/pull/2469> and <https://github.com/Pylons/pyramid/pull/1876>
- Add a new “view deriver” concept to Pyramid to allow framework authors to inject elements into the standard Pyramid view pipeline and affect all views in an application. This is similar to a decorator except that it has access to options passed to `config.add_view` and can affect other stages of the pipeline such as the raw response from a view or prior to security checks. See <https://github.com/Pylons/pyramid/pull/2021>

- Allow a leading `=` on the key of the request param predicate. For example, `'=abc=1'` is equivalent down to `request.params['=abc'] == '1'`. See <https://github.com/Pylons/pyramid/pull/1370>
- A new `request.invoke_exception_view(...)` method which can be used to invoke an exception view and get back a response. This is useful for rendering an exception view outside of the context of the excview tween where you may need more control over the request. See <https://github.com/Pylons/pyramid/pull/2393>
- Allow using variable substitutions like `%(LOGGING_LOGGER_ROOT_LEVEL)s` for logging sections of the `.ini` file and populate these variables from the `pserve` command line – e.g.: `pserve development.ini LOGGING_LOGGER_ROOT_LEVEL=DEBUG` See <https://github.com/Pylons/pyramid/pull/2399>

Documentation Changes

- A complete overhaul of the docs:
 - Use `pip` instead of `easy_install`.
 - Become opinionated by preferring Python 3.4 or greater to simplify installation of Python and its required packaging tools.
 - Use `venv` for the tool, and virtual environment for the thing created, instead of `virtualenv`.
 - Use `py.test` and `pytest-cov` instead of `nose` and `coverage`.
 - Further updates to the scaffolds as well as tutorials and their `src` files.

See <https://github.com/Pylons/pyramid/pull/2468>

- A complete overhaul of the `alchemy` scaffold as well as the Wiki2 SQLAlchemy + URLDispatch tutorial to introduce more modern features into the usage of SQLAlchemy with Pyramid and provide a better starting point for new projects. See <https://github.com/Pylons/pyramid/pull/2024>

Bug Fixes

- Fix `pserve --browser` to use the `--server-name` instead of the app name when selecting a section to use. This was only working for people who had `server` and `app` sections with the same name, for example `[app:main]` and `[server:main]`. See <https://github.com/Pylons/pyramid/pull/2292>

Deprecations

- The `check_csrf` view predicate has been deprecated. Use the new `require_csrf` option or the `pyramid.require_default_csrf` setting to ensure that the `BadCSRFToken` exception is raised. See <https://github.com/Pylons/pyramid/pull/2413>
- Support for Python 3.3 will be removed in Pyramid 1.8. <https://github.com/Pylons/pyramid/issues/2477>
- Python 2.6 is no longer supported by Pyramid. See <https://github.com/Pylons/pyramid/issues/2368>
- Dropped Python 3.2 support. See <https://github.com/Pylons/pyramid/pull/2256>

1.6 (2016-01-03)

Deprecations

- Continue removal of `pserve` daemon/process management features by deprecating `--user` and `--group` options. See <https://github.com/Pylons/pyramid/pull/2190>

1.6b3 (2015-12-17)

Backward Incompatibilities

- Remove the `cachebust` option from `config.add_static_view`. See `config.add_cache_buster` for the new way to attach cache busters to static assets. See <https://github.com/Pylons/pyramid/pull/2186>
- Modify the `pyramid.interfaces.ICacheBuster` API to be a simple callable instead of an object with `match` and `pregenerate` methods. Cache busters are now focused solely on generation. Matching has been dropped.

Note this affects usage of `pyramid.static.QueryStringCacheBuster` and `pyramid.static.ManifestCacheBuster`.

See <https://github.com/Pylons/pyramid/pull/2186>

Features

- Add a new `config.add_cache_buster` API for attaching cache busters to static assets. See <https://github.com/Pylons/pyramid/pull/2186>

Bug Fixes

- Ensure that `IAAssetDescriptor.abspath` always returns an absolute path. There were cases depending on the process CWD that a relative path would be returned. See <https://github.com/Pylons/pyramid/issues/2188>

1.6b2 (2015-10-15)

Features

- Allow asset specifications to be supplied to `pyramid.static.ManifestCacheBuster` instead of requiring a filesystem path.

1.6b1 (2015-10-15)

Backward Incompatibilities

- IPython and BPython support have been removed from pshell in the core. To continue using them on Pyramid 1.6+ you must install the binding packages explicitly:

```
$ pip install pyramid_ipython  
  
or  
  
$ pip install pyramid_bpython
```

- Remove default cache busters introduced in 1.6a1 including `PathSegmentCacheBuster`, `PathSegmentMd5CacheBuster`, and `QueryStringMd5CacheBuster`. See <https://github.com/Pylons/pyramid/pull/2116>

Features

- Additional shells for `pshell` can now be registered as entrypoints. See <https://github.com/Pylons/pyramid/pull/1891> and <https://github.com/Pylons/pyramid/pull/2012>
- The variables injected into `pshell` are now displayed with their docstrings instead of the default `str(obj)` when possible. See <https://github.com/Pylons/pyramid/pull/1929>
- Add new `pyramid.static.ManifestCacheBuster` for use with external asset pipelines as well as examples of common usages in the narrative. See <https://github.com/Pylons/pyramid/pull/2116>
- Fix `pserve --reload` to not crash on syntax errors!!! See <https://github.com/Pylons/pyramid/pull/2125>
- Fix an issue when user passes unparsed strings to `pyramid.session.CookieSession` and `pyramid.authentication.AuthTktCookieHelper` for time related parameters `timeout`, `reissue_time`, `max_age` that expect an integer value. See <https://github.com/Pylons/pyramid/pull/2050>

Bug Fixes

- `pyramid.httpexceptions.HTTPException` now defaults to 520 Unknown Error instead of None None to conform with changes in WebOb 1.5. See <https://github.com/Pylons/pyramid/pull/1865>
- `pshell` will now preserve the capitalization of variables in the `[pshell]` section of the INI file. This makes exposing classes to the shell a little more straightforward. See <https://github.com/Pylons/pyramid/pull/1883>
- Fixed usage of `pserve --monitor-restart --daemon` which would fail in horrible ways. See <https://github.com/Pylons/pyramid/pull/2118>
- Explicitly prevent `pserve --reload --daemon` from being used. It's never been supported but would work and fail in weird ways. See <https://github.com/Pylons/pyramid/pull/2119>
- Fix an issue on Windows when running `pserve --reload` in which the process failed to fork because it could not find the `pserve` script to run. See <https://github.com/Pylons/pyramid/pull/2138>

Deprecations

- Deprecate `pserve --monitor-restart` in favor of user's using a real process manager such as Systemd or Upstart as well as Python-based solutions like Circus and Supervisor. See <https://github.com/Pylons/pyramid/pull/2120>

1.6a2 (2015-06-30)

Bug Fixes

- Ensure that `pyramid.httpexceptions.exception_response` returns the appropriate “concrete” class for 400 and 500 status codes. See <https://github.com/Pylons/pyramid/issues/1832>
- Fix an infinite recursion bug introduced in 1.6a1 when `pyramid.view.render_view_to_response` was called directly or indirectly. See <https://github.com/Pylons/pyramid/issues/1643>
- Further fix the JSONP renderer by prefixing the returned content with a comment. This should mitigate attacks from Flash (See CVE-2014-4671). See <https://github.com/Pylons/pyramid/pull/1649>
- Allow periods and brackets ([]) in the JSONP callback. The original fix was overly-restrictive and broke Angular. See <https://github.com/Pylons/pyramid/pull/1649>

1.6a1 (2015-04-15)

Features

- `pcreate` will now ask for confirmation if invoked with an argument for a project name that already exists or is importable in the current environment. See <https://github.com/Pylons/pyramid/issues/1357> and <https://github.com/Pylons/pyramid/pull/1837>
- Make it possible to subclass `pyramid.request.Request` and also use `pyramid.request.Request.add_request.method`. See <https://github.com/Pylons/pyramid/issues/1529>

- The `pyramid.config.Configurator` has grown the ability to allow actions to call other actions during a commit-cycle. This enables much more logic to be placed into actions, such as the ability to invoke other actions or group them for improved conflict detection. We have also exposed and documented the config phases that Pyramid uses in order to further assist in building conforming addons. See <https://github.com/Pylons/pyramid/pull/1513>
- Add `pyramid.request.apply_request_extensions` function which can be used in testing to apply any request extensions configured via `config.add_request_method`. Previously it was only possible to test the extensions by going through Pyramid's router. See <https://github.com/Pylons/pyramid/pull/1581>
- `pcreate` when run without a scaffold argument will now print information on the missing flag, as well as a list of available scaffolds. See <https://github.com/Pylons/pyramid/pull/1566> and <https://github.com/Pylons/pyramid/issues/1297>
- Added support / testing for 'pypy3' under Tox and Travis. See <https://github.com/Pylons/pyramid/pull/1469>
- Automate code coverage metrics across py2 and py3 instead of just py2. See <https://github.com/Pylons/pyramid/pull/1471>
- Cache busting for static resources has been added and is available via a new argument to `pyramid.config.Configurator.add_static_view: cachebust`. Core APIs are shipped for both cache busting via query strings and path segments and may be extended to fit into custom asset pipelines. See <https://github.com/Pylons/pyramid/pull/1380> and <https://github.com/Pylons/pyramid/pull/1583>
- Add `pyramid.config.Configurator.root_package` attribute and `init` parameter to assist with includeable packages that wish to resolve resources relative to the package in which the `Configurator` was created. This is especially useful for addons that need to load asset specs from settings, in which case it is may be natural for a developer to define imports or assets relative to the top-level package. See <https://github.com/Pylons/pyramid/pull/1337>
- Added line numbers to the log formatters in the scaffolds to assist with debugging. See <https://github.com/Pylons/pyramid/pull/1326>
- Add new HTTP exception objects for status codes 428 Precondition Required, 429 Too Many Requests and 431 Request Header Fields Too Large in `pyramid.httpexceptions`. See <https://github.com/Pylons/pyramid/pull/1372/files>
- The `pshell` script will now load a `PYTHONSTARTUP` file if one is defined in the environment prior to launching the interpreter. See <https://github.com/Pylons/pyramid/pull/1448>

- Make it simple to define `notfound` and `forbidden` views that wish to use the default exception-response view but with altered predicates and other configuration options. The `view` argument is now optional in `config.add_notfound_view` and `config.add_forbidden_view`. See <https://github.com/Pylons/pyramid/issues/494>
- Greatly improve the readability of the `pcreate` shell script output. See <https://github.com/Pylons/pyramid/pull/1453>
- Improve robustness to timing attacks in the `AuthTktCookieHelper` and the `SignedCookieSessionFactory` classes by using the `stdlib`'s `hmac.compare_digest` if it is available (such as Python 2.7.7+ and 3.3+). See <https://github.com/Pylons/pyramid/pull/1457>
- Assets can now be overridden by an absolute path on the filesystem when using the `config.override_asset` API. This makes it possible to fully support serving up static content from a mutable directory while still being able to use the `request.static_url` API and `config.add_static_view`. Previously it was not possible to use `config.add_static_view` with an absolute path **and** generate urls to the content. This change replaces the call, `config.add_static_view('/abs/path', 'static')`, with `config.add_static_view('myapp:static', 'static')` and `config.override_asset(to_override='myapp:static/', override_with='/abs/path/')`. The `myapp:static` asset spec is completely made up and does not need to exist - it is used for generating urls via `request.static_url('myapp:static/foo.png')`. See <https://github.com/Pylons/pyramid/issues/1252>
- Added `pyramid.config.Configurator.set_response_factory` and the `response_factory` keyword argument to the `Configurator` for defining a factory that will return a custom `Response` class. See <https://github.com/Pylons/pyramid/pull/1499>
- Allow an iterator to be returned from a renderer. Previously it was only possible to return bytes or unicode. See <https://github.com/Pylons/pyramid/pull/1417>
- `pserve` can now take a `-b` or `--browser` option to open the server URL in a web browser. See <https://github.com/Pylons/pyramid/pull/1533>
- Overall improvements for the `proutes` command. Added `--format` and `--glob` arguments to the command, introduced the `method` column for displaying available request methods, and improved the `view` output by showing the module instead of just `__repr__`. See <https://github.com/Pylons/pyramid/pull/1488>
- Support keyword-only arguments and function annotations in views in Python 3. See <https://github.com/Pylons/pyramid/pull/1556>

- `request.response` will no longer be mutated when using the `pyramid.renderers.render_to_response()` API. It is now necessary to pass in a `response=` argument to `render_to_response` if you wish to supply the renderer with a custom response object for it to use. If you do not pass one then a response object will be created using the application's `IResponseFactory`. Almost all renderers mutate the `request.response` response object (for example, the JSON renderer sets `request.response.content_type` to `application/json`). However, when invoking `render_to_response` it is not expected that the response object being returned would be the same one used later in the request. The response object returned from `render_to_response` is now explicitly different from `request.response`. This does not change the API of a renderer. See <https://github.com/Pylons/pyramid/pull/1563>
- The `append_slash` argument of `Configurator().add_notfound_view()` will now accept anything that implements the `IResponse` interface and will use that as the response class instead of the default `HTTPFound`. See <https://github.com/Pylons/pyramid/pull/1610>

Bug Fixes

- The JSONP renderer created JavaScript code in such a way that a callback variable could be used to arbitrarily inject javascript into the response object. <https://github.com/Pylons/pyramid/pull/1627>
- Work around an issue where `pserve --reload` would leave terminal echo disabled if it reloaded during a `pdb` session. See <https://github.com/Pylons/pyramid/pull/1577>, <https://github.com/Pylons/pyramid/pull/1592>
- `pyramid.wsgi.wsgiapp` and `pyramid.wsgi.wsgiapp2` now raise `ValueError` when accidentally passed `None`. See <https://github.com/Pylons/pyramid/pull/1320>
- Fix an issue whereby predicates would be resolved as `maybe_dotted` in the introspectable but not when passed for registration. This would mean that `add_route_predicate` for example can not take a string and turn it into the actual callable function. See <https://github.com/Pylons/pyramid/pull/1306>
- Fix `pyramid.testing.setUp` to return a `Configurator` with a proper package. Previously it was not possible to do package-relative includes using the returned `Configurator` during testing. There is now a `package` argument that can override this behavior as well. See <https://github.com/Pylons/pyramid/pull/1322>
- Fix an issue where a `pyramid.response.FileResponse` may apply a charset where it does not belong. See <https://github.com/Pylons/pyramid/pull/1251>

- Work around a bug introduced in Python 2.7.7 on Windows where `mimetypes.guess_type` returns Unicode rather than str for the content type, unlike any previous version of Python. See <https://github.com/Pylons/pyramid/issues/1360> for more information.
- `pcreate` now normalizes the package name by converting hyphens to underscores. See <https://github.com/Pylons/pyramid/pull/1376>
- Fix an issue with the final response/finished callback being unable to add another callback to the list. See <https://github.com/Pylons/pyramid/pull/1373>
- Fix a failing unittest caused by differing mimetypes across various OSs. See <https://github.com/Pylons/pyramid/issues/1405>
- Fix route generation for static view asset specifications having no path. See <https://github.com/Pylons/pyramid/pull/1377>
- Allow the `pyramid.renderers.JSONP` renderer to work even if there is no valid request object. In this case it will not wrap the object in a callback and thus behave just like the `pyramid.renderers.JSON` renderer. See <https://github.com/Pylons/pyramid/pull/1561>
- Prevent “parameters to load are deprecated” `DeprecationWarning` from `setuptools>=11.3`. See <https://github.com/Pylons/pyramid/pull/1541>
- Avoiding sharing the `IRenderer` objects across threads when attached to a view using the `render=` argument. These renderers were instantiated at time of first render and shared between requests, causing potentially subtle effects like `pyramid.reload_templates = true` failing to work in `pyramid_mako`. See <https://github.com/Pylons/pyramid/pull/1575> and <https://github.com/Pylons/pyramid/issues/1268>
- Avoiding timing attacks against CSRF tokens. See <https://github.com/Pylons/pyramid/pull/1574>
- `request.finished_callbacks` and `request.response_callbacks` now default to an iterable instead of `None`. It may be checked for a length of 0. This was the behavior in 1.5.

Deprecations

- The `pserve` command’s daemonization features have been deprecated. This includes the `[start, stop, restart, status]` subcommands as well as the `--daemon`, `--stop-server`, `--pid-file`, and `--status` flags.

Please use a real process manager in the future instead of relying on the `pserve` to daemonize itself. Many options exist including your Operating System’s services such as `Systemd` or `Upstart`, as well as Python-based solutions like `Circus` and `Supervisor`.

See <https://github.com/Pylons/pyramid/pull/1641>

- Renamed the principal argument to `pyramid.security.remember()` to `userid` in order to clarify its intended purpose. See <https://github.com/Pylons/pyramid/pull/1399>

Docs

- Moved the documentation for `accept` on `Configurator.add_view` to no longer be part of the predicate list. See <https://github.com/Pylons/pyramid/issues/1391> for a bug report stating `not_` was failing on `accept`. Discussion with @mcdonc led to the conclusion that it should not be documented as a predicate. See <https://github.com/Pylons/pyramid/pull/1487> for this PR
- Removed logging configuration from Quick Tutorial ini files except for scaffolding- and logging-related chapters to avoid needing to explain it too early.
- Clarify a previously-implied detail of the `ISession.invalidate` API documentation.
- Improve and clarify the documentation on what Pyramid defines as a `principal` and a `userid` in its security APIs. See <https://github.com/Pylons/pyramid/pull/1399>
- Add documentation of command line programs (p* scripts). See <https://github.com/Pylons/pyramid/pull/2191>

Scaffolds

- Update scaffold generating machinery to return the version of pyramid and pyramid docs for use in scaffolds. Updated starter, alchemy and zodb templates to have links to correctly versioned documentation and reflect which pyramid was used to generate the scaffold.
- Removed non-ascii copyright symbol from templates, as this was causing the scaffolds to fail for project generation.
- You can now run the scaffolding func tests via `tox py2-scaffolds` and `tox py3-scaffolds`.

1.5 (2014-04-08)

- Python 3.4 compatibility.
- Avoid crash in `pserve --reload` under Py3k, when iterating over possibly mutated `sys.modules`.
- `UnencryptedCookieSessionFactoryConfig` failed if the secret contained higher order characters. See <https://github.com/Pylons/pyramid/issues/1246>
- Fixed a bug in `UnencryptedCookieSessionFactoryConfig` and `SignedCookieSessionFactory` where `timeout=None` would cause a new session to always be created. Also in `SignedCookieSessionFactory` a `reissue_time=None` would cause an exception when modifying the session. See <https://github.com/Pylons/pyramid/issues/1247>
- Updated docs and scaffolds to keep in step with new 2.0 release of `Lingua`. This included removing all `setup.cfg` files from scaffolds and documentation environments.

1.5b1 (2014-02-08)**Features**

- We no longer eagerly clear `request.exception` and `request.exc_info` in the exception view tween. This makes it possible to inspect exception information within a finished callback. See <https://github.com/Pylons/pyramid/issues/1223>.

1.5a4 (2014-01-28)**Features**

- Updated scaffolds with new theme, fixed documentation and sample project.

Bug Fixes

- Depend on a newer version of WebOb so that we pull in some crucial bug-fixes that were show-stoppers for functionality in Pyramid.
- Add a trailing semicolon to the JSONP response. This fixes JavaScript syntax errors for old IE versions. See <https://github.com/Pylons/pyramid/pull/1205>
- Fix a memory leak when the configurator's `set_request_property` method was used or when the configurator's `add_request_method` method was used with the `property=True` attribute. See <https://github.com/Pylons/pyramid/issues/1212>.

1.5a3 (2013-12-10)**Features**

- An authorization API has been added as a method of the request: `request.has_permission`.

`request.has_permission` is a method-based alternative to the `pyramid.security.has_permission` API and works exactly the same. The older API is now deprecated.

- Property API attributes have been added to the request for easier access to authentication data: `request.authenticated_userid`, `request.unauthenticated_userid`, and `request.effective_principals`.

These are analogues, respectively, of `pyramid.security.authenticated_userid`, `pyramid.security.unauthenticated_userid`, and `pyramid.security.effective_principals`. They operate exactly the same, except they are attributes of the request instead of functions accepting a request. They are properties, so they cannot be assigned to. The older function-based APIs are now deprecated.

- Pyramid's console scripts (`pserve`, `pviews`, etc) can now be run directly, allowing custom arguments to be sent to the python interpreter at runtime. For example:

```
python -3 -m pyramid.scripts.pserve development.ini
```

- Added a specific subclass of `HTTPBadRequest` named `pyramid.exceptions.BadCSRFToken` which will now be raised in response to failures in `check_csrf_token`. See <https://github.com/Pylons/pyramid/pull/1149>
- Added a new `SignedCookieSessionFactory` which is very similar to the `UnencryptedCookieSessionFactoryConfig` but with a clearer focus on signing content. The custom serializer arguments to this function should only focus on serializing, unlike its predecessor which required the serializer to also perform signing. See <https://github.com/Pylons/pyramid/pull/1142> . Note that cookies generated using `SignedCookieSessionFactory` are not compatible with cookies generated using `UnencryptedCookieSessionFactory`, so existing user session data will be destroyed if you switch to it.
- Added a new `BaseCookieSessionFactory` which acts as a generic cookie factory that can be used by framework implementors to create their own session implementations. It provides a reusable API which focuses strictly on providing a dictionary-like object that properly handles renewals, timeouts, and conformance with the `ISession` API. See <https://github.com/Pylons/pyramid/pull/1142>
- The anchor argument to `pyramid.request.Request.route_url` and `pyramid.request.Request.resource_url` and their derivatives will now be escaped via URL quoting to ensure minimal conformance. See <https://github.com/Pylons/pyramid/pull/1183>
- Allow sending of `_query` and `_anchor` options to `pyramid.request.Request.static_url` when an external URL is being generated. See <https://github.com/Pylons/pyramid/pull/1183>

- You can now send a string as the `_query` argument to `pyramid.request.Request.route_url` and `pyramid.request.Request.resource_url` and their derivatives. When a string is sent instead of a list or dictionary, it is URL-quoted however it does not need to be in `k=v` form. This is useful if you want to be able to use a different query string format than `x-www-form-urlencoded`. See <https://github.com/Pylons/pyramid/pull/1183>
- `pyramid.testing.DummyRequest` now has a `domain` attribute to match the new WebOb 1.3 API. Its value is `example.com`.

Bug Fixes

- Fix the `pcreate` script so that when the target directory name ends with a slash it does not produce a non-working project directory structure. Previously saying `pcreate -s starter /foo/bar/` produced different output than saying `pcreate -s starter /foo/bar`. The former did not work properly.
- Fix the `principals_allowed_by_permission` method of `ACLAuthorizationPolicy` so it anticipates a callable `__acl__` on resources. Previously it did not try to call the `__acl__` if it was callable.
- The `pviews` script did not work when a url required custom request methods in order to perform traversal. Custom methods and descriptors added via `pyramid.config.Configurator.add_request_method` will now be present, allowing traversal to continue. See <https://github.com/Pylons/pyramid/issues/1104>
- Remove unused `renderer` argument from `Configurator.add_route`.
- Allow the `BasicAuthenticationPolicy` to work with non-ascii usernames and passwords. The charset is not passed as part of the header and different browsers alternate between UTF-8 and Latin-1, so the policy now attempts to decode with UTF-8 first, and will fallback to Latin-1. See <https://github.com/Pylons/pyramid/pull/1170>
- The `@view_defaults` now apply to notfound and forbidden views that are defined as methods of a decorated class. See <https://github.com/Pylons/pyramid/issues/1173>

Documentation

- Added a “Quick Tutorial” to go with the Quick Tour
- Removed mention of `pyramid_beaker` from docs. Beaker is no longer maintained. Point people at `pyramid_redis_sessions` instead.
- Add documentation for `pyramid.interfaces.IRendererFactory` and `pyramid.interfaces.IRenderer`.

Backwards Incompatibilities

- The key/values in the `_query` parameter of `request.route_url` and the query parameter of `request.resource_url` (and their variants), used to encode a value of `None` as the string `'None'`, leaving the resulting query string to be `a=b&key=None`. The value is now dropped in this situation, leaving a query string of `a=b&key=`. See <https://github.com/Pylons/pyramid/issues/1119>

Deprecations

- Deprecate the `pyramid.interfaces.ITemplateRenderer` interface. It was ill-defined and became unused when Mako and Chameleon template bindings were split into their own packages.
- The `pyramid.session.UnencryptedCookieSessionFactoryConfig` API has been deprecated and is superseded by the `pyramid.session.SignedCookieSessionFactory`. Note that while the cookies generated by the `UnencryptedCookieSessionFactoryConfig` are compatible with cookies generated by old releases, cookies generated by the `SignedCookieSessionFactory` are not. See <https://github.com/Pylons/pyramid/pull/1142>
- The `pyramid.security.has_permission` API is now deprecated. Instead, use the newly-added `has_permission` method of the request object.
- The `pyramid.security.effective_principals` API is now deprecated. Instead, use the newly-added `effective_principals` attribute of the request object.
- The `pyramid.security.authenticated_userid` API is now deprecated. Instead, use the newly-added `authenticated_userid` attribute of the request object.
- The `pyramid.security.unauthenticated_userid` API is now deprecated. Instead, use the newly-added `unauthenticated_userid` attribute of the request object.

Dependencies

- Pyramid now depends on `WebOb>=1.3` (it uses `webob.cookies.CookieProfile` from 1.3+).

1.5a2 (2013-09-22)

Features

- Users can now provide dotted Python names to as the `factory` argument the Configurator methods named `add_{view,route,subscriber}_predicate` (instead of passing the predicate factory directly, you can pass a dotted name which refers to the factory).

Bug Fixes

- Fix an exception in `pyramid.path.package_name` when resolving the package name for namespace packages that had no `__file__` attribute.

Backwards Incompatibilities

- Pyramid no longer depends on or configures the Mako and Chameleon templating system renderers by default. Disincluding these templating systems by default means that the Pyramid core has fewer dependencies and can run on future platforms without immediate concern for the compatibility of its templating add-ons. It also makes maintenance slightly more effective, as different people can maintain the templating system add-ons that they understand and care about without needing commit access to the Pyramid core, and it allows users who just don't want to see any packages they don't use come along for the ride when they install Pyramid.

This means that upon upgrading to Pyramid 1.5a2+, projects that use either of these templating systems will see a traceback that ends something like this when their application attempts to render a Chameleon or Mako template:

```
ValueError: No such renderer factory .pt
```

Or:

```
ValueError: No such renderer factory .mako
```

Or:

```
ValueError: No such renderer factory .mak
```

Support for Mako templating has been moved into an add-on package named `pyramid_mako`, and support for Chameleon templating has been moved into an add-on package named `pyramid_chameleon`. These packages are drop-in replacements for the old built-in support for these templating languages. All you have to do is install them and make them active in your configuration to register renderer factories for `.pt` and/or `.mako` (or `.mak`) to make your application work again.

To re-add support for Chameleon and/or Mako template renderers into your existing projects, follow the below steps.

If you depend on Mako templates:

- Make sure the `pyramid_mako` package is installed. One way to do this is by adding `pyramid_mako` to the `install_requires` section of your package's `setup.py` file and afterwards rerunning `setup.py develop`:

```
setup(
    #...
    install_requires=[
        'pyramid_mako',          # new dependency
        'pyramid',
        #...
    ],
)
```

- Within the portion of your application which instantiates a Pyramid `pyramid.config.Configurator` (often the `main()` function in your project's `__init__.py` file), tell Pyramid to include the `pyramid_mako` includeme:

```
config = Configurator(...)
config.include('pyramid_mako')
```

If you depend on Chameleon templates:

- Make sure the `pyramid_chameleon` package is installed. One way to do this is by adding `pyramid_chameleon` to the `install_requires` section of your package's `setup.py` file and afterwards rerunning `setup.py develop`:

```

setup(
    #...
    install_requires=[
        'pyramid_chameleon',      # new dependency
        'pyramid',
        #...
    ],
)

```

- Within the portion of your application which instantiates a Pyramid `~pyramid.config.Configurator` (often the `main()` function in your project's `__init__.py` file), tell Pyramid to include the `pyramid_chameleon` includeme:

```

config = Configurator(...)
config.include('pyramid_chameleon')

```

Note that it's also fine to install these packages into *older* Pyramids for forward compatibility purposes. Even if you don't upgrade to Pyramid 1.5 immediately, performing the above steps in a Pyramid 1.4 installation is perfectly fine, won't cause any difference, and will give you forward compatibility when you eventually do upgrade to Pyramid 1.5.

With the removal of Mako and Chameleon support from the core, some unit tests that use the `pyramid.renderers.render*` methods may begin to fail. If any of your unit tests are invoking either `pyramid.renderers.render()` or `pyramid.renderers.render_to_response()` with either Mako or Chameleon templates then the `pyramid.config.Configurator` instance in effect during the unit test should be also be updated to include the addons, as shown above. For example:

```

class ATest(unittest.TestCase):
    def setUp(self):
        self.config = pyramid.testing.setUp()
        self.config.include('pyramid_mako')

    def test_it(self):
        result = pyramid.renderers.render('mypkg:templates/home.mako',
        ↪ {})

```

Or:

```

class ATest(unittest.TestCase):
    def setUp(self):

```

```
self.config = pyramid.testing.setUp()
self.config.include('pyramid_chameleon')

def test_it(self):
    result = pyramid.renderers.render('mypkg:templates/home.pt', {})
    ↪)
```

- If you're using the Pyramid debug toolbar, when you upgrade Pyramid to 1.5a2+, you'll also need to upgrade the `pyramid_debugtoolbar` package to at least version 1.0.8, as older toolbar versions are not compatible with Pyramid 1.5a2+ due to the removal of Mako support from the core. It's fine to use this newer version of the toolbar code with older Pyramids too.
- Removed the `request.response_*` varying attributes. These attributes have been deprecated since Pyramid 1.1, and as per the deprecation policy, have now been removed.
- `request.response` will no longer be mutated when using the `pyramid.renderers.render()` API. Almost all renderers mutate the `request.response` response object (for example, the JSON renderer sets `request.response.content_type` to `application/json`), but this is only necessary when the renderer is generating a response; it was a bug when it was done as a side effect of calling `pyramid.renderers.render()`.
- Removed the `bfg2pyramid` fixer script.
- The `pyramid.events.NewResponse` event is now sent **after** response callbacks are executed. It previously executed before response callbacks were executed. Rationale: it's more useful to be able to inspect the response after response callbacks have done their jobs instead of before.
- Removed the class named `pyramid.view.static` that had been deprecated since Pyramid 1.1. Instead use `pyramid.static.static_view` with `use_subpath=True` argument.
- Removed the `pyramid.view.is_response` function that had been deprecated since Pyramid 1.1. Use the `pyramid.request.Request.is_response` method instead.
- Removed the ability to pass the following arguments to `pyramid.config.Configurator.add_route`: `view`, `view_context`, `view_for`, `view_permission`, `view_renderer`, and `view_attr`. Using these arguments had been deprecated since Pyramid 1.1. Instead of passing view-related arguments to `add_route`, use a separate call to `pyramid.config.Configurator.add_view` to associate a view with a route using its `route_name` argument. Note that this impacts the `pyramid.config.Configurator.add_static_view` function too, because it delegates to `add_route`.
- Removed the ability to influence and query a `pyramid.request.Request` object as if it were a dictionary. Previously it was possible to use methods like `__getitem__`, `get`, `items`, and other dictlike methods to access values in the WSGI environment. This behavior had been deprecated since Pyramid 1.1. Use methods of `request.environ` (a real dictionary) instead.

- Removed ancient backwards compatibility hack in `pyramid.traversal.DefaultRootFactory` which populated the `__dict__` of the factory with the `matchdict` values for compatibility with BFG 0.9.
- The `renderer_globals_factory` argument to the `pyramid.config.Configurator` constructor and its ```setup_registry` method has been removed. The `set_renderer_globals_factory` method of `pyramid.config.Configurator` has also been removed. The (internal) `pyramid.interfaces.IRendererGlobals` interface was also removed. These arguments, methods and interfaces had been deprecated since 1.1. Use a `BeforeRender` event subscriber as documented in the “Hooks” chapter of the Pyramid narrative documentation instead of providing renderer globals values to the configurator.

Deprecations

- The `pyramid.config.Configurator.set_request_property` method now issues a deprecation warning when used. It had been docs-deprecated in 1.4 but did not issue a deprecation warning when used.

1.5a1 (2013-08-30)

Features

- A new http exception subclass named `pyramid.httpexceptions.HTTPSuccessful` was added. You can use this class as the context of an exception view to catch all 200-series “exceptions” (e.g. “raise `HTTPOk`”). This also allows you to catch *only* the `HTTPOk` exception itself; previously this was impossible because a number of other exceptions (such as `HTTPNoContent`) inherited from `HTTPOk`, but now they do not.
- You can now generate “hybrid” `urldispatch/traversal` URLs more easily by using the new `route_name`, `route_kw` and `route_remainder_name` arguments to `request.resource_url` and `request.resource_path`. See the new section of the “Combining Traversal and URL Dispatch” documentation chapter entitled “Hybrid URL Generation”.
- It is now possible to escape double braces in Pyramid scaffolds (unescaped, these represent replacement values). You can use `\\{a\\}` to represent a “bare” `{a}`. See <https://github.com/Pylons/pyramid/pull/862>

- Add `localizer` and `locale_name` properties (reified) to the request. See <https://github.com/Pylons/pyramid/issues/508>. Note that the `pyramid.i18n.get_localizer` and `pyramid.i18n.get_locale_name` functions now simply look up these properties on the request.
- Add `pdistreport` script, which prints the Python version in use, the Pyramid version in use, and the version number and location of all Python distributions currently installed.
- Add the ability to invert the result of any view, route, or subscriber predicate using the `not_` class. For example:

```
from pyramid.config import not_  
  
@view_config(route_name='myroute', request_method=not_('POST'))  
def myview(request): ...
```

The above example will ensure that the view is called if the request method is not POST (at least if no other view is more specific).

The `pyramid.config.not_` class can be used against any value that is a predicate value passed in any of these contexts:

- `pyramid.config.Configurator.add_view`
 - `pyramid.config.Configurator.add_route`
 - `pyramid.config.Configurator.add_subscriber`
 - `pyramid.view.view_config`
 - `pyramid.events.subscriber`
- `scripts/prequest.py`: add support for submitting PUT and PATCH requests. See <https://github.com/Pylons/pyramid/pull/1033>. add support for submitting OPTIONS and PROPFIND requests, and allow users to specify basic authentication credentials in the request via a `--login` argument to the script. See <https://github.com/Pylons/pyramid/pull/1039>.
 - `ACLAuthorizationPolicy` supports `__acl__` as a callable. This removes the ambiguity between the potential `AttributeError` that would be raised on the context when the property was not defined and the `AttributeError` that could be raised from any user-defined code within a dynamic property. It is recommended to define a dynamic ACL as a callable to avoid this ambiguity. See <https://github.com/Pylons/pyramid/issues/735>.

- Allow a protocol-relative URL (e.g. `//example.com/images`) to be passed to `pyramid.config.Configurator.add_static_view`. This allows externally-hosted static URLs to be generated based on the current protocol.
- The `AuthTktAuthenticationPolicy` has two new options to configure its domain usage:
 - `parent_domain`: if set the authentication cookie is set on the parent domain. This is useful if you have multiple sites sharing the same domain.
 - `domain`: if provided the cookie is always set for this domain, bypassing all usual logic.

See <https://github.com/Pylons/pyramid/pull/1028>, <https://github.com/Pylons/pyramid/pull/1072> and <https://github.com/Pylons/pyramid/pull/1078>.

- The `AuthTktAuthenticationPolicy` now supports IPv6 addresses when using the `include_ip=True` option. This is possibly incompatible with alternative `auth_tkt` implementations, as the specification does not define how to properly handle IPv6. See <https://github.com/Pylons/pyramid/issues/831>.
- Make it possible to use variable arguments via `pyramid.paster.get_appsettings`. This also allowed the generated `initialize_db` script from the alchemy scaffold to grow support for options in the form `a=1 b=2` so you can fill in values in a parameterized `.ini` file, e.g. `initialize_myapp_db etc/development.ini a=1 b=2`. See <https://github.com/Pylons/pyramid/pull/911>
- The `request.session.check_csrf_token()` method and the `check_csrf` view predicate now take into account the value of the HTTP header named `X-CSRF-Token` (as well as the `csrf_token` form parameter, which they always did). The header is tried when the form parameter does not exist.
- View lookup will now search for valid views based on the inheritance hierarchy of the context. It tries to find views based on the most specific context first, and upon predicate failure, will move up the inheritance chain to test views found by the super-type of the context. In the past, only the most specific type containing views would be checked and if no matching view could be found then a `PredicateMismatch` would be raised. Now predicate mismatches don't hide valid views registered on super-types. Here's an example that now works:

```
class IResource(Interface):  
  
    ...  
  
@view_config(context=IResource)  
def get(context, request):
```

```
...

@view_config(context=IResource, request_method='POST')
def post(context, request):

    ...

@view_config(context=IResource, request_method='DELETE')
def delete(context, request):

    ...

@implementer(IResource)
class MyResource:

    ...

@view_config(context=MyResource, request_method='POST')
def override_post(context, request):

    ...
```

Previously the `override_post` view registration would hide the `get` and `delete` views in the context of `MyResource` – leading to a predicate mismatch error when trying to use `GET` or `DELETE` methods. Now the views are found and no predicate mismatch is raised. See <https://github.com/Pylons/pyramid/pull/786> and <https://github.com/Pylons/pyramid/pull/1004> and <https://github.com/Pylons/pyramid/pull/1046>

- The `pserve` command now takes a `-v` (or `--verbose`) flag and a `-q` (or `--quiet`) flag. Output from running `pserve` can be controlled using these flags. `-v` can be specified multiple times to increase verbosity. `-q` sets verbosity to 0 unconditionally. The default verbosity level is 1.
- The `alchemy` scaffold tests now provide better coverage. See <https://github.com/Pylons/pyramid/pull/1029>
- The `pyramid.config.Configurator.add_route` method now supports being called with an external URL as pattern. See <https://github.com/Pylons/pyramid/issues/611> and the documentation section in the “URL Dispatch” chapter entitled “External Routes” for more information.

Bug Fixes

- It was not possible to use `pyramid.httpexceptions.HTTPException` as the context of an exception view as very general catchall for http-related exceptions when you wanted that exception view to override the default exception view. See <https://github.com/Pylons/pyramid/issues/985>

- When the `pyramid.reload_templates` setting was true, and a Chameleon template was reloaded, and the renderer specification named a macro (e.g. `foo#macroname.pt`), renderings of the template after the template was reloaded due to a file change would produce the entire template body instead of just a rendering of the macro. See <https://github.com/Pylons/pyramid/issues/1013>.
- Fix an obscure problem when combining a virtual root with a route with a `*traverse` in its pattern. Now the traversal path generated in such a configuration will be correct, instead of an element missing a leading slash.
- Fixed a Mako renderer bug returning a tuple with a previous `defname` value in some circumstances. See <https://github.com/Pylons/pyramid/issues/1037> for more information.
- Make the `pyramid.config.assets.PackageOverrides` object implement the API for `__loader__` objects specified in PEP 302. Proxies to the `__loader__` set by the importer, if present; otherwise, raises `NotImplementedError`. This makes Pyramid static view overrides work properly under Python 3.3 (previously they would not). See <https://github.com/Pylons/pyramid/pull/1015> for more information.
- `mako_templating`: added defensive workaround for non-importability of `mako` due to upstream `markupsafe` dropping Python 3.2 support. Mako templating will no longer work under the combination of MarkupSafe 0.17 and Python 3.2 (although the combination of MarkupSafe 0.17 and Python 3.3 or any supported Python 2 version will work OK).
- Spaces and dots may now be in mako renderer template paths. This was broken when support for the new `makodef` syntax was added in 1.4a1. See <https://github.com/Pylons/pyramid/issues/950>
- `pyramid.debug_authorization=true` will now correctly print out `Allowed` for views registered with `NO_PERMISSION_REQUIRED` instead of invoking the `permits` method of the authorization policy. See <https://github.com/Pylons/pyramid/issues/954>
- Pyramid failed to install on some systems due to being packaged with some test files containing higher order characters in their names. These files have now been removed. See <https://github.com/Pylons/pyramid/issues/981>
- `pyramid.testing.DummyResource` didn't define `__bool__`, so code under Python 3 would use `__len__` to find truthiness; this usually caused an instance of `DummyResource` to be "falsy" instead of "truthy". See <https://github.com/Pylons/pyramid/pull/1032>
- The `alchemy` scaffold would break when the database was MySQL during tables creation. See <https://github.com/Pylons/pyramid/pull/1049>
- The `current_route_url` method now attaches the query string to the URL by default. See <https://github.com/Pylons/pyramid/issues/1040>
- Make `pserve.cherryserver_runner` Python 3 compatible. See <https://github.com/Pylons/pyramid/issues/718>

Backwards Incompatibilities

- Modified the `current_route_url` method in `pyramid.Request`. The method previously returned the URL without the query string by default, it now does attach the query string unless it is overridden.
- The `route_url` and `route_path` APIs no longer quote `/` to `%2F` when a replacement value contains a `/`. This was pointless, as WSGI servers always unquote the slash anyway, and Pyramid never sees the quoted value.
- It is no longer possible to set a `locale_name` attribute of the request, nor is it possible to set a `localizer` attribute of the request. These are now “reified” properties that look up a locale name and localizer respectively using the machinery described in the “Internationalization” chapter of the documentation.
- If you send an `X-Vhm-Root` header with a value that ends with a slash (or any number of slashes), the trailing slash(es) will be removed before a URL is generated when you use `request.resource_url` or `request.resource_path`. Previously the virtual root path would not have trailing slashes stripped, which would influence URL generation.
- The `pyramid.interfaces.IResourceURL` interface has now grown two new attributes: `virtual_path_tuple` and `physical_path_tuple`. These should be the tuple form of the resource’s path (physical and virtual).

1.4 (2012-12-18)

Docs

- Fix functional tests in the ZODB tutorial

1.4b3 (2012-12-10)

- Packaging release only, no code changes. 1.4b2 was a brownbag release due to missing directories in the tarball.

1.4b2 (2012-12-10)

Docs

- Scaffolding is now PEP-8 compliant (at least for a brief shining moment).
- Tutorial improvements.

Backwards Incompatibilities

- Modified the `_depth` argument to `pyramid.view.view_config` to accept a value relative to the invocation of `view_config` itself. Thus, when it was previously expecting a value of 1 or greater, to reflect that the caller of `view_config` is 1 stack frame away from `venusian.attach`, this implementation detail is now hidden.
- Modified the `_backframes` argument to `pyramid.util.action_method` in a similar way to the changes described to `_depth` above. This argument remains undocumented, but might be used in the wild by some insane person.

1.4b1 (2012-11-21)

Features

- Small microspeed enhancement which anticipates that a `pyramid.response.Response` object is likely to be returned from a view. Some code is shortcut if the class of the object returned by a view is this class. A similar microoptimization was done to `pyramid.request.Request.is_response`.
- Make it possible to use variable arguments on `p*` commands (`pserve`, `pshell`, `pviews`, etc) in the form `a=1 b=2` so you can fill in values in parameterized `.ini` file, e.g. `pshell etc/development.ini http_port=8080`. See <https://github.com/Pylons/pyramid/pull/714>
- A somewhat advanced and obscure feature of Pyramid event handlers is their ability to handle “multi-interface” notifications. These notifications have traditionally presented multiple objects to the subscriber callable. For instance, if an event was sent by code like this:

```
registry.notify(event, context)
```

In the past, in order to catch such an event, you were obligated to write and register an event subscriber that mentioned both the event and the context in its argument list:

```
@subscriber([SomeEvent, SomeContextType])
def asubscriber(event, context):
    pass
```

In many subscriber callables registered this way, it was common for the logic in the subscriber callable to completely ignore the second and following arguments (e.g. `context` in the above example might be ignored), because they usually existed as attributes of the event anyway. You could usually get the same value by doing `event.context` or similar.

The fact that you needed to put an extra argument which you usually ignored in the subscriber callable body was only a minor annoyance until we added “subscriber predicates”, used to narrow the set of circumstances under which a subscriber will be executed, in a prior 1.4 alpha release. Once those were added, the annoyance was escalated, because subscriber predicates needed to accept the same argument list and arity as the subscriber callables that they were configured against. So, for example, if you had these two subscriber registrations in your code:

```
@subscriber([SomeEvent, SomeContextType])
def asubscriber(event, context):
    pass

@subscriber(SomeOtherEvent)
def asubscriber(event):
    pass
```

And you wanted to use a subscriber predicate:

```
@subscriber([SomeEvent, SomeContextType], mypredicate=True)
def asubscriber1(event, context):
    pass

@subscriber(SomeOtherEvent, mypredicate=True)
def asubscriber2(event):
    pass
```

If an existing `mypredicate` subscriber predicate had been written in such a way that it accepted only one argument in its `__call__`, you could not use it against a subscription which named more

than one interface in its subscriber interface list. Similarly, if you had written a subscriber predicate that accepted two arguments, you couldn't use it against a registration that named only a single interface type.

For example, if you created this predicate:

```
class MyPredicate(object):
    # portions elided...
    def __call__(self, event):
        return self.val == event.context.foo
```

It would not work against a multi-interface-registered subscription, so in the above example, when you attempted to use it against `asubscriber1`, it would fail at runtime with a `TypeError`, claiming something was attempting to call it with too many arguments.

To hack around this limitation, you were obligated to design the `mypredicate` predicate to expect to receive in its `__call__` either a single `event` argument (a `SomeOtherEvent` object) *or* a pair of arguments (a `SomeEvent` object and a `SomeContextType` object), presumably by doing something like this:

```
class MyPredicate(object):
    # portions elided...
    def __call__(self, event, context=None):
        return self.val == event.context.foo
```

This was confusing and bad.

In order to allow people to ignore unused arguments to subscriber callables and to normalize the relationship between event subscribers and subscriber predicates, we now allow both subscribers and subscriber predicates to accept only a single `event` argument even if they've been subscribed for notifications that involve multiple interfaces. Subscribers and subscriber predicates that accept only one argument will receive the first object passed to `notify`; this is typically (but not always) the event object. The other objects involved in the subscription lookup will be discarded. You can now write an event subscriber that accepts only `event` even if it subscribes to multiple interfaces:

```
@subscriber([SomeEvent, SomeContextType])
def asubscriber(event):
    # this will work!
```

This prevents you from needing to match the subscriber callable parameters to the subscription type unnecessarily, especially when you don't make use of any argument in your subscribers except for the event object itself.

Note, however, that if the event object is not the first object in the call to `notify`, you'll run into trouble. For example, if `notify` is called with the context argument first:


```
registry.notify(context, event)
```

You won't be able to take advantage of the event-only feature. It will “work”, but the object received by your event handler won't be the event object, it will be the context object, which won't be very useful:

```
@subscriber([SomeContextType, SomeEvent])
def asubscriber(event):
    # bzzt! you'll be getting the context here as ``event``, and it'll
    # be useless
```

Existing multiple-argument subscribers continue to work without issue, so you should continue use those if your system notifies using multiple interfaces and the first interface is not the event interface. For example:

```
@subscriber([SomeContextType, SomeEvent])
def asubscriber(context, event):
    # this will still work!
```

The event-only feature makes it possible to use a subscriber predicate that accepts only a request argument within both multiple-interface subscriber registrations and single-interface subscriber registrations. You needn't make slightly different variations of predicates depending on the subscription type arguments. Instead, just write all your subscriber predicates so they only accept `event` in their `__call__` and they'll be useful across all registrations for subscriptions that use an event as their first argument, even ones which accept more than just `event`.

However, the same caveat applies to predicates as to subscriber callables: if you're subscribing to a multi-interface event, and the first interface is not the event interface, the predicate won't work properly. In such a case, you'll need to match the predicate `__call__` argument ordering and composition to the ordering of the interfaces. For example, if the registration for the subscription uses `[SomeContext, SomeEvent]`, you'll need to reflect that in the ordering of the parameters of the predicate's `__call__` method:

```
def __call__(self, context, event):
    return event.request.path.startswith(self.val)
```

tl;dr: 1) When using multi-interface subscriptions, always use the event type as the first subscription registration argument and 2) When 1 is true, use only `event` in your subscriber and subscriber predicate parameter lists, no matter how many interfaces the subscriber is notified with. This combination will result in the maximum amount of reusability of subscriber predicates and the least amount of thought on your part. Drink responsibly.

Bug Fixes

- A failure when trying to locate the attribute `__text__` on route and view predicates existed when the `debug_routematch` setting was true or when the `pviews` command was used. See <https://github.com/Pylons/pyramid/pull/727>

Documentation

- Sync up tutorial source files with the files that are rendered by the scaffold that each uses.

1.4a4 (2012-11-14)

Features

- `pyramid.authentication.AuthTktAuthenticationPolicy` has been updated to support newer hashing algorithms such as `sha512`. Existing applications should consider updating if possible for improved security over the default `md5` hashing.
- Added an `effective_principals` route and view predicate.
- Do not allow the `userid` returned from the `authenticated_userid` or the `userid` that is one of the list of principals returned by `effective_principals` to be either of the strings `system`, `Everyone` or `system.Authenticated` when any of the built-in authorization policies that live in `pyramid.authentication` are in use. These two strings are reserved for internal usage by Pyramid and they will not be accepted as valid `userids`.
- Slightly better debug logging from `pyramid.authentication.RepozeWho1AuthenticationPolicy`.
- `pyramid.security.view_execution_permitted` used to return `True` if no view could be found. It now raises a `TypeError` exception in that case, as it doesn't make sense to assert that a nonexistent view is execution-permitted. See <https://github.com/Pylons/pyramid/issues/299>.
- Allow a `_depth` argument to `pyramid.view.view_config`, which will permit limited composition reuse of the decorator by other software that wants to provide custom decorators that are much like `view_config`.
- Allow an iterable of decorators to be passed to `pyramid.config.Configurator.add_view`. This allows views to be wrapped by more than one decorator without requiring combining the decorators yourself.

Bug Fixes

- In the past if a renderer returned `None`, the body of the resulting response would be set explicitly to the empty string. Instead, now, the body is left unchanged, which allows the renderer to set a body itself by using e.g. `request.response.body = b'foo'`. The body set by the renderer will be unmolested on the way out. See <https://github.com/Pylons/pyramid/issues/709>
- In uncommon cases, the `pyramid_excview_tween_factory` might have inadvertently raised a `KeyError` looking for `request_iface` as an attribute of the request. It no longer fails in this case. See <https://github.com/Pylons/pyramid/issues/700>
- Be more tolerant of potential error conditions in `match_param` and `physical_path` predicate implementations; instead of raising an exception, return `False`.
- `pyramid.view.render_view` was not functioning properly under Python 3.x due to a byte/unicode discrepancy. See <https://github.com/Pylons/pyramid/issues/721>

Deprecations

- `pyramid.authentication.AuthTktAuthenticationPolicy` will emit a warning if an application is using the policy without explicitly passing a `hashalg` argument. This is because the default is “md5” which is considered theoretically subject to collision attacks. If you really want “md5” then you must specify it explicitly to get rid of the warning.

Documentation

- All of the tutorials that use `pyramid.authentication.AuthTktAuthenticationPolicy` now explicitly pass `sha512` as a `hashalg` argument.

Internals

- Move `TopologicalSorter` from `pyramid.config.util` to `pyramid.util`, move `CyclicDependencyError` from `pyramid.config.util` to `pyramid.exceptions`, rename `Singleton` to `Sentinel` and move from `pyramid.config.util` to `pyramid.util`; this is in an effort to move that stuff that may be an API one day out of `pyramid.config.util`, because that package should never be imported from non-Pyramid code. `TopologicalSorter` is still not an API, but may become one.
- Get rid of shady monkeypatching of `pyramid.request.Request` and `pyramid.response.Response` done within the `__init__.py` of `Pyramid`. `WebOb` no longer relies on this being done. Instead, the `ResponseClass` attribute of the `Pyramid Request` class is assigned to the `Pyramid response` class; that’s enough to satisfy `WebOb` and behave as it did before with the monkeypatching.

1.4a3 (2012-10-26)

Bug Fixes

- The `match_param` predicate's text method was fixed to sort its values. Part of <https://github.com/Pylons/pyramid/pull/705>
- 1.4a `pyramid.scripting.prepare` behaved differently than 1.3 series function of same name. In particular, if passed a request, it would not set the `registry` attribute of the request like 1.3 did. A symptom would be that passing a request to `pyramid.paster.bootstrap` (which uses the function) that did not have a `registry` attribute could assume that the registry would be attached to the request by Pyramid. This assumption could be made in 1.3, but not in 1.4. The assumption can now be made in 1.4 too (a registry is attached to a request passed to `bootstrap` or `prepare`).
- When registering a view configuration that named a Chameleon ZPT renderer with a macro name in it (e.g. `renderer='some/template#somemacro.pt'`) as well as a view configuration without a macro name in it that pointed to the same template (e.g. `renderer='some/template.pt'`), internal caching could confuse the two, and your code might have rendered one instead of the other.

Features

- Allow multiple values to be specified to the `request_param` view/route predicate as a sequence. Previously only a single string value was allowed. See <https://github.com/Pylons/pyramid/pull/705>
- Comments with references to documentation sections placed in scaffold `.ini` files.
- Added an HTTP Basic authentication policy at `pyramid.authentication.BasicAuthAuthenticationPolicy`.
- The Configurator `testing_securitypolicy` method now returns the policy object it creates.
- The Configurator `testing_securitypolicy` method accepts two new arguments: `remember_result` and `forget_result`. If supplied, these values influence the result of the policy's `remember` and `forget` methods, respectively.
- The `DummySecurityPolicy` created by `testing_securitypolicy` now sets a `forgotten` value on the policy (the value `True`) when its `forget` method is called.

- The `DummySecurityPolicy` created by `testing_securitypolicy` now sets a `remembered` value on the policy, which is the value of the `principal` argument it's called with when its `remember` method is called.
- New `physical_path` view predicate. If specified, this value should be a string or a tuple representing the physical traversal path of the context found via traversal for this predicate to match as true. For example: `physical_path= '/'` or `physical_path= '/a/b/c'` or `physical_path= ('', 'a', 'b', 'c')`. This is not a path prefix match or a regex, it's a whole-path match. It's useful when you want to always potentially show a view when some object is traversed to, but you can't be sure about what kind of object it will be, so you can't use the `context` predicate. The individual path elements inbetween slash characters or in tuple elements should be the Unicode representation of the name of the resource and should not be encoded in any way.

1.4a2 (2012-09-27)

Bug Fixes

- When trying to determine Mako defnames and Chameleon macro names in asset specifications, take into account that the filename may have a hyphen in it. See <https://github.com/Pylons/pyramid/pull/692>

Features

- A new `pyramid.session.check_csrf_token` convenience function was added.
- A `check_csrf` view predicate was added. For example, you can now do `config.add_view(someview, check_csrf=True)`. When the predicate is checked, if the `csrf_token` value in `request.params` matches the CSRF token in the request's session, the view will be permitted to execute. Otherwise, it will not be permitted to execute.
- Add `Base.metadata.bind = engine` to alchemy template, so that tables defined imperatively will work.

Documentation

- update wiki2 SQLA tutorial with the changes required after inserting `Base.metadata.bind = engine` into the alchemy scaffold.

1.4a1 (2012-09-16)

Bug Fixes

- Forward port from 1.3 branch: When no authentication policy was configured, a call to `pyramid.security.effective_principals` would unconditionally return the empty list. This was incorrect, it should have unconditionally returned `[Everyone]`, and now does.
- Explicit url dispatch regexes can now contain colons. <https://github.com/Pylons/pyramid/issues/629>
- On at least one 64-bit Ubuntu system under Python 3.2, using the `view_config` decorator caused a `RuntimeError: dictionary changed size during iteration` exception. It no longer does. See <https://github.com/Pylons/pyramid/issues/635> for more information.
- In Mako Templates lookup, check if the uri is already adjusted and bring it back to an asset spec. Normally occurs with inherited templates or included components. <https://github.com/Pylons/pyramid/issues/606> <https://github.com/Pylons/pyramid/issues/607>
- In Mako Templates lookup, check for absolute uri (using mako directories) when mixing up inheritance with asset specs. <https://github.com/Pylons/pyramid/issues/662>
- HTTP Accept headers were not being normalized causing potentially conflicting view registrations to go unnoticed. Two views that only differ in the case ('text/html' vs. 'text/HTML') will now raise an error. <https://github.com/Pylons/pyramid/pull/620>
- Forward-port from 1.3 branch: when registering multiple views with an `accept` predicate in a Pyramid application running under Python 3, you might have received a `TypeError: unorderable types: function() < function()` exception.

Features

- Python 3.3 compatibility.
- `Configurator.add_directive` now accepts arbitrary callables like partials or objects implementing `__call__` which don't have `__name__` and `__doc__` attributes. See <https://github.com/Pylons/pyramid/issues/621> and <https://github.com/Pylons/pyramid/pull/647>.
- Third-party custom view, route, and subscriber predicates can now be added for use by view authors via `pyramid.config.Configurator.add_view_predicate`, `pyramid.config.Configurator.add_route_predicate` and `pyramid.config.Configurator.add_subscriber_predicate`. So, for example, doing this:

```
config.add_view_predicate('abc', my.package.ABCPredicate)
```

Might allow a view author to do this in an application that configured that predicate:

```
@view_config(abc=1)
```

Similar features exist for `add_route`, and `add_subscriber`. See “Adding A Third Party View, Route, or Subscriber Predicate” in the Hooks chapter for more information.

Note that changes made to support the above feature now means that only actions registered using the same “order” can conflict with one another. It used to be the case that actions registered at different orders could potentially conflict, but to my knowledge nothing ever depended on this behavior (it was a bit silly).

- Custom objects can be made easily JSON-serializable in Pyramid by defining a `__json__` method on the object’s class. This method should return values natively serializable by `json.dumps` (such as ints, lists, dictionaries, strings, and so forth).
- The JSON renderer now allows for the definition of custom type adapters to convert unknown objects to JSON serializations.
- As of this release, the `request_method` predicate, when used, will also imply that HEAD is implied when you use GET. For example, using `@view_config(request_method='GET')` is equivalent to using `@view_config(request_method=('GET', 'HEAD'))`. Using `@view_config(request_method=('GET', 'POST'))` is equivalent to using `@view_config(request_method=('GET', 'HEAD', 'POST'))`. This is because HEAD is a variant of GET that omits the body, and WebOb has special support to return an empty body when a HEAD is used.
- `config.add_request_method` has been introduced to support extending request objects with arbitrary callables. This method expands on the previous `config.set_request_property` by supporting methods as well as properties. This method now causes less code to be executed at request construction time than `config.set_request_property` in version 1.3.
- Don’t add a `?` to URLs generated by `request.resource_url` if the `query` argument is provided but empty.
- Don’t add a `?` to URLs generated by `request.route_url` if the `_query` argument is provided but empty.

- The static view machinery now raises (rather than returns) `HTTPNotFound` and `HTTPMovedPermanently` exceptions, so these can be caught by the Not Found View (and other exception views).
- The Mako renderer now supports a def name in an asset spec. When the def name is present in the asset spec, the system will render the template def within the template and will return the result. An example asset spec is `package:path/to/template#defname.mako`. This will render the def named `defname` inside the `template.mako` template instead of rendering the entire template. The old way of returning a tuple in the form `('defname', {})` from the view is supported for backward compatibility,
- The Chameleon ZPT renderer now accepts a macro name in an asset spec. When the macro name is present in the asset spec, the system will render the macro listed as a `define-macro` and return the result instead of rendering the entire template. An example asset spec: `package:path/to/template#macroname.pt`. This will render the macro defined as `macroname` within the `template.pt` template instead of the entire template.
- When there is a predicate mismatch exception (seen when no view matches for a given request due to predicates not working), the exception now contains a textual description of the predicate which didn't match.
- An `add_permission` directive method was added to the Configurator. This directive registers a free-standing permission introspectable into the Pyramid introspection system. Frameworks built atop Pyramid can thus use the `permissions` introspectable category data to build a comprehensive list of permissions supported by a running system. Before this method was added, permissions were already registered in this introspectable category as a side effect of naming them in an `add_view` call, this method just makes it possible to arrange for a permission to be put into the `permissions` introspectable category without naming it along with an associated view. Here's an example of usage of `add_permission`:

```
config = Configurator()
config.add_permission('view')
```

- The `UnencryptedCookieSessionFactoryConfig` now accepts `signed_serialize` and `signed_deserialize` hooks which may be used to influence how the sessions are marshalled (by default this is done with `HMAC+pickle`).
- `pyramid.testing.DummyRequest` now supports methods supplied by the `pyramid.util.InstancePropertyMixin` class such as `set_property`.
- Request properties and methods added via `config.set_request_property` or `config.add_request_method` are now available to tweens.

- Request properties and methods added via `config.set_request_property` or `config.add_request_method` are now available in the request object returned from `pyramid.paster.bootstrap`.
- `request.context` of environment request during bootstrap is now the root object if a context isn't already set on a provided request.
- The `pyramid.decorator.reify` function is now an API, and was added to the API documentation.
- Added the `pyramid.testing.testConfig` context manager, which can be used to generate a configurator in a test, e.g. with `testing.testConfig(...)` :.
- Users can now invoke a subrequest from within view code using a new `request.invoke_subrequest` API.

Deprecations

- The `pyramid.config.Configurator.set_request_property` has been documentation-deprecated. The method remains usable but the more featureful `pyramid.config.Configurator.add_request_method` should be used in its place (it has all of the same capabilities but can also extend the request object with methods).

Backwards Incompatibilities

- The Pyramid router no longer adds the values `bfg.routes.route` or `bfg.routes.matchdict` to the request's WSGI environment dictionary. These values were docs-deprecated in `repoze.bfg` 1.0 (effectively seven minor releases ago). If your code depended on these values, use `request.matched_route` and `request.matchdict` instead.
- It is no longer possible to pass an environ dictionary directly to `pyramid.traversal.ResourceTreeTraverser.__call__` (aka `ModelGraphTraverser.__call__`). Instead, you must pass a request object. Passing an environment instead of a request has generated a deprecation warning since Pyramid 1.1.
- Pyramid will no longer work properly if you use the `webob.request.LegacyRequest` as a request factory. Instances of the `LegacyRequest` class have a `request.path_info` which return a string. This Pyramid release assumes that `request.path_info` will unconditionally be Unicode.

- The functions from `pyramid.chameleon_zpt` and `pyramid.chameleon_text` named `get_renderer`, `get_template`, `render_template`, and `render_template_to_response` have been removed. These have issued a deprecation warning upon import since Pyramid 1.0. Use `pyramid.renderers.get_renderer()`, `pyramid.renderers.get_renderer().implementation()`, `pyramid.renderers.render()` or `pyramid.renderers.render_to_response` respectively instead of these functions.
- The `pyramid.configuration` module was removed. It had been deprecated since Pyramid 1.0 and printed a deprecation warning upon its use. Use `pyramid.config` instead.
- The `pyramid.paster.PyramidTemplate` API was removed. It had been deprecated since Pyramid 1.1 and issued a warning on import. If your code depended on this, adjust your code to import `pyramid.scaffolds.PyramidTemplate` instead.
- The `pyramid.settings.get_settings()` API was removed. It had been printing a deprecation warning since Pyramid 1.0. If your code depended on this API, use `pyramid.threadlocal.get_current_registry().settings` instead or use the `settings` attribute of the registry available from the request (`request.registry.settings`).
- These APIs from the `pyramid.testing` module were removed. They have been printing deprecation warnings since Pyramid 1.0:
 - `registerDummySecurityPolicy`, use `pyramid.config.Configurator.testing_securitypolicy` instead.
 - `registerResources` (aka `registerModels`), use `pyramid.config.Configurator.testing_resources` instead.
 - `registerEventListener`, use `pyramid.config.Configurator.testing_add_subscriber` instead.
 - `registerTemplateRenderer` (aka `registerDummyRenderer`), use `pyramid.config.Configurator.testing_add_template` instead.
 - `registerView`, use `pyramid.config.Configurator.add_view` instead.
 - `registerUtility`, use `pyramid.config.Configurator.registry.registerUtility` instead.
 - `registerAdapter`, use `pyramid.config.Configurator.registry.registerAdapter` instead.

- `registerSubscriber`, `add_subscriber` instead. use `pyramid.config.Configurator`.
- `registerRoute`, use `pyramid.config.Configurator.add_route` instead.
- `registerSettings`, use `pyramid.config.Configurator.add_settings` instead.
- In Pyramid 1.3 and previous, the `__call__` method of a `Response` object was invoked before any finished callbacks were executed. As of this release, the `__call__` method of a `Response` object is invoked *after* finished callbacks are executed. This is in support of the `request.invoke_subrequest` feature.
- The 200-series exception responses named `HTTPCreated`, `HTTPAccepted`, `HTTPNonAuthoritativeInformation`, `HTTPNoContent`, `HTTPResetContent`, and `HTTPPartialContent` in `pyramid.httpexceptions` no longer inherit from `HTTPOk`. Instead they inherit from a new base class named `HTTPSuccessful`. This will have no effect on you unless you've registered an exception view for `HTTPOk` and expect that exception view to catch all the aforementioned exceptions.

Documentation

- Added an “Upgrading Pyramid” chapter to the narrative documentation. It describes how to cope with deprecations and removals of Pyramid APIs and how to show Pyramid-generated deprecation warnings while running tests and while running a server.
- Added a “Invoking a Subrequest” chapter to the documentation. It describes how to use the new `request.invoke_subrequest` API.

Dependencies

- Pyramid now requires WebOb 1.2b3+ (the prior Pyramid release only relied on 1.2dev+). This is to ensure that we obtain a version of WebOb that returns `request.path_info` as text.

1.3 (2012-03-21)

Bug Fixes

- When `pyramid.wsgi.wsgiapp2` calls the downstream WSGI app, the app's `environ` will no longer have (deprecated and potentially misleading) `bfg.routes.matchdict` or `bfg.routes.route` keys in it. A symptom of this bug would be a `wsgiapp2`-wrapped Pyramid app finding the wrong view because it mistakenly detects that a route was matched when, in fact, it was not.
- The fix for issue <https://github.com/Pylons/pyramid/issues/461> (which made it possible for instance methods to be used as view callables) introduced a backwards incompatibility when methods that declared only a request argument were used. See <https://github.com/Pylons/pyramid/issues/503>

1.3b3 (2012-03-17)

Bug Fixes

- `config.add_view(<aninstancemethod>)` raised `AttributeError` involving `__text__`. See <https://github.com/Pylons/pyramid/issues/461>
- Remove references to do-nothing `pyramid.debug_templates` setting in all Pyramid-provided `.ini` files. This setting previously told Chameleon to render better exceptions; now Chameleon always renders nice exceptions regardless of the value of this setting.

Scaffolds

- The `alchemy` scaffold now shows an informative error message in the browser if the person creating the project forgets to run the initialization script.
- The `alchemy` scaffold initialization script is now called `initialize_<projectname>_db` instead of `populate_<projectname>`.

Documentation

- Wiki tutorials improved due to collaboration at PyCon US 2012 sprints.

1.3b2 (2012-03-02)

Bug Fixes

- The method `pyramid.request.Request.partial_application_url` is no longer in the API docs. It was meant to be a private method; its publication in the documentation as an API method was a mistake, and it has been renamed to something private.
- When a static view was registered using an absolute filesystem path on Windows, the `request.static_url` function did not work to generate URLs to its resources. Symptom: “No static URL definition matching `c:\foo\bar\baz`”.
- Make all tests pass on Windows XP.
- Bug in ACL authentication checking on Python 3: the `permits` and `principals_allowed_by_permission` method of `pyramid.authorization.ACLAuthenticationPolicy` could return an inappropriate `True` value when a permission on an ACL was a string rather than a sequence, and then only if the ACL permission string was a substring of the permission value passed to the function.

This bug effects no Pyramid deployment under Python 2; it is a bug that exists only in deployments running on Python 3. It has existed since Pyramid 1.3a1.

This bug was due to the presence of an `__iter__` attribute on strings under Python 3 which is not present under strings in Python 2.

1.3b1 (2012-02-26)

Bug Fixes

- `pyramid.config.Configurator.with_package` didn't work if the `Configurator` was an old-style `pyramid.configuration.Configurator` instance.
- Pyramid authorization policies did not show up in the introspector.

Deprecations

- All references to the `tmpl_context` request variable were removed from the docs. Its existence in Pyramid is confusing for people who were never Pylons users. It was added as a porting convenience for Pylons users in Pyramid 1.0, but it never caught on because the Pyramid rendering system is a lot different than Pylons' was, and alternate ways exist to do what it was designed to offer in Pylons. It will continue to exist "forever" but it will not be recommended or mentioned in the docs.

1.3a9 (2012-02-22)

Features

- Add an `introspection` boolean to the `Configurator` constructor. If this is `True`, actions registered using the `Configurator` will be registered with the introspector. If it is `False`, they won't. The default is `True`. Setting it to `False` during action processing will prevent introspection for any following registration statements, and setting it to `True` will start them up again. This addition is to service a requirement that the debug toolbar's own views and methods not show up in the introspector.
- New API: `pyramid.config.Configurator.add_notfound_view`. This is a wrapper for `pyramid.Config.configurator.add_view` which provides easy `append_slash` support and does the right thing about permissions. It should be preferred over calling `add_view` directly with `context=HTTPNotFound` as was previously recommended.
- New API: `pyramid.view.notfound_view_config`. This is a decorator constructor like `pyramid.view.view_config` that calls `pyramid.config.Configurator.add_notfound_view` when scanned. It should be preferred over using `pyramid.view.view_config` with `context=HTTPNotFound` as was previously recommended.
- New API: `pyramid.config.Configurator.add_forbidden_view`. This is a wrapper for `pyramid.Config.configurator.add_view` which does the right thing about permissions. It should be preferred over calling `add_view` directly with `context=HTTPForbidden` as was previously recommended.
- New API: `pyramid.view.forbidden_view_config`. This is a decorator constructor like `pyramid.view.view_config` that calls `pyramid.config.Configurator.add_forbidden_view` when scanned. It should be preferred over using `pyramid.view.view_config` with `context=HTTPForbidden` as was previously recommended.
- New APIs: `pyramid.response.FileResponse` and `pyramid.response.FileIter`, for usage in views that must serve files "manually".

Backwards Incompatibilities

- Remove `pyramid.config.Configurator.with_context` class method. It was never an API, it is only used by `pyramid_zcml` and its functionality has been moved to that package's latest release. This means that you'll need to use the 0.9.2 or later release of `pyramid_zcml` with this release of Pyramid.
- The `introspector` argument to the `pyramid.config.Configurator` constructor API has been removed. It has been replaced by the boolean `introspection` flag.
- The `pyramid.registry.noop_introspector` API object has been removed.
- The older deprecated `set_notfound_view` `Configurator` method is now an alias for the new `add_notfound_view` `Configurator` method. Likewise, the older deprecated `set_forbidden_view` is now an alias for the new `add_forbidden_view`. This has the following impact: the context sent to views with a `(context, request)` call signature registered via the `set_notfound_view` or `set_forbidden_view` will now be an exception object instead of the actual resource context found. Use `request.context` to get the actual resource context. It's also recommended to disuse `set_notfound_view` in favor of `add_notfound_view`, and disuse `set_forbidden_view` in favor of `add_forbidden_view` despite the aliasing.

Deprecations

- The API documentation for `pyramid.view.append_slash_notfound_view` and `pyramid.view.AppendSlashNotFoundViewFactory` was removed. These names still exist and are still importable, but they are no longer APIs. Use `pyramid.config.Configurator.add_notfound_view(append_slash=True)` or `pyramid.view.notfound_view_config(append_slash=True)` to get the same behavior.
- The `set_forbidden_view` and `set_notfound_view` methods of the `Configurator` were removed from the documentation. They have been deprecated since Pyramid 1.1.

Bug Fixes

- The static file response object used by `config.add_static_view` opened the static file twice, when it only needed to open it once.
- The `AppendSlashNotFoundViewFactory` used `request.path` to match routes. This was wrong because `request.path` contains the script name, and this would cause it to fail in circumstances where the script name was not empty. It should have used `request.path_info`, and now does.

Documentation

- Updated the “Creating a Not Found View” section of the “Hooks” chapter, replacing explanations of registering a view using `add_view` or `view_config` with ones using `add_notfound_view` or `notfound_view_config`.
- Updated the “Creating a Not Forbidden View” section of the “Hooks” chapter, replacing explanations of registering a view using `add_view` or `view_config` with ones using `add_forbidden_view` or `forbidden_view_config`.
- Updated the “Redirecting to Slash-Appended Routes” section of the “URL Dispatch” chapter, replacing explanations of registering a view using `add_view` or `view_config` with ones using `add_notfound_view` or `notfound_view_config`.
- Updated all tutorials to use `pyramid.view.forbidden_view_config` rather than `pyramid.view.view_config` with an `HTTPForbidden` context.

1.3a8 (2012-02-19)

Features

- The `scan` method of a `Configurator` can be passed an `ignore` argument, which can be a string, a callable, or a list consisting of strings and/or callables. This feature allows submodules, subpackages, and global objects from being scanned. See <http://readthedocs.org/docs/venusian/en/latest/#ignore-scan-argument> for more information about how to use the `ignore` argument to `scan`.
- Better error messages when a view callable returns a value that cannot be converted to a response (for example, when a view callable returns a dictionary without a `renderer` defined, or doesn’t return any value at all). The error message now contains information about the view callable itself as well as the result of calling it.
- Better error message when a `.pyc`-only module is `config.include`-ed. This is not permitted due to error reporting requirements, and a better error message is shown when it is attempted. Previously it would fail with something like “`AttributeError: ‘NoneType’ object has no attribute ‘rfind’`”.
- Add `pyramid.config.Configurator.add_traverser` API method. See the Hooks narrative documentation section entitled “Changing the Traverser” for more information. This is not a new feature, it just provides an API for adding a traverser without needing to use the ZCA API.

- Add `pyramid.config.Configurator.add_resource_url_adapter` API method. See the Hooks narrative documentation section entitled “Changing How `pyramid.request.Request.resource_url` Generates a URL” for more information. This is not a new feature, it just provides an API for adding a resource url adapter without needing to use the ZCA API.
- The system value `req` is now supplied to renderers as an alias for `request`. This means that you can now, for example, in a template, do `req.route_url(...)` instead of `request.route_url(...)`. This is purely a change to reduce the amount of typing required to use `request` methods and attributes from within templates. The value `request` is still available too, this is just an alternative.
- A new interface was added: `pyramid.interfaces.IResourceURL`. An adapter implementing its interface can be used to override resource URL generation when `request.resource_url` is called. This interface replaces the now-deprecated `pyramid.interfaces.IContextURL` interface.
- The dictionary passed to a resource’s `__resource_url__` method (see “Overriding Resource URL Generation” in the “Resources” chapter) now contains an `app_url` key, representing the application URL generated during `request.resource_url`. It represents a potentially customized URL prefix, containing potentially custom scheme, host and port information passed by the user to `request.resource_url`. It should be used instead of `request.application_url` where necessary.
- The `request.resource_url` API now accepts these arguments: `app_url`, `scheme`, `host`, and `port`. The `app_url` argument can be used to replace the URL prefix wholesale during url generation. The `scheme`, `host`, and `port` arguments can be used to replace the respective default values of `request.application_url` partially.
- A new API named `request.resource_path` now exists. It works like `request.resource_url` but produces a relative URL rather than an absolute one.
- The `request.route_url` API now accepts these arguments: `_app_url`, `_scheme`, `_host`, and `_port`. The `_app_url` argument can be used to replace the URL prefix wholesale during url generation. The `_scheme`, `_host`, and `_port` arguments can be used to replace the respective default values of `request.application_url` partially.

Backwards Incompatibilities

- The `pyramid.interfaces.IContextURL` interface has been deprecated. People have been instructed to use this to register a resource url adapter in the “Hooks” chapter to use to influence `request.resource_url` URL generation for resources found via custom traversers since Pyramid 1.0.

The interface still exists and registering such an adapter still works, but this interface will be removed from the software after a few major Pyramid releases. You should replace it with an equivalent `pyramid.interfaces.IResourceURL` adapter, registered using the new `pyramid.config.Configurator.add_resource_url_adapter` API. A deprecation warning is now emitted when a `pyramid.interfaces.IContextURL` adapter is found when `request.resource_url` is called.

Documentation

- Don't create a `session` instance in SQLA Wiki tutorial, use raw `DBSession` instead (this is more common in real SQLA apps).

Scaffolding

- Put `pyramid.includes` targets within ini files in scaffolds on separate lines in order to be able to tell people to comment out only the `pyramid_debugtoolbar` line when they want to disable the toolbar.

Dependencies

- Depend on `venusian >= 1.0a3` to provide scan ignore support.

Internal

- Create a “MakoRendererFactoryHelper” that provides customizable settings key prefixes. Allows settings prefixes other than “mako.” to be used to create different factories that don't use the global mako settings. This will be useful for the debug toolbar, which can currently be sabotaged by someone using custom mako configuration settings.

1.3a7 (2012-02-07)

Features

- More informative error message when a `config.include` cannot find an `includeme`. See <https://github.com/Pylons/pyramid/pull/392>.
- Internal: catch unhashable discriminators early (raise an error instead of allowing them to find their way into `resolveConflicts`).
- The `match_param` view predicate now accepts a string or a tuple. This replaces the broken behavior of accepting a dict. See <https://github.com/Pylons/pyramid/issues/425> for more information.

Bug Fixes

- The process will now restart when `pserve` is used with the `--reload` flag when the `development.ini` file (or any other `.ini` file in use) is changed. See <https://github.com/Pylons/pyramid/issues/377> and <https://github.com/Pylons/pyramid/pull/411>
- The `prequest` script would fail when used against URLs which did not return HTML or text. See <https://github.com/Pylons/pyramid/issues/381>

Backwards Incompatibilities

- The `match_param` view predicate no longer accepts a dict. This will have no negative affect because the implementation was broken for dict-based arguments.

Documentation

- Add a traversal hello world example to the narrative docs.

1.3a6 (2012-01-20)

Features

- New API: `pyramid.config.Configurator.set_request_property`. Add lazy property descriptors to a request without changing the request factory. This method provides conflict detection and is the suggested way to add properties to a request.
- Responses generated by Pyramid's `static_view` now use a `wsgi.file_wrapper` (see <http://www.python.org/dev/peps/pep-0333/#optional-platform-specific-file-handling>) when one is provided by the web server.

Bug Fixes

- Views registered with an `accept` could not be overridden correctly with a different view that had the same predicate arguments. See <https://github.com/Pylons/pyramid/pull/404> for more information.
- When using a dotted name for a `view` argument to `Configurator.add_view` that pointed to a class with a `view_defaults` decorator, the view defaults would not be applied. See <https://github.com/Pylons/pyramid/issues/396>.
- Static URL paths were URL-quoted twice. See <https://github.com/Pylons/pyramid/issues/407>.

1.3a5 (2012-01-09)

Bug Fixes

- The `pyramid.view.view_defaults` decorator did not work properly when more than one view relied on the defaults being different for configuration conflict resolution. See <https://github.com/Pylons/pyramid/issues/394>.

Backwards Incompatibilities

- The `path_info` route and view predicates now match against `request.upath_info` (Unicode) rather than `request.path_info` (indeterminate value based on Python 3 vs. Python 2). This has to be done to normalize matching on Python 2 and Python 3.

1.3a4 (2012-01-05)

Features

- New API: `pyramid.request.Request.set_property`. Add lazy property descriptors to a request without changing the request factory. New properties may be reified, effectively caching the value for the lifetime of the instance. Common use-cases for this would be to get a database connection for the request or identify the current user.
- Use the `waitress` WSGI server instead of `wsgiref` in scaffolding.

Bug Fixes

- The documentation of `pyramid.events.subscriber` indicated that using it as a decorator with no arguments like this:

```
@subscriber()
def somefunc(event):
    pass
```

Would register `somefunc` to receive all events sent via the registry, but this was untrue. Instead, it would receive no events at all. This has now been fixed and the code matches the documentation. See also <https://github.com/Pylons/pyramid/issues/386>

- Literal portions of route patterns were not URL-quoted when `route_url` or `route_path` was used to generate a URL or path.
- The result of `route_path` or `route_url` might have been `unicode` or `str` depending on the input. It is now guaranteed to always be `str`.
- URL matching when the pattern contained non-ASCII characters in literal parts was indeterminate. Now the pattern supplied to `add_route` is assumed to be either: a `unicode` value, or a `str` value that contains only ASCII characters. If you now want to match the path info from a URL that contains high order characters, you can pass the Unicode representation of the decoded path portion in the pattern.
- When using a `traverse=` route predicate, traversal would fail with a `URLDecodeError` if there were any high-order characters in the traversal pattern or in the matched dynamic segments.
- Using a dynamic segment named `traverse` in a route pattern like this:

```
config.add_route('trav_route', 'traversal/{traverse:.*}')
```

Would cause a `UnicodeDecodeError` when the route was matched and the matched portion of the URL contained any high-order characters. See <https://github.com/Pylons/pyramid/issues/385> .

- When using a `*traverse` stararg in a route pattern, a URL that matched that possessed a `@@` in its name (signifying a view name) would be inappropriately quoted by the traversal machinery during traversal, resulting in the view not being found properly. See <https://github.com/Pylons/pyramid/issues/382> and <https://github.com/Pylons/pyramid/issues/375> .

Backwards Incompatibilities

- String values passed to `route_url` or `route_path` that are meant to replace “remainder” matches will now be URL-quoted except for embedded slashes. For example:

```
config.add_route('remain', '/foo*remainder')
request.route_path('remain', remainder='abc / def')
# -> '/foo/abc%20/%20def'
```

Previously string values passed as remainder replacements were tacked on untouched, without any URL-quoting. But this doesn't really work logically if the value passed is Unicode (raw unicode cannot be placed in a URL or in a path) and it is inconsistent with the rest of the URL generation machinery if the value is a string (it won't be quoted unless by the caller).

Some folks will have been relying on the older behavior to tack on query string elements and anchor portions of the URL; sorry, you'll need to change your code to use the `_query` and/or `_anchor` arguments to `route_path` or `route_url` to do this now.

- If you pass a bytestring that contains non-ASCII characters to `add_route` as a pattern, it will now fail at startup time. Use Unicode instead.

1.3a3 (2011-12-21)

Features

- Added a `prequest` script (along the lines of `paster request`). It is documented in the “Command-Line Pyramid” chapter in the section entitled “Invoking a Request”.
- Add undocumented `__discriminator__` API to derived view callables. e.g. `adapters.lookup(...).__discriminator__(context, request)`. It will be used by superdynamic systems that require the discriminator to be used for introspection after manual view lookup.

Bug Fixes

- Normalized exit values and `-h` output for all `p*` scripts (`pviews`, `proutes`, etc).

Documentation

- Added a section named “Making Your Script into a Console Script” in the “Command-Line Pyramid” chapter.
- Removed the “Running Pyramid on Google App Engine” tutorial from the main docs. It survives on in the Cookbook (http://docs.pylonsproject.org/projects/pyramid_cookbook/en/latest/deployment/gae.html). Rationale: it provides the correct info for the Python 2.5 version of GAE only, and this version of Pyramid does not support Python 2.5.

1.3a2 (2011-12-14)

Features

- New API: `pyramid.view.view_defaults`. If you use a class as a view, you can use the new `view_defaults` class decorator on the class to provide defaults to the view configuration information used by every `@view_config` decorator that decorates a method of that class. It also works against view configurations involving a class made imperatively.
- Added a backwards compatibility knob to `pcreate` to emulate `paster create` handling for the `--list-templates` option.
- Changed scaffolding machinery around a bit to make it easier for people who want to have extension scaffolds that can work across Pyramid 1.0.X, 1.1.X, 1.2.X and 1.3.X. See the new “Creating Pyramid Scaffolds” chapter in the narrative documentation for more info.

Documentation

- Added documentation to “View Configuration” narrative documentation chapter about `view_defaults` class decorator.
- Added API docs for `view_defaults` class decorator.
- Added an API docs chapter for `pyramid.scaffolds`.
- Added a narrative docs chapter named “Creating Pyramid Scaffolds”.

Backwards Incompatibilities

- The `template_renderer` method of `pyramid.scaffolds.PyramidScaffold` was renamed to `render_template`. If you were overriding it, you’re a bad person, because it wasn’t an API before now. But we’re nice so we’re letting you know.

1.3a1 (2011-12-09)

Features

- Python 3.2 compatibility.
- New `pyramid.compat` module and API documentation which provides Python 2/3 straddling support for Pyramid add-ons and development environments.
- A `mako.directories` setting is no longer required to use Mako templates Rationale: Mako template renderers can be specified using an absolute asset spec. An entire application can be written with such asset specs, requiring no ordered lookup path.
- `bpython` interpreter compatibility in `pshell`. See the “Command-Line Pyramid” narrative docs chapter for more information.
- Added `get_appsettings` API function to the `pyramid.paster` module. This function returns the settings defined within an `[app: ...]` section in a PasteDeploy ini file.
- Added `setup_logging` API function to the `pyramid.paster` module. This function sets up Python logging according to the logging configuration in a PasteDeploy ini file.
- Configuration conflict reporting is reported in a more understandable way (“Line 11 in file...” vs. a repr of a tuple of similar info).
- A configuration introspection system was added; see the narrative documentation chapter entitled “Pyramid Configuration Introspection” for more information. New APIs: `pyramid.registry.Introspectable`, `pyramid.config.Configurator.introspector`, `pyramid.config.Configurator.introspectable`, `pyramid.registry.Registry.introspector`.
- Allow extra keyword arguments to be passed to the `pyramid.config.Configurator.action` method.
- New APIs: `pyramid.path.AssetResolver` and `pyramid.path.DottedNameResolver`. The former can be used to resolve asset specifications, the latter can be used to resolve dotted names to modules or packages.

Bug Fixes

- Make test suite pass on 32-bit systems; closes #286. closes #306. See also <https://github.com/Pylons/pyramid/issues/286>
- The `pyramid.view.view_config` decorator did not accept a `match_params` predicate argument. See <https://github.com/Pylons/pyramid/pull/308>
- The `AuthTktCookieHelper` could potentially generate Unicode headers inappropriately when the `tokens` argument to `remember` was used. See <https://github.com/Pylons/pyramid/pull/314>.
- The `AuthTktAuthenticationPolicy` did not use a timing-attack-aware string comparator. See <https://github.com/Pylons/pyramid/pull/320> for more info.
- The `DummySession` in `pyramid.testing` now generates a new CSRF token if one doesn't yet exist.
- `request.static_url` now generates URL-quoted URLs when fed a `path` argument which contains characters that are unsuitable for URLs. See <https://github.com/Pylons/pyramid/issues/349> for more info.
- Prevent a scaffold rendering from being named `site` (conflicts with Python internal `site.py`).
- Support for using instances as targets of the `pyramid.wsgi.wsgiapp` and `pyramid.wsgi.wsgiapp2` functions. See <https://github.com/Pylons/pyramid/pull/370> for more info.

Backwards Incompatibilities

- Pyramid no longer runs on Python 2.5 (which includes the most recent release of Jython and the Python 2.5 version of GAE as of this writing).
- The `paster` command is no longer the documented way to create projects, start the server, or run debugging commands. To create projects from scaffolds, `paster create` is replaced by the `pcreate` console script. To serve up a project, `paster serve` is replaced by the `pserve` console script. New console scripts named `pshell`, `pviews`, `proutes`, and `ptweens` do what their `paster <commandname>` equivalents used to do. Rationale: the Paste and PasteScript packages do not run under Python 3.
- The default WSGI server run as the result of `pserve` from newly rendered scaffolding is now the `wsgiref` WSGI server instead of the `paste.httpserver` server. Rationale: the Paste and PasteScript packages do not run under Python 3.

- The `pshell` command (see “`paster pshell`”) no longer accepts a `--disable-ipython` command-line argument. Instead, it accepts a `-p` or `--python-shell` argument, which can be any of the values `python`, `ipython` or `bpython`.
- Removed the `pyramid.renderers.renderer_from_name` function. It has been deprecated since Pyramid 1.0, and was never an API.
- To use ZCML with versions of Pyramid ≥ 1.3 , you will need `pyramid_zcml` version ≥ 0.8 and `zope.configuration` version $\geq 3.8.0$. The `pyramid_zcml` package version 0.8 is backwards compatible all the way to Pyramid 1.0, so you won’t be warned if you have older versions installed and upgrade Pyramid “in-place”; it may simply break instead.

Dependencies

- Pyramid no longer depends on the `zope.component` package, except as a testing dependency.
- Pyramid now depends on a `zope.interface` $\geq 3.8.0$, `WebOb` $\geq 1.2dev$, `repoze.lru` ≥ 0.4 , `zope.deprecation` $\geq 3.5.0$, `translationstring` ≥ 0.4 (for Python 3 compatibility purposes). It also, as a testing dependency, depends on `WebTest` $\geq 1.3.1$ for the same reason.
- Pyramid no longer depends on the Paste or PasteScript packages.

Documentation

- The SQLAlchemy Wiki tutorial has been updated. It now uses `@view_config` decorators and an explicit database population script.
- Minor updates to the ZODB Wiki tutorial.
- A narrative documentation chapter named “Extending Pyramid Configuration” was added; it describes how to add a new directive, and how use the `pyramid.config.Configurator.action` method within custom directives. It also describes how to add introspectable objects.
- A narrative documentation chapter named “Pyramid Configuration Introspection” was added. It describes how to query the introspection system.

Scaffolds

- Rendered scaffolds have now been changed to be more relocatable (fewer mentions of the package name within files in the package).
- The `routesalchemy` scaffold has been renamed `alchemy`, replacing the older (traversal-based) `alchemy` scaffold (which has been retired).
- The `starter` scaffold now uses URL dispatch by default.

1.2 (2011-09-12)

Features

- Route pattern replacement marker names can now begin with an underscore. See <https://github.com/Pylons/pyramid/issues/276>.

1.2b3 (2011-09-11)

Bug Fixes

- The route prefix was not taken into account when a static view was added in an “include”. See <https://github.com/Pylons/pyramid/issues/266>.

1.2b2 (2011-09-08)

Bug Fixes

- The 1.2b1 tarball was a brownbag (particularly for Windows users) because it contained filenames with stray quotation marks in inappropriate places. We depend on `setuptools-git` to produce release tarballs, and when it was run to produce the 1.2b1 tarball, it didn’t yet cope well with files present in git repositories with high-order characters in their filenames.

Documentation

- Minor tweaks to the “Introduction” narrative chapter example app and wording.

1.2b1 (2011-09-08)

Bug Fixes

- Sometimes falling back from territory translations (`de_DE`) to language translations (`de`) would not work properly when using a localizer. See <https://github.com/Pylons/pyramid/issues/263>
- The static file serving machinery could not serve files that started with a `.` (dot) character.
- Static files with high-order (super-ASCII) characters in their names could not be served by a static view. The static file serving machinery inappropriately URL-quoted path segments in filenames when asking for files from the filesystem.
- Within `pyramid.traversal.traversal_path`, canonicalize URL segments from UTF-8 to Unicode before checking whether a segment matches literally one of `.`, the empty string, or `..` in case there’s some sneaky way someone might tunnel those strings via UTF-8 that don’t match the literals before decoded.

Documentation

- Added a “What Makes Pyramid Unique” section to the Introduction narrative chapter.

1.2a6 (2011-09-06)

Bug Fixes

- `AuthTktAuthenticationPolicy` with a `reissue_time` interfered with logout. See <https://github.com/Pylons/pyramid/issues/262>.

Internal

- Internalize code previously depended upon as imports from the `paste.auth` module (future-proof).
- Replaced use of `paste.urlparser.StaticURLParser` with a derivative of Chris Rossi's "happy" static file serving code (futureproof).
- Fixed test suite; on some systems tests would fail due to indeterminate test run ordering and a double-push-single-pop of a shared test variable.

Behavior Differences

- An ETag header is no longer set when serving a static file. A Last-Modified header is set instead.
- Static file serving no longer supports the `wsgi.file_wrapper` extension.
- Instead of returning a 403 `Forbidden` error when a static file is served that cannot be accessed by the Pyramid process' user due to file permissions, an `IOError` (or similar) will be raised.

Scaffolds

- All scaffolds now send the `cache_max_age` parameter to the `add_static_view` method.

1.2a5 (2011-09-04)

Bug Fixes

- The `route_prefix` of a configurator was not properly taken into account when registering routes in certain circumstances. See <https://github.com/Pylons/pyramid/issues/260>

Dependencies

- The `zope.configuration` package is no longer a dependency.

1.2a4 (2011-09-02)

Features

- Support an `onerror` keyword argument to `pyramid.config.Configurator.scan()`. This `onerror` keyword argument is passed to `venusian.Scanner.scan()` to influence error behavior when an exception is raised during scanning.
- The `request_method` predicate argument to `pyramid.config.Configurator.add_view` and `pyramid.config.Configurator.add_route` is now permitted to be a tuple of HTTP method names. Previously it was restricted to being a string representing a single HTTP method name.
- Undeprecated `pyramid.traversal.find_model`, `pyramid.traversal.model_path`, `pyramid.traversal.model_path_tuple`, and `pyramid.url.model_url`, which were all deprecated in Pyramid 1.0. There's just not much cost to keeping them around forever as aliases to their renamed `resource_*` prefixed functions.
- Undeprecated `pyramid.view.bfg_view`, which was deprecated in Pyramid 1.0. This is a low-cost alias to `pyramid.view.view_config` which we'll just keep around forever.

Dependencies

- Pyramid now requires Venusian 1.0a1 or better to support the `onerror` keyword argument to `pyramid.config.Configurator.scan`.

1.2a3 (2011-08-29)

Bug Fixes

- Pyramid did not properly generate static URLs using `pyramid.url.static_url` when passed a caller-package relative path due to a refactoring done in 1.2a1.
- The `settings` object emitted a deprecation warning any time `__getattr__` was called upon it. However, there are legitimate situations in which `__getattr__` is called on arbitrary objects (e.g. `hasattr`). Now, the `settings` object only emits the warning upon successful lookup.

Internal

- Use `config.with_package` in `view_config` decorator rather than manufacturing a new renderer helper (cleanup).

1.2a2 (2011-08-27)

Bug Fixes

- When a `renderers=` argument is not specified to the `Configurator` constructor, eagerly register and commit the default renderer set. This permits the overriding of the default renderers, which was broken in 1.2a1 without a commit directly after `Configurator` construction.
- Mako rendering exceptions had the wrong value for an error message.
- An `include` could not set a root factory successfully because the `Configurator` constructor unconditionally registered one that would be treated as if it were “the word of the user”.

Features

- A session factory can now be passed in using the dotted name syntax.

1.2a1 (2011-08-24)

Features

- The `[pshell]` section in an ini configuration file now treats a `setup` key as a dotted name that points to a callable that is passed the bootstrap environment. It can mutate the environment as necessary for great justice.
- A new configuration setting named `pyramid.includes` is now available. It is described in the “Environment Variables and .ini Files Settings” narrative documentation chapter.
- Added a `route_prefix` argument to the `pyramid.config.Configurator.include` method. This argument allows you to compose URL dispatch applications together. See the section entitled “Using a Route Prefix to Compose Applications” in the “URL Dispatch” narrative documentation chapter.

- Added a `pyramid.security.NO_PERMISSION_REQUIRED` constant for use in `permission=` statements to view configuration. This constant has a value of the string `__no_permission_required__`. This string value was previously referred to in documentation; now the documentation uses the constant.
- Added a decorator-based way to configure a response adapter: `pyramid.response.response_adapter`. This decorator has the same use as `pyramid.config.Configurator.add_response_adapter` but it's declarative.
- The `pyramid.events.BeforeRender` event now has an attribute named `rendering_val`. This can be used to introspect the value returned by a view in a `BeforeRender` subscriber.
- New configurator directive: `pyramid.config.Configurator.add_tween`. This directive adds a “tween”. A “tween” is used to wrap the Pyramid router's primary request handling function. This is a feature may be used by Pyramid framework extensions, to provide, for example, view timing support and as a convenient place to hang bookkeeping code.

Tweens are further described in the narrative docs section in the Hooks chapter, named “Registering Tweens”.

- New paster command `paster ptweens`, which prints the current “tween” configuration for an application. See the section entitled “Displaying Tweens” in the Command-Line Pyramid chapter of the narrative documentation for more info.
- The Pyramid debug logger now uses the standard logging configuration (usually set up by Paste as part of startup). This means that output from e.g. `debug_notfound`, `debug_authorization`, etc. will go to the normal logging channels. The logger name of the debug logger will be the package name of the *caller* of the `Configurator`'s constructor.
- A new attribute is available on request objects: `exc_info`. Its value will be `None` until an exception is caught by the Pyramid router, after which it will be the result of `sys.exc_info()`.
- `pyramid.testing.DummyRequest` now implements the `add_finished_callback` and `add_response_callback` methods.
- New methods of the `pyramid.config.Configurator` class: `set_authentication_policy` and `set_authorization_policy`. These are meant to be consumed mostly by add-on authors.
- New Configurator method: `set_root_factory`.
- Pyramid no longer eagerly commits some default configuration statements at `Configurator` construction time, which permits values passed in as constructor arguments (e.g. `authentication_policy` and `authorization_policy`) to override the same settings obtained via an “include”.

- Better Mako rendering exceptions via `pyramid.mako_templating.MakoRenderingException`
- New request methods: `current_route_url`, `current_route_path`, and `static_path`.
- New functions in `pyramid.url`: `current_route_path` and `static_path`.
- The `pyramid.request.Request.static_url` API (and its brethren `pyramid.request.Request.static_path`, `pyramid.url.static_url`, and `pyramid.url.static_path`) now accept an absolute filename as a “path” argument. This will generate a URL to an asset as long as the filename is in a directory which was previously registered as a static view. Previously, trying to generate a URL to an asset using an absolute file path would raise a `ValueError`.
- The `RemoteUserAuthenticationPolicy`, `AuthTktAuthenticationPolicy`, and `SessionAuthenticationPolicy` constructors now accept an additional keyword argument named `debug`. By default, this keyword argument is `False`. When it is `True`, debug information will be sent to the Pyramid debug logger (usually on `stderr`) when the `authenticated_userid` or `effective_principals` method is called on any of these policies. The output produced can be useful when trying to diagnose authentication-related problems.
- New view predicate: `match_param`. Example: a view added via `config.add_view(aview, match_param='action=edit')` will be called only when the `request.matchdict` has a value inside it named `action` with a value of `edit`.

Internal

- The Pyramid “exception view” machinery is now implemented as a “tween” (`pyramid.tweens.excview_tween_factory`).
- `WSGIHTTPException` (`HTTPFound`, `HTTPNotFound`, etc) now has a new API named “prepare” which renders the body and content type when it is provided with a WSGI environ. Required for debug toolbar.
- Once `__call__` or `prepare` is called on a `WSGIHTTPException`, the body will be set, and subsequent calls to `__call__` will always return the same body. Delete the `body` attribute to rerender the exception body.
- Previously the `pyramid.events.BeforeRender` event *wrapped* a dictionary (it addressed it as its `_system` attribute). Now it *is* a dictionary (it inherits from `dict`), and it’s the value that is passed to templates as a top-level dictionary.

- The `route_url`, `route_path`, `resource_url`, `static_url`, and `current_route_url` functions in the `pyramid.url` package now delegate to a method on the request they've been passed, instead of the other way around. The `pyramid.request.Request` object now inherits from a mixin named `pyramid.url.URLMethodsMixin` to make this possible, and all url/path generation logic is embedded in this mixin.
- Refactor `pyramid.config` into a package.
- Removed the `_set_security_policies` method of the `Configurator`.
- Moved the `StaticURLInfo` class from `pyramid.static` to `pyramid.config.views`.
- Move the `Settings` class from `pyramid.settings` to `pyramid.config.settings`.
- Move the `OverrideProvider`, `PackageOverrides`, `DirectoryOverride`, and `FileOverride` classes from `pyramid.asset` to `pyramid.config.assets`.

Deprecations

- All Pyramid-related deployment settings (e.g. `debug_all`, `debug_notfound`) are now meant to be prefixed with the prefix `pyramid..` For example: `debug_all` -> `pyramid.debug_all`. The old non-prefixed settings will continue to work indefinitely but supplying them may eventually print a deprecation warning. All scaffolds and tutorials have been changed to use prefixed settings.
- The `settings` dictionary now raises a deprecation warning when you attempt to access its values via `__getattr__` instead of via `__getitem__`.

Backwards Incompatibilities

- If a string is passed as the `debug_logger` parameter to a `Configurator`, that string is considered to be the name of a global Python logger rather than a dotted name to an instance of a logger.
- The `pyramid.config.Configurator.include` method now accepts only a single callable argument (a sequence of callables used to be permitted). If you are passing more than one callable to `pyramid.config.Configurator.include`, it will break. You now must now instead make a separate call to the method for each callable. This change was introduced to support the `route_prefix` feature of `include`.
- It may be necessary to more strictly order configuration route and view statements when using an “autocommitting” `Configurator`. In the past, it was possible to add a view which named a route name before adding a route with that name when you used an autocommitting configurator. For example:

```
config = Configurator(autocommit=True)
config.add_view('my.pkg.someview', route_name='foo')
config.add_route('foo', '/foo')
```

The above will raise an exception when the view attempts to add itself. Now you must add the route before adding the view:

```
config = Configurator(autocommit=True)
config.add_route('foo', '/foo')
config.add_view('my.pkg.someview', route_name='foo')
```

This won't effect "normal" users, only people who have legacy BFG codebases that used an autocommitting configurator and possibly tests that use the configurator API (the configurator returned by `pyramid.testing.setUp` is an autocommitting configurator). The right way to get around this is to use a non-autocommitting configurator (the default), which does not have these directive ordering requirements.

- The `pyramid.config.Configurator.add_route` directive no longer returns a route object. This change was required to make route vs. view configuration processing work properly.

Documentation

- Narrative and API documentation which used the `route_url`, `route_path`, `resource_url`, `static_url`, and `current_route_url` functions in the `pyramid.url` package have now been changed to use eponymous methods of the request instead.
- Added a section entitled "Using a Route Prefix to Compose Applications" to the "URL Dispatch" narrative documentation chapter.
- Added a new module to the API docs: `pyramid.tweens`.
- Added a "Registering Tweens" section to the "Hooks" narrative chapter.
- Added a "Displaying Tweens" section to the "Command-Line Pyramid" narrative chapter.
- Added documentation for the `pyramid.tweens` and `pyramid.includes` configuration settings to the "Environment Variables and .ini Files Settings" chapter.
- Added a Logging chapter to the narrative docs (based on the Pylons logging docs, thanks Phil).

- Added a Paste chapter to the narrative docs (moved content from the Project chapter).
- Added the `pyramid.interfaces.IDict` interface representing the methods of a dictionary, for documentation purposes only (`IMultiDict` and `IBeforeRender` inherit from it).
- All tutorials now use - The `route_url`, `route_path`, `resource_url`, `static_url`, and `current_route_url` methods of the request rather than the function variants imported from `pyramid.url`.
- The ZODB wiki tutorial now uses the `pyramid_zodbconn` package rather than the `repoze.zodbconn` package to provide ZODB integration.

Dependency Changes

- Pyramid now relies on PasteScript \geq 1.7.4. This version contains a feature important for allowing flexible logging configuration.

Scaffolds

- All scaffolds now use the `pyramid_tm` package rather than the `repoze.tm2` middleware to manage transaction management.
- The ZODB scaffold now uses the `pyramid_zodbconn` package rather than the `repoze.zodbconn` package to provide ZODB integration.
- All scaffolds now use the `pyramid_debugtoolbar` package rather than the `WebError` package to provide interactive debugging features.
- Projects created via a scaffold no longer depend on the `WebError` package at all; configuration in the `production.ini` file which used to require its `error_catcher` middleware has been removed. Configuring error catching / email sending is now the domain of the `pyramid_exclog` package (see http://docs.pylonsproject.org/projects/pyramid_exclog/en/latest/).

Bug Fixes

- Fixed an issue with the default renderer not working at certain times. See <https://github.com/Pylons/pyramid/issues/249>

1.1 (2011-07-22)

Features

- Added the `pyramid.renderers.null_renderer` object as an API. The null renderer is an object that can be used in advanced integration cases as input to the view configuration `renderer=` argument. When the null renderer is used as a view renderer argument, Pyramid avoids converting the view callable result into a Response object. This is useful if you want to reuse the view configuration and lookup machinery outside the context of its use by the Pyramid router. This feature was added for consumption by the `pyramid_rpc` package, which uses view configuration and lookup outside the context of a router in exactly this way. `pyramid_rpc` has been broken under 1.1 since 1.1b1; adding it allows us to make it work again.
- Change all scaffolding templates that point to `docs.pylonsproject.org` to use `/projects/pyramid/current` rather than `/projects/pyramid/dev`.

Internals

- Remove `compat` code that served only the purpose of providing backwards compatibility with Python 2.4.
- Add a deprecation warning for non-API function `pyramid.renderers.renderer_from_name` which has seen use in the wild.
- Add a `clone` method to `pyramid.renderers.RendererHelper` for use by the `pyramid.view.view_config` decorator.

Documentation

- Fixed two typos in wiki2 (SQLA + URL Dispatch) tutorial.
- Reordered chapters in narrative section for better new user friendliness.
- Added more indexing markers to sections in documentation.

1.1b4 (2011-07-18)

Documentation

- Added a section entitled “Writing a Script” to the “Command-Line Pyramid” chapter.

Backwards Incompatibilities

- We added the `pyramid.scripting.make_request` API too hastily in 1.1b3. It has been removed. Sorry for any inconvenience. Use the `pyramid.request.Request.blank` API instead.

Features

- The `paster pshell`, `paster pviews`, and `paster proutes` commands each now under the hood uses `pyramid.paster.bootstrap`, which makes it possible to supply an `.ini` file without naming the “right” section in the file that points at the actual Pyramid application. Instead, you can generally just run `paster {pshell|proutes|pviews} development.ini` and it will do mostly the right thing.

Bug Fixes

- Omit custom environ variables when rendering a custom exception template in `pyramid.httpexceptions.WSGIHTTPException._set_default_attrs`; stringifying these may trigger code that should not be executed; see <https://github.com/Pylons/pyramid/issues/239>

1.1b3 (2011-07-15)

Features

- Fix corner case to ease semifunctional testing of views: create a new `renderinfo` to clear out old registry on a rescan. See <https://github.com/Pylons/pyramid/pull/234>.

- New API class: `pyramid.static.static_view`. This supersedes the deprecated `pyramid.view.static` class. `pyramid.static.static_view` by default serves up documents as the result of the request's `path_info`, attribute rather than its `subpath` attribute (the inverse was true of `pyramid.view.static`, and still is). `pyramid.static.static_view` exposes a `use_subpath` flag for use when you want the static view to behave like the older deprecated version.
- A new API function `pyramid.paster.bootstrap` has been added to make writing scripts that bootstrap a Pyramid environment easier, e.g.:

```
from pyramid.paster import bootstrap
info = bootstrap('/path/to/my/development.ini')
request = info['request']
print request.route_url('myroute')
```

- A new API function `pyramid.scripting.prepare` has been added. It is a lower-level analogue of `pyramid.paster.bootstrap` that accepts a request and a registry instead of a config file argument, and is used for the same purpose:

```
from pyramid.scripting import prepare
info = prepare(registry=myregistry)
request = info['request']
print request.route_url('myroute')
```

- A new API function `pyramid.scripting.make_request` has been added. The resulting request will have a `registry` attribute. It is meant to be used in conjunction with `pyramid.scripting.prepare` and/or `pyramid.paster.bootstrap` (both of which accept a request as an argument):

```
from pyramid.scripting import make_request
request = make_request('/')
```

- New API attribute `pyramid.config.global_registries` is an iterable object that contains references to every Pyramid registry loaded into the current process via `pyramid.config.Configurator.make_app`. It also has a `last` attribute containing the last registry loaded. This is used by the scripting machinery, and is available for introspection.

Deprecations

- The `pyramid.view.static` class has been deprecated in favor of the newer `pyramid.static.static_view` class. A deprecation warning is raised when it is used. You should replace it with a reference to `pyramid.static.static_view` with the `use_subpath=True` argument.

Bug Fixes

- Without a mo-file loaded for the combination of domain/locale, `pyramid.i18n.Localizer.pluralize` run using that domain/locale combination raised an inscrutable “translations object has no attr ‘plural’” error. Now, instead it “works” (it uses a germanic pluralization by default). It’s nonsensical to try to pluralize something without translations for that locale/domain available, but this behavior matches the behavior of `pyramid.i18n.Localizer.translate` so it’s at least consistent; see <https://github.com/Pylons/pyramid/issues/235>.

1.1b2 (2011-07-13)

Features

- New environment setting `PYRAMID_PREVENT_HTTP_CACHE` and new configuration file value `prevent_http_cache`. These are synonymous and allow you to prevent HTTP cache headers from being set by Pyramid’s `http_cache` machinery globally in a process. see the “Influencing HTTP Caching” section of the “View Configuration” narrative chapter and the detailed documentation for this setting in the “Environment Variables and Configuration Settings” narrative chapter.

Behavior Changes

- Previously, If a `BeforeRender` event subscriber added a value via the `__setitem__` or `update` methods of the event object with a key that already existed in the renderer globals dictionary, a `KeyError` was raised. With the deprecation of the “`add_renderer_globals`” feature of the configurator, there was no way to override an existing value in the renderer globals dictionary that already existed. Now, the event object will overwrite an older value that is already in the globals dictionary when its `__setitem__` or `update` is called (as well as the new `setdefault` method), just like a plain old dictionary. As a result, for maximum interoperability with other third-party subscribers, if you write an event subscriber meant to be used as a `BeforeRender` subscriber, your subscriber code will now need to (using `.get` or `__contains__` of the event object) ensure no value already exists in the renderer globals dictionary before setting an overriding value.

Bug Fixes

- The `Configurator.add_route` method allowed two routes with the same route to be added without an intermediate `config.commit()`. If you now receive a `ConfigurationError` at startup time that appears to be `add_route` related, you'll need to either a) ensure that all of your route names are unique or b) call `config.commit()` before adding a second route with the name of a previously added name or c) use a `Configurator` that works in `autocommit` mode.
- The `pyramid_routesalchemy` and `pyramid_alchemy` scaffolds inappropriately used `DBSession.rollback()` instead of `transaction.abort()` in one place.
- We now clear `request.response` before we invoke an exception view; an exception view will be working with a `request.response` that has not been touched by any code prior to the exception.
- Views associated with routes with spaces in the route name may not have been looked up correctly when using Pyramid with `zope.interface` 3.6.4 and better. See <https://github.com/Pylons/pyramid/issues/232>.

Documentation

- Wiki2 (SQLAlchemy + URL Dispatch) tutorial `models.initialize_sql` didn't match the `pyramid_routesalchemy` scaffold function of the same name; it didn't get synchronized when it was changed in the scaffold.
- New documentation section in View Configuration narrative chapter: "Influencing HTTP Caching".

1.1b1 (2011-07-10)

Features

- It is now possible to invoke `paster pshell` even if the `paste.ini` file section name pointed to in its argument is not actually a Pyramid WSGI application. The shell will work in a degraded mode, and will warn the user. See "The Interactive Shell" in the "Creating a Pyramid Project" narrative documentation section.
- `paster pshell` now offers more built-in global variables by default (including `app` and `settings`). See "The Interactive Shell" in the "Creating a Pyramid Project" narrative documentation section.

- It is now possible to add a `[pshell]` section to your application's `.ini` configuration file, which influences the global names available to a `pshell` session. See “Extending the Shell” in the “Creating a Pyramid Project” narrative documentation chapter.
- The `config.scan` method has grown a `**kw` argument. `kw` argument represents a set of keyword arguments to pass to the `Venusian Scanner` object created by Pyramid. (See the `Venusian` documentation for more information about `Scanner`).
- New request property: `json_body`. This property will return the JSON-decoded variant of the request body. If the request body is not well-formed JSON, this property will raise an exception.
- A new value `http_cache` can be used as a view configuration parameter.

When you supply an `http_cache` value to a view configuration, the `Expires` and `Cache-Control` headers of a response generated by the associated view callable are modified. The value for `http_cache` may be one of the following:

- A nonzero integer. If it's a nonzero integer, it's treated as a number of seconds. This number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=3600` instructs the requesting browser to ‘cache this response for an hour, please’.
- A `datetime.timedelta` instance. If it's a `datetime.timedelta` instance, it will be converted into a number of seconds, and that number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=datetime.timedelta(days=1)` instructs the requesting browser to ‘cache this response for a day, please’.
- Zero (0). If the value is zero, the `Cache-Control` and `Expires` headers present in all responses from this view will be composed such that client browser cache (and any intermediate caches) are instructed to never cache the response.
- A two-tuple. If it's a two tuple (e.g. `http_cache=(1, {'public':True})`), the first value in the tuple may be a nonzero integer or a `datetime.timedelta` instance; in either case this value will be used as the number of seconds to cache the response. The second value in the tuple must be a dictionary. The values present in the dictionary will be used as input to the `Cache-Control` response header. For example: `http_cache=(3600, {'public':True})` means ‘cache for an hour, and add `public` to the `Cache-Control` header of the response’. All keys and values supported by the `webob.cachecontrol.CacheControl` interface may be added to the dictionary. Supplying `{'public':True}` is equivalent to calling `response.cache_control.public = True`.

Providing a non-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value)` within your view's body.

Providing a two-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value[0], **value[1])` within your view's body.

If you wish to avoid influencing, the `Expires` header, and instead wish to only influence `Cache-Control` headers, pass a tuple as `http_cache` with the first element of `None`, e.g.: `(None, {'public':True})`.

Bug Fixes

- Framework wrappers of the original view (such as `http_cached` and so on) relied on being able to trust that the response they were receiving was an `IResponse`. It wasn't always, because the response was resolved by the router instead of early in the view wrapping process. This has been fixed.

Documentation

- Added a section in the “Webob” chapter named “Dealing With A JSON-Encoded Request Body” (usage of `request.json_body`).

Behavior Changes

- The `paster pshell`, `paster proutes`, and `paster pviews` commands now take a single argument in the form `/path/to/config.ini#sectionname` rather than the previous 2-argument spelling `/path/to/config.ini sectionname`. `#sectionname` may be omitted, in which case `#main` is assumed.

1.1a4 (2011-07-01)

Bug Fixes

- `pyramid.testing.DummyRequest` now raises deprecation warnings when attributes deprecated for `pyramid.request.Request` are accessed (like `response_content_type`). This is for the benefit of folks running unit tests which use `DummyRequest` instead of a “real” request, so they know things are deprecated without necessarily needing a functional test suite.
- The `pyramid.events.subscriber` directive behaved contrary to the documentation when passed more than one interface object to its constructor. For example, when the following listener was registered:

```
@subscriber(IFoo, IBar)
def expects_ifoo_events_and_ibar_events(event):
    print event
```

The Events chapter docs claimed that the listener would be registered and listening for both `IFoo` and `IBar` events. Instead, it registered an “object event” subscriber which would only be called if an `IObjectEvent` was emitted where the object interface was `IFoo` and the event interface was `IBar`.

The behavior now matches the documentation. If you were relying on the buggy behavior of the 1.0 `subscriber` directive in order to register an object event subscriber, you must now pass a sequence to indicate you’d like to register a subscriber for an object event. e.g.:

```
@subscriber([IFoo, IBar])
def expects_object_event(object, event):
    print object, event
```

Features

- Add JSONP renderers (see “JSONP renderers” in the Renderers chapter of the documentation).

Deprecations

- Deprecated the `set_renderer_globals_factory` method of the `Configurator` and the `renderer_globals` `Configurator` constructor parameter.

Documentation

- The Wiki and Wiki2 tutorial “Tests” chapters each had two bugs: neither did told the user to depend on WebTest, and 2 tests failed in each as the result of changes to Pyramid itself. These issues have been fixed.
- Move 1.0.X CHANGES.txt entries to HISTORY.txt.

1.1a3 (2011-06-26)

Features

- Added `mako.preprocessor` config file parameter; allows for a Mako preprocessor to be specified as a Python callable or Python dotted name. See <https://github.com/Pylons/pyramid/pull/183> for rationale.

Bug fixes

- Pyramid would raise an `AttributeError` in the Configurator when attempting to set a `__text__` attribute on a custom predicate that was actually a classmethod. See <https://github.com/Pylons/pyramid/pull/217>.
- Accessing or setting deprecated `response_*` attrs on request (e.g. `response_content_type`) now issues a deprecation warning at access time rather than at rendering time.

1.1a2 (2011-06-22)

Bug Fixes

- 1.1a1 broke Akhet by not providing a backwards compatibility import shim for `pyramid.paster.PyramidTemplate`. Now one has been added, although a deprecation warning is emitted when Akhet imports it.
- If multiple specs were provided in a single call to `config.add_translation_dirs`, the directories were inserted into the beginning of the directory list in the wrong order: they were inserted in the reverse of the order they were provided in the `*specs` list (items later in the list were added before ones earlier in the list). This is now fixed.

Backwards Incompatibilities

- The pyramid Router attempted to set a value into the key `environ['repoze.bfg.message']` when it caught a view-related exception for backwards compatibility with applications written for `repoze.bfg` during error handling. It did this by using code that looked like so:

```
# "why" is an exception object
try:
    msg = why[0]
except:
    msg = ''

environ['repoze.bfg.message'] = msg
```

Use of the value `environ['repoze.bfg.message']` was docs-deprecated in Pyramid 1.0. Our standing policy is to not remove features after a deprecation for two full major releases, so this code was originally slated to be removed in Pyramid 1.2. However, computing the `repoze.bfg.message` value was the source of at least one bug found in the wild (<https://github.com/Pylons/pyramid/issues/199>), and there isn't a foolproof way to both preserve backwards compatibility and to fix the bug. Therefore, the code which sets the value has been removed in this release. Code in exception views which relies on this value's presence in the environment should now use the `exception` attribute of the request (e.g. `request.exception[0]`) to retrieve the message instead of relying on `request.environ['repoze.bfg.message']`.

1.1a1 (2011-06-20)

Documentation

- The term “template” used to refer to both “paster templates” and “rendered templates” (templates created by a rendering engine. i.e. Mako, Chameleon, Jinja, etc.). “Paster templates” will now be referred to as “scaffolds”, whereas the name for “rendered templates” will remain as “templates.”
- The `wiki` (ZODB+Traversal) tutorial was updated slightly.
- The `wiki2` (SQLA+URL Dispatch) tutorial was updated slightly.
- `Make` `pyramid.interfaces.IAuthenticationPolicy` and `pyramid.interfaces.IAuthorizationPolicy` public interfaces, and refer to them within the `pyramid.authentication` and `pyramid.authorization` API docs.

- Render the function definitions for each exposed interface in `pyramid.interfaces`.
- Add missing docs reference to `pyramid.config.Configurator.set_view_mapper` and refer to it within Hooks chapter section named “Using a View Mapper”.
- Added section to the “Environment Variables and .ini File Settings” chapter in the narrative documentation section entitled “Adding a Custom Setting”.
- Added documentation for a “multidict” (e.g. the API of `request.POST`) as interface API documentation.
- Added a section to the “URL Dispatch” narrative chapter regarding the new “static” route feature.
- Added “What’s New in Pyramid 1.1” to HTML rendering of documentation.
- Added API docs for `pyramid.authentication.SessionAuthenticationPolicy`.
- Added API docs for `pyramid.httpexceptions.exception_response`.
- Added “HTTP Exceptions” section to Views narrative chapter including a description of `pyramid.httpexceptions.exception_response`.

Features

- Add support for language fallbacks: when trying to translate for a specific territory (such as `en_GB`) fall back to translations for the language (ie `en`). This brings the translation behaviour in line with GNU gettext and fixes partially translated texts when using C extensions.
- New authentication policy: `pyramid.authentication.SessionAuthenticationPolicy`, which uses a session to store credentials.
- Accessing the `response` attribute of a `pyramid.request.Request` object (e.g. `request.response` within a view) now produces a new `pyramid.response.Response` object. This feature is meant to be used mainly when a view configured with a renderer needs to set response attributes: all renderers will use the `Response` object implied by `request.response` as the response object returned to the router.

`request.response` can also be used by code in a view that does not use a renderer, however the response object that is produced by `request.response` must be returned when a renderer is not in play (it is not a “global” response).

- Integers and longs passed as elements to `pyramid.url.resource_url` or `pyramid.request.Request.resource_url` e.g. `resource_url(context, request, 1, 2)` (1 and 2 are the elements) will now be converted implicitly to strings in the result. Previously passing integers or longs as elements would cause a `TypeError`.
- `pyramid_alchemy` paster template now uses `query.get` rather than `query.filter_by` to take better advantage of identity map caching.
- `pyramid_alchemy` paster template now has unit tests.
- Added `pyramid.i18n.make_localizer` API (broken out from `get_localizer` guts).
- An exception raised by a `NewRequest` event subscriber can now be caught by an exception view.
- It is now possible to get information about why Pyramid raised a `Forbidden` exception from within an exception view. The `ACLDenied` object returned by the `permits` method of each stock authorization policy (`pyramid.interfaces.IAuthorizationPolicy.permits`) is now attached to the `Forbidden` exception as its `result` attribute. Therefore, if you've created a `Forbidden` exception view, you can see the ACE, ACL, permission, and principals involved in the request as e.g. `context.result.permission`, `context.result.acl`, etc within the logic of the `Forbidden` exception view.
- Don't explicitly prevent the `timeout` from being lower than the `reissue_time` when setting up an `AuthTktAuthenticationPolicy` (previously such a configuration would raise a `ValueError`, now it's allowed, although typically nonsensical). Allowing the nonsensical configuration made the code more understandable and required fewer tests.
- A new paster command named `paster pviews` was added. This command prints a summary of potentially matching views for a given path. See the section entitled "Displaying Matching Views for a Given URL" in the "View Configuration" chapter of the narrative documentation for more information.
- The `add_route` method of the `Configurator` now accepts a `static` argument. If this argument is `True`, the added route will never be considered for matching when a request is handled. Instead, it will only be useful for URL generation via `route_url` and `route_path`. See the section entitled "Static Routes" in the URL Dispatch narrative chapter for more information.
- A default exception view for the context `pyramid.interfaces.IExceptionResponse` is now registered by default. This means that an instance of any exception response class imported from `pyramid.httpexceptions` (such as `HTTPFound`) can now be raised from within view code; when raised, this exception view will render the exception to a response.
- A function named `pyramid.httpexceptions.exception_response` is a shortcut that can be used to create HTTP exception response objects using an HTTP integer status code.

- The `Configurator` now accepts an additional keyword argument named `exceptionresponse_view`. By default, this argument is populated with a default exception view function that will be used when a response is raised as an exception. When `None` is passed for this value, an exception view for responses will not be registered. Passing `None` returns the behavior of raising an HTTP exception to that of Pyramid 1.0 (the exception will propagate to middleware and to the WSGI server).
- The `pyramid.request.Request` class now has a `ResponseClass` interface which points at `pyramid.response.Response`.
- The `pyramid.response.Response` class now has a `RequestClass` interface which points at `pyramid.request.Request`.
- It is now possible to return an arbitrary object from a Pyramid view callable even if a renderer is not used, as long as a suitable adapter to `pyramid.interfaces.IResponse` is registered for the type of the returned object by using the new `pyramid.config.Configurator.add_response_adapter` API. See the section in the Hooks chapter of the documentation entitled “Changing How Pyramid Treats View Responses”.
- The Pyramid router will now, by default, call the `__call__` method of `WebOb` response objects when returning a WSGI response. This means that, among other things, the `conditional_response` feature of `WebOb` response objects will now behave properly.
- New method named `pyramid.request.Request.is_response`. This method should be used instead of the `pyramid.view.is_response` function, which has been deprecated.

Bug Fixes

- URL pattern markers used in URL dispatch are permitted to specify a custom regex. For example, the pattern `/ {foo:\d+}` means to match `/12345` (`foo==12345` in the match dictionary) but not `/abc`. However, custom regexes in a pattern marker which used squiggly brackets did not work. For example, `/ {foo:\d{4}}` would fail to match `/1234` and `/ {foo:\d{1,2}}` would fail to match `/1` or `/11`. One level of inner squiggly brackets is now recognized so that the prior two patterns given as examples now work. See also <https://github.com/Pylons/pyramid/issues/#issue/123>.
- Don't send port numbers along with domain information in cookies set by `AuthTktCookieHelper` (see <https://github.com/Pylons/pyramid/issues/131>).
- `pyramid.url.route_path` (and the shortcut `pyramid.request.Request.route_url` method) now include the WSGI `SCRIPT_NAME` at the front of the path if it is not empty (see <https://github.com/Pylons/pyramid/issues/135>).

- `pyramid.testing.DummyRequest` now has a `script_name` attribute (the empty string).
- Don't quote `:@&+$`, symbols in `*elements` passed to `pyramid.url.route_url` or `pyramid.url.resource_url` (see <https://github.com/Pylons/pyramid/issues#issue/141>).
- Include `SCRIPT_NAME` in redirects issued by `pyramid.view.append_slash_notfound_view` (see <https://github.com/Pylons/pyramid/issues#issue/149>).
- Static views registered with `config.add_static_view` which also included a `permission` keyword argument would not work as expected, because `add_static_view` also registered a route factory internally. Because a route factory was registered internally, the context checked by the Pyramid permission machinery never had an ACL. `add_static_view` no longer registers a route with a factory, so the default root factory will be used.
- `config.add_static_view` now passes extra keyword arguments it receives to `config.add_route` (calling `add_static_view` is mostly logically equivalent to adding a view of the type `pyramid.static.static_view` hooked up to a route with a subpath). This makes it possible to pass e.g., `factory=` to `add_static_view` to protect a particular static view with a custom ACL.
- `testing.DummyRequest` used the wrong registry (the global registry) as `self.registry` if a dummy request was created *before* `testing.setUp` was executed (`testing.setUp` pushes a local registry onto the threadlocal stack). Fixed by implementing `registry` as a property for `DummyRequest` instead of eagerly assigning an attribute. See also <https://github.com/Pylons/pyramid/issues/165>
- When visiting a URL that represented a static view which resolved to a subdirectory, the `index.html` of that subdirectory would not be served properly. Instead, a redirect to `/subdir` would be issued. This has been fixed, and now visiting a subdirectory that contains an `index.html` within a static view returns the `index.html` properly. See also <https://github.com/Pylons/pyramid/issues/67>.
- Redirects issued by a static view did not take into account any existing `SCRIPT_NAME` (such as one set by a url mapping composite). Now they do.
- The `pyramid.wsgi.wsgiapp2` decorator did not take into account the `SCRIPT_NAME` in the origin request.
- The `pyramid.wsgi.wsgiapp2` decorator effectively only worked when it decorated a view found via traversal; it ignored the `PATH_INFO` that was part of a url-dispatch-matched view.

Deprecations

- Deprecated all assignments to `request.response_*` attributes (for example `request.response_content_type = 'foo'` is now deprecated). Assignments and mutations of assignable request attributes that were considered by the framework for response influence are now deprecated: `response_content_type`, `response_headerlist`, `response_status`, `response_charset`, and `response_cache_for`. Instead of assigning these to the request object for later detection by the rendering machinery, users should use the appropriate API of the Response object created by accessing `request.response` (e.g. code which does `request.response_content_type = 'abc'` should be changed to `request.response.content_type = 'abc'`).
- Passing view-related parameters to `pyramid.config.Configurator.add_route` is now deprecated. Previously, a view was permitted to be connected to a route using a set of `view*` parameters passed to the `add_route` method of the Configurator. This was a shorthand which replaced the need to perform a subsequent call to `add_view`. For example, it was valid (and often recommended) to do:

```
config.add_route('home', '/', view='mypackage.views.myview',
                 view_renderer='some/renderer.pt')
```

Passing `view*` arguments to `add_route` is now deprecated in favor of connecting a view to a predefined route via `Configurator.add_view` using the route's `route_name` parameter. As a result, the above example should now be spelled:

```
config.add_route('home', '/')
config.add_view('mypackage.views.myview', route_name='home')
               renderer='some/renderer.pt')
```

This deprecation was done to reduce confusion observed in IRC, as well as to (eventually) reduce documentation burden (see also <https://github.com/Pylons/pyramid/issues/164>). A deprecation warning is now issued when any view-related parameter is passed to `Configurator.add_route`.

- Passing an environ dictionary to the `__call__` method of a “traverser” (e.g. an object that implements `pyramid.interfaces.ITraverser` such as an instance of `pyramid.traversal.ResourceTreeTraverser`) as its `request` argument now causes a deprecation warning to be emitted. Consumer code should pass a `request` object instead. The fact that passing an environ dict is permitted has been documentation-deprecated since `repoze.bfg` 1.1, and this capability will be removed entirely in a future version.

- The following (undocumented, dictionary-like) methods of the `pyramid.request.Request` object have been deprecated: `__contains__`, `__delitem__`, `__getitem__`, `__iter__`, `__setitem__`, `get`, `has_key`, `items`, `iteritems`, `itervalues`, `keys`, `pop`, `popitem`, `setdefault`, `update`, and `values`. Usage of any of these methods will cause a deprecation warning to be emitted. These methods were added for internal compatibility in `repoze.bfg` 1.1 (code that currently expects a request object expected an `environ` object in BFG 1.0 and before). In a future version, these methods will be removed entirely.
- Deprecated `pyramid.view.is_response` function in favor of (newly-added) `pyramid.request.Request.is_response` method. Determining if an object is truly a valid response object now requires access to the registry, which is only easily available as a request attribute. The `pyramid.view.is_response` function will still work until it is removed, but now may return an incorrect answer under some (very uncommon) circumstances.

Behavior Changes

- The default Mako renderer is now configured to escape all HTML in expression tags. This is intended to help prevent XSS attacks caused by rendering unsanitized input from users. To revert this behavior in user's templates, they need to filter the expression through the 'n' filter. For example, `${ myhtml | n }`. See <https://github.com/Pylons/pyramid/issues/193>.
- A custom request factory is now required to return a request object that has a `response` attribute (or "reified"/lazy property) if they the request is meant to be used in a view that uses a renderer. This `response` attribute should be an instance of the class `pyramid.response.Response`.
- The JSON and string renderer factories now assign to `request.response.content_type` rather than `request.response_content_type`.
- Each built-in renderer factory now determines whether it should change the content type of the response by comparing the response's content type against the response's default content type; if the content type is the default content type (usually `text/html`), the renderer changes the content type (to `application/json` or `text/plain` for JSON and string renderers respectively).
- The `pyramid.wsgi.wsgiapp2` now uses a slightly different method of figuring out how to "fix" `SCRIPT_NAME` and `PATH_INFO` for the downstream application. As a result, those values may differ slightly from the perspective of the downstream application (for example, `SCRIPT_NAME` will now never possess a trailing slash).
- Previously, `pyramid.request.Request` inherited from `webob.request.Request` and implemented `__getattr__`, `__setattr__` and `__delattr__` itself in order to override "adhoc attr" WebOb behavior where attributes of the request are stored in the environ. Now, `pyramid.request.Request` object inherits from (the more recent) `webob.request.BaseRequest` instead of `webob.request.Request`, which provides the same behavior. `pyramid.request.Request` no longer implements its own `__getattr__`, `__setattr__` or `__delattr__` as a result.

- `pyramid.response.Response` is now a *subclass* of `webob.response.Response` (in order to directly implement the `pyramid.interfaces.IResponse` interface).
- The “exception response” objects importable from `pyramid.httpexceptions` (e.g. `HTTPNotFound`) are no longer just import aliases for classes that actually live in `webob.exc`. Instead, we’ve defined our own exception classes within the module that mirror and emulate the `webob.exc` exception response objects almost entirely. See the “Design Defense” doc section named “Pyramid Uses its Own HTTP Exception Classes” for more information.

Backwards Incompatibilities

- Pyramid no longer supports Python 2.4. Python 2.5 or better is required to run Pyramid 1.1+.
- The Pyramid router now, by default, expects response objects returned from view callables to implement the `pyramid.interfaces.IResponse` interface. Unlike the Pyramid 1.0 version of this interface, objects which implement `IResponse` now must define a `__call__` method that accepts `environ` and `start_response`, and which returns an `app_iter` iterable, among other things. Previously, it was possible to return any object which had the three `WebOb` `app_iter`, `headerlist`, and `status` attributes as a response, so this is a backwards incompatibility. It is possible to get backwards compatibility back by registering an adapter to `IResponse` from the type of object you’re now returning from view callables. See the section in the Hooks chapter of the documentation entitled “Changing How Pyramid Treats View Responses”.
- The `pyramid.interfaces.IResponse` interface is now much more extensive. Previously it defined only `app_iter`, `status` and `headerlist`; now it is basically intended to directly mirror the `webob.Response` API, which has many methods and attributes.
- The `pyramid.httpexceptions` classes named `HTTPFound`, `HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPSeeOther`, `HTTPUseProxy`, and `HTTPTemporaryRedirect` now accept `location` as their first positional argument rather than detail. This means that you can do, e.g. `return pyramid.httpexceptions.HTTPFound('http://foo')` rather than `return pyramid.httpexceptions.HTTPFound(location='http://foo')` (the latter will of course continue to work).

Dependencies

- Pyramid now depends on `WebOb` `>= 1.0.2` as tests depend on the bugfix in that release: “Fix handling of WSGI environs with missing `SCRIPT_NAME`”. (Note that in reality, everyone should probably be using 1.0.4 or better though, as `WebOb` 1.0.2 and 1.0.3 were effectively brownbag releases.)

1.0 (2011-01-30)

Documentation

- Fixed bug in ZODB Wiki tutorial (missing dependency on `docutils` in “models” step within `setup.py`).
- Removed API documentation for `pyramid.testing` APIs named `registerDummySecurityPolicy`, `registerResources`, `registerModels`, `registerEventListener`, `registerTemplateRenderer`, `registerDummyRenderer`, `registerView`, `registerUtility`, `registerAdapter`, `registerSubscriber`, `registerRoute`, and `registerSettings`.
- Moved “Using ZODB With ZEO” and “Using repoze.catalog Within Pyramid” tutorials out of core documentation and into the Pyramid Tutorials site (http://docs.pylonsproject.org/projects/pyramid_tutorials/en/latest/).
- Changed “Cleaning up After a Request” section in the URL Dispatch chapter to use `request.add_finished_callback` instead of jamming an object with a `__del__` into the WSGI environment.
- Remove duplication of `add_route` API documentation from URL Dispatch narrative chapter.
- Remove duplication of API and narrative documentation in `pyramid.view.view_config` API docs by pointing to `pyramid.config.add_view` documentation and narrative chapter documentation.
- Removed some API documentation duplicated in narrative portions of documentation
- Removed “Overall Flow of Authentication” from SQLAlchemy + URL Dispatch wiki tutorial due to print space concerns (moved to Pyramid Tutorials site).

Bug Fixes

- Deprecated-since-BFG-1.2 APIs from `pyramid.testing` now properly emit deprecation warnings.
- Added `egg:repoze.retry#retry` middleware to the WSGI pipeline in ZODB templates (retry ZODB conflict errors which occur in normal operations).
- Removed duplicate implementations of `is_response`. Two competing implementations existed: one in `pyramid.config` and one in `pyramid.view`. Now the one defined in `pyramid.view` is used internally by `pyramid.config` and continues to be advertised as an API.

1.0b3 (2011-01-28)

Bug Fixes

- Use `©` instead of copyright symbol in paster templates / tutorial templates for the benefit of folks who cutnpaste and save to a non-UTF8 format.
- `pyramid.view.append_slash_notfound_view` now preserves GET query parameters across redirects.

Documentation

- Beef up documentation related to `set_default_permission`: explicitly mention that default permissions also protect exception views.
- Paster templates and tutorials now use spaces instead of tabs in their HTML templates.

1.0b2 (2011-01-24)

Bug Fixes

- The `production.ini` generated by all paster templates now have an effective logging level of `WARN`, which prevents e.g. SQLAlchemy statement logging and other inappropriate output.
- The `production.ini` of the `pyramid_routesalchemy` and `pyramid_alchemy` paster templates did not have a `sqlalchemy` logger section, preventing paster serve `production.ini` from working.
- The `pyramid_routesalchemy` and `pyramid_alchemy` paster templates used the `{{package}}` variable in a place where it should have used the `{{project}}` variable, causing applications created with uppercase letters e.g. `paster create -t pyramid_routesalchemy Dibbus` to fail to start when `paster serve development.ini` was used against the result. See <https://github.com/Pylons/pyramid/issues/#issue/107>
- The `render_view` method of `pyramid.renderers.RendererHelper` passed an incorrect value into the renderer for `renderer_info`. It now passes an instance of `RendererHelper` instead of a dictionary, which is consistent with other usages. See <https://github.com/Pylons/pyramid/issues/#issue/106>
- A bug existed in the `pyramid.authentication.AuthTktCookieHelper` which would break any usage of an `AuthTktAuthenticationPolicy` when one was configured to reissue its tokens (`reissue_time < timeout / max_age`). Symptom: `ValueError: ('Invalid token %r', '')`. See <https://github.com/Pylons/pyramid/issues/#issue/108>.

1.0b1 (2011-01-21)

Features

- The `AuthTktAuthenticationPolicy` now accepts a `tokens` parameter via `pyramid.security.remember`. The value must be a sequence of strings. Tokens are placed into the `auth_tkt` “tokens” field and returned in the `auth_tkt` cookie.
- Add `wild_domain` argument to `AuthTktAuthenticationPolicy`, which defaults to `True`. If it is set to `False`, the feature of the policy which sets a cookie with a wildcard domain will be turned off.
- Add a `MANIFEST.in` file to each paster template. See <https://github.com/Pylons/pyramid/issues#issue/95>

Bug Fixes

- `testing.setUp` now adds a `settings` attribute to the registry (both when it’s passed a registry without any settings and when it creates one).
- The `testing.setUp` function now takes a `settings` argument, which should be a dictionary. Its values will subsequently be available on the returned `config` object as `config.registry.settings`.

Documentation

- Added “What’s New in Pyramid 1.0” chapter to HTML rendering of documentation.
- Merged caseman-master narrative editing branch, many wording fixes and extensions.
- Fix deprecated example showing `chameleon_zpt` API call in testing narrative chapter.
- Added “Adding Methods to the Configurator via `add_directive`” section to Advanced Configuration narrative chapter.
- Add docs for `add_finished_callback`, `add_response_callback`, `route_path`, `route_url`, and `static_url` methods to `pyramid.request.Request` API docs.
- Add (minimal) documentation about using I18N within Mako templates to “Internationalization and Localization” narrative chapter.
- Move content of “Forms” chapter back to “Views” chapter; I can’t think of a better place to put it.
- Slightly improved interface docs for `IAuthorizationPolicy`.
- Minimally explain usage of custom regular expressions in URL dispatch replacement markers within URL Dispatch chapter.

Deprecations

- Using the `pyramid.view.bfg_view` alias for `pyramid.view.view_config` (a backwards compatibility shim) now issues a deprecation warning.

Backwards Incompatibilities

- Using `testing.setUp` now registers an `ISettings` utility as a side effect. Some test code which queries for this utility after `testing.setUp` via `queryAdapter` will expect a return value of `None`. This code will need to be changed.
- When a `pyramid.exceptions.Forbidden` error is raised, its status code now `403 Forbidden`. It was previously `401 Unauthorized`, for backwards compatibility purposes with `repoze.bfg`. This change will cause problems for users of Pyramid with `repoze.who`, which intercepts `401 Unauthorized` by default, but allows `403 Forbidden` to pass through. Those deployments will need to configure `repoze.who` to also react to `403 Forbidden`.
- The default value for the `cookie_on_exception` parameter to `pyramid.session.UnencryptedCookieSessionFactory` is now `True`. This means that when view code causes an exception to be raised, and the session has been mutated, a cookie will be sent back in the response. Previously its default value was `False`.

Paster Templates

- The `pyramid_zodb`, `pyramid_routesalchemy` and `pyramid_alchemy` paster templates now use a default “commit veto” hook when configuring the `repoze.tm2` transaction manager in `development.ini`. This prevents a transaction from being committed when the response status code is within the 400 or 500 ranges. See also <http://docs.repoze.org/tm2/#using-a-commit-veto>.

1.0a10 (2011-01-18)

Bug Fixes

- URL dispatch now properly handles a `.`, `*` or `*` appearing in a regex match when used inside brackets. Resolves issue #90.

Backwards Incompatibilities

- The `add_handler` method of a Configurator has been removed from the Pyramid core. Handlers are now a feature of the `pyramid_handlers` package, which can be downloaded from PyPI. Documentation for the package should be available via http://docs.pylonsproject.org/projects/pyramid_handlers/en/latest/, which describes how to add a configuration statement to your `main` block to reobtain this method. You will also need to add an `install_requires` dependency upon `pyramid_handlers` to your `setup.py` file.
- The `load_zcml` method of a Configurator has been removed from the Pyramid core. Loading ZCML is now a feature of the `pyramid_zcml` package, which can be downloaded from PyPI. Documentation for the package should be available via http://docs.pylonsproject.org/projects/pyramid_zcml/en/latest/, which describes how to add a configuration statement to your `main` block to reobtain this method. You will also need to add an `install_requires` dependency upon `pyramid_zcml` to your `setup.py` file.
- The `pyramid.includes` subpackage has been removed. ZCML files which use include the package `pyramid.includes` (e.g. `<include package="pyramid.includes"/>`) now must include the `pyramid_zcml` package instead (e.g. `<include package="pyramid_zcml"/>`).
- The `pyramid.view.action` decorator has been removed from the Pyramid core. Handlers are now a feature of the `pyramid_handlers` package. It should now be imported from `pyramid_handlers` e.g. `from pyramid_handlers import action`.
- The `handler` ZCML directive has been removed. It is now a feature of the `pyramid_handlers` package.
- The `pylons_minimal`, `pylons_basic` and `pylons_sqla` pasteur templates were removed. Use `pyramid_sqla` (available from PyPI) as a generic replacement for Pylons-esque development.
- The `make_app` function has been removed from the `pyramid.router` module. It continues life within the `pyramid_zcml` package. This leaves the `pyramid.router` module without any API functions.
- The `configure_zcml` setting within the deployment settings (within `**settings` passed to a Pyramid `main` function) has ceased to have any meaning.

Features

- `pyramid.testing.setUp` and `pyramid.testing.tearDown` have been undeprecated. They are now the canonical setup and teardown APIs for test configuration, replacing “direct” creation of a Configurator. This is a change designed to provide a facade that will protect against any future Configurator deprecations.
- Add `charset` attribute to `pyramid.testing.DummyRequest` (unconditionally UTF-8).
- Add `add_directive` method to configurator, which allows framework extenders to add methods to the configurator (ala ZCML directives).
- When `Configurator.include` is passed a *module* as an argument, it defaults to attempting to find and use a callable named `includeme` within that module. This makes it possible to use `config.include('some.module')` rather than `config.include('some.module.somefunc')` as long as the `include` function within `some.module` is named `includeme`.
- The `bfg2pyramid` script now converts ZCML `include` tags that have `repoze.bfg.includes` as a package attribute to the value `pyramid_zcml`. For example, `<include package="repoze.bfg.includes">` will be converted to `<include package="pyramid_zcml">`.

Paster Templates

- All paster templates now use `pyramid.testing.setUp` and `pyramid.testing.tearDown` rather than creating a Configurator “by hand” within their `tests.py` module, as per decision in features above.
- The `starter_zcml` paster template has been moved to the `pyramid_zcml` package.

Documentation

- The `wiki` and `wiki2` tutorials now use `pyramid.testing.setUp` and `pyramid.testing.tearDown` rather than creating a Configurator “by hand”, as per decision in features above.
- The “Testing” narrative chapter now explains `pyramid.testing.setUp` and `pyramid.testing.tearDown` instead of Configurator creation and `Configurator.begin()` and `Configurator.end()`.
- Document the `request.override_renderer` attribute within the narrative “Renderers” chapter in a section named “Overriding A Renderer at Runtime”.
- The “Declarative Configuration” narrative chapter has been removed (it was moved to the `pyramid_zcml` package).
- Most references to ZCML in narrative chapters have been removed or redirected to `pyramid_zcml` locations.

Deprecations

- Deprecation warnings related to import of the following API functions were added: `pyramid.traversal.find_model`, `pyramid.traversal.model_path`, `pyramid.traversal.model_path_tuple`, `pyramid.url.model_url`. The instructions emitted by the deprecation warnings instruct the developer to change these method spellings to their resource equivalents. This is a consequence of the mass concept rename of “model” to “resource” performed in 1.0a7.

1.0a9 (2011-01-08)

Bug Fixes

- The `proutes` command tried too hard to resolve the view for printing, resulting in exceptions when an exceptional root factory was encountered. Instead of trying to resolve the view, if it cannot, it will now just print `<unknown>`.
- The `self` argument was included in new methods of the `ISession` interface signature, causing `pyramid_beaker` tests to fail.
- Readd `pyramid.traversal.model_path_tuple` as an alias for `pyramid.traversal.resource_path_tuple` for backwards compatibility.

Features

- Add a new API `pyramid.url.current_route_url`, which computes a URL based on the “current” route (if any) and its `matchdict` values.
- `config.add_view` now accepts a `decorator` keyword argument, a callable which will decorate the view callable before it is added to the registry.
- If a handler class provides an `__action_decorator__` attribute (usually a classmethod or staticmethod), use that as the decorator for each view registration for that handler.
- The `pyramid.interfaces.IAuthenticationPolicy` interface now specifies an `unauthenticated_userid` method. This method supports an important optimization required by people who are using persistent storages which do not support object caching and whom want to create a “user object” as a request attribute.

- A new API has been added to the `pyramid.security` module named `unauthenticated_userid`. This API function calls the `unauthenticated_userid` method of the effective security policy.
- An `unauthenticated_userid` method has been added to the dummy authentication policy returned by `pyramid.config.Configurator.testing_securitypolicy`. It returns the same thing as that the dummy authentication policy's `authenticated_userid` method.
- The class `pyramid.authentication.AuthTktCookieHelper` is now an API. This class can be used by third-party authentication policy developers to help in the mechanics of authentication cookie-setting.
- New constructor argument to `Configurator`: `default_view_mapper`. Useful to create systems that have alternate view calling conventions. A view mapper allows objects that are meant to be used as view callables to have an arbitrary argument list and an arbitrary result. The object passed as `default_view_mapper` should implement the `pyramid.interfaces.IViewMapperFactory` interface.
- add a `set_view_mapper` API to `Configurator`. Has the same result as passing `default_view_mapper` to the `Configurator` constructor.
- `config.add_view` now accepts a `mapper` keyword argument, which should either be `None`, a string representing a Python dotted name, or an object which is an `IViewMapperFactory`. This feature is not useful for “civilians”, only for extension writers.
- Allow static `renderer` provided during view registration to be overridden at request time via a request attribute named `override_renderer`, which should be the name of a previously registered `renderer`. Useful to provide “omnipresent” RPC using existing rendered views.
- Instances of `pyramid.testing.DummyRequest` now have a `session` object, which is mostly a dictionary, but also implements the other session API methods for flash and CSRF.

Backwards Incompatibilities

- Since the `pyramid.interfaces.IAuthenticationPolicy` interface now specifies that a policy implementation must implement an `unauthenticated_userid` method, all third-party custom authentication policies now must implement this method. It, however, will only be called when the global function named `pyramid.security.unauthenticated_userid` is invoked, so if you're not invoking that, you will not notice any issues.
- `pyramid.interfaces.ISession.get_csrf_token` now mandates that an implementation should return a *new* token if one doesn't already exist in the session (previously it would return `None`). The internal sessioning implementation has been changed.

Documentation

- The (weak) “Converting a CMF Application to Pyramid” tutorial has been removed from the tutorials section. It was moved to the `pyramid_tutorials` Github repository.
- The “Resource Location and View Lookup” chapter has been replaced with a variant of Rob Miller’s “Much Ado About Traversal” (originally published at <http://blog.nonsequitarian.org/2010/much-ado-about-traversal/>).
- Many minor wording tweaks and refactorings (merged Casey Duncan’s docs fork, in which he is working on general editing).
- Added (weak) description of new view mapper feature to Hooks narrative chapter.
- Split views chapter into 2: View Callables and View Configuration.
- Reorder Renderers and Templates chapters after View Callables but before View Configuration.
- Merge Session Objects, Cross-Site Request Forgery, and Flash Messaging chapter into a single Sessions chapter.
- The Wiki and Wiki2 tutorials now have much nicer CSS and graphics.

Internals

- The “view derivation” code is now factored into a set of classes rather than a large number of standalone functions (a side effect of the view mapper refactoring).
- The `pyramid.renderer.RendererHelper` class has grown a `render_view` method, which is used by the default view mapper (a side effect of the view mapper refactoring).
- The object passed as `renderer` to the “view driver” is now an instance of `pyramid.renderers.RendererHelper` rather than a dictionary (a side effect of view mapper refactoring).
- The class used as the “page template” in `pyramid.chameleon_text` was removed, in preference to using a Chameleon-inbuilt version.
- A view callable wrapper registered in the registry now contains an `__original_view__` attribute which references the original view callable (or class).
- The (non-API) method of all internal authentication policy implementations previously named `_get_userid` is now named `unauthenticated_userid`, promoted to an API method. If you were overriding this method, you’ll now need to override it as `unauthenticated_userid` instead.
- Remove (non-API) function of `config.py` named `_map_view`.

1.0a8 (2010-12-27)

Bug Fixes

- The name `registry` was not available in the `paster pshell` environment under IPython.

Features

- If a resource implements a `__resource_url__` method, it will be called as the result of invoking the `pyramid.url.resource_url` function to generate a URL, overriding the default logic. See the new “Generating The URL Of A Resource” section within the Resources narrative chapter.
- Added flash messaging, as described in the “Flash Messaging” narrative documentation chapter.
- Added CSRF token generation, as described in the narrative chapter entitled “Preventing Cross-Site Request Forgery Attacks”.
- Prevent misunderstanding of how the `view` and `view_permission` arguments to `add_route` work by raising an exception during configuration if view-related arguments exist but no `view` argument is passed.
- Add `paster proute` command which displays a summary of the routing table. See the narrative documentation section within the “URL Dispatch” chapter entitled “Displaying All Application Routes”.

Paster Templates

- The `pyramid_zodb` Paster template no longer employs ZCML. Instead, it is based on scanning.

Documentation

- Added “Generating The URL Of A Resource” section to the Resources narrative chapter (includes information about overriding URL generation using `__resource_url__`).
- Added “Generating the Path To a Resource” section to the Resources narrative chapter.
- Added “Finding a Resource by Path” section to the Resources narrative chapter.
- Added “Obtaining the Lineage of a Resource” to the Resources narrative chapter.
- Added “Determining if a Resource is In The Lineage of Another Resource” to Resources narrative chapter.
- Added “Finding the Root Resource” to Resources narrative chapter.
- Added “Finding a Resource With a Class or Interface in Lineage” to Resources narrative chapter.
- Added a “Flash Messaging” narrative documentation chapter.
- Added a narrative chapter entitled “Preventing Cross-Site Request Forgery Attacks”.
- Changed the “ZODB + Traversal Wiki Tutorial” based on changes to `pyramid_zodb` Paster template.
- Added “Advanced Configuration” narrative chapter which documents how to deal with configuration conflicts, two-phase configuration, `include` and `commit`.
- Fix API documentation rendering for `pyramid.view.static`
- Add “Pyramid Provides More Than One Way to Do It” to Design Defense documentation.
- Changed “Static Assets” narrative chapter: clarify that `name` represents a prefix unless it’s a URL, added an example of a root-relative static view fallback for URL dispatch, added an example of creating a simple view that returns the body of a file.
- Move ZCML usage in Hooks chapter to Declarative Configuration chapter.
- Merge “Static Assets” chapter into the “Assets” chapter.
- Added narrative documentation section within the “URL Dispatch” chapter entitled “Displaying All Application Routes” (for `paster proutes` command).

1.0a7 (2010-12-20)

Terminology Changes

- The Pyramid concept previously known as “model” is now known as “resource”. As a result:
 - The following API changes have been made:


```
pyramid.url.model_url ->
    pyramid.url.resource_url

pyramid.traversal.find_model ->
    pyramid.url.find_resource

pyramid.traversal.model_path ->
    pyramid.traversal.resource_path

pyramid.traversal.model_path_tuple ->
    pyramid.traversal.resource_path_tuple

pyramid.traversal.ModelGraphTraverser ->
    pyramid.traversal.ResourceTreeTraverser

pyramid.config.Configurator.testing_models ->
    pyramid.config.Configurator.testing_resources

pyramid.testing.registerModels ->
    pyramid.testing.registerResources

pyramid.testing.DummyModel ->
    pyramid.testing.DummyResource
```

- All documentation which previously referred to “model” now refers to “resource”.
- The `starter` and `starter_zcml` pasteur templates now have a `resources.py` module instead of a `models.py` module.
- Positional argument names of various APIs have been changed from `model` to `resource`.

Backwards compatibility shims have been left in place in all cases. They will continue to work “forever”.

- The Pyramid concept previously known as “resource” is now known as “asset”. As a result:
 - The (non-API) module previously known as `pyramid.resource` is now known as `pyramid.asset`.
 - All docs that previously referred to “resource specification” now refer to “asset specification”.
 - The following API changes were made:

```
pyramid.config.Configurator.absolute_resource_spec ->
    pyramid.config.Configurator.absolute_asset_spec

pyramid.config.Configurator.override_resource ->
    pyramid.config.Configurator.override_asset
```

- The ZCML directive previously known as `resource` is now known as `asset`.
- The setting previously known as `BFG_RELOAD_RESOURCES` (envvar) or `reload_resources` (config file) is now known, respectively, as `PYRAMID_RELOAD_ASSETS` and `reload_assets`.

Backwards compatibility shims have been left in place in all cases. They will continue to work “forever”.

Bug Fixes

- Make it possible to successfully run all tests via `nosetests` command directly (rather than indirectly via `python setup.py nosetests`).
- When a configuration conflict is encountered during scanning, the conflict exception now shows the decorator information that caused the conflict.

Features

- Added `debug_routematch` configuration setting that logs matched routes (including the `matchdict` and `predicates`).
- The name `registry` is now available in a `pshell` environment by default. It is the application registry object.

Environment

- All environment variables which used to be prefixed with `BFG_` are now prefixed with `PYRAMID_` (e.g. `BFG_DEBUG_NOTFOUND` is now `PYRAMID_DEBUG_NOTFOUND`)

Documentation

- Added “Debugging Route Matching” section to the `urldispatch` narrative documentation chapter.
- Added reference to `PYRAMID_DEBUG_ROUTEMATCH` envvar and `debug_routematch` config file setting to the Environment narrative docs chapter.
- Changed “Project” chapter slightly to expand on use of `paster pshell`.
- Direct Jython users to Mako rather than Jinja2 in “Install” narrative chapter.
- Many changes to support terminological renaming of “model” to “resource” and “resource” to “asset”.
- Added an example of `WebTest` functional testing to the testing narrative chapter.
- Rearranged chapter ordering by popular demand (URL dispatch first, then traversal). Put hybrid chapter after views chapter.
- Split off “Renderers” as its own chapter from “Views” chapter in narrative documentation.

Paster Templates

- Added `debug_routematch = false` to all paster templates.

Dependencies

- Depend on `Venusian >= 0.5` (for scanning conflict exception decoration).

1.0a6 (2010-12-15)

Bug Fixes

- 1.0a5 introduced a bug when `pyramid.config.Configurator.scan` was used without a package argument (e.g. `config.scan()` as opposed to `config.scan('packagename')`). The symptoms were: lots of deprecation warnings printed to the console about imports of deprecated Pyramid functions and classes and non-detection of view callables decorated with `view_config` decorators. This has been fixed.
- Tests now pass on Windows (no bugs found, but a few tests in the test suite assumed UNIX path segments in filenames).

Documentation

- If you followed it to-the-letter, the ZODB+Traversal Wiki tutorial would instruct you to run a test which would fail because the view callable generated by the `pyramid_zodb` tutorial used a one-arg view callable, but the test in the sample code used a two-arg call.
- Updated ZODB+Traversal tutorial `setup.py` of all steps to match what's generated by `pyramid_zodb`.
- Fix reference to `repoze.bfg.traversalwrapper` in “Models” chapter (point at `pyramid_traversalwrapper` instead).

1.0a5 (2010-12-14)

Features

- Add a handler ZCML directive. This directive does the same thing as `pyramid.configuration.add_handler`.
- A new module named `pyramid.config` was added. It subsumes the duties of the older `pyramid.configuration` module.
- The new `pyramid.config.Configurator` class has API methods that the older `pyramid.configuration.Configurator` class did not: `with_context` (a classmethod), `include`, `action`, and `commit`. These methods exist for imperative application extensibility purposes.
- The `pyramid.testing.setUp` function now accepts an `autocommit` keyword argument, which defaults to `True`. If it is passed `False`, the `Config` object returned by `setUp` will be a non-autocommitting `Config` object.
- Add logging configuration to all pasteur templates.
- `pyramid_alchemy`, `pyramid_routesalchemy`, and `pylons_sqla` pasteur templates now use idiomatic `SQLAlchemy` configuration in their respective `.ini` files and Python code.
- `pyramid.testing.DummyRequest` now has a class variable, `query_string`, which defaults to the empty string.
- Add support for json on GAE by catching `NotImplementedError` and importing `simplejson` from `django.utils`.
- The Mako renderer now accepts a resource specification for `mako.module_directory`.
- New boolean Mako settings variable `mako.strict_undefined`. See Mako Context Variables for its meaning.

Dependencies

- Depend on Mako 0.3.6+ (we now require the `strict_undefined` feature).

Bug Fixes

- When creating a Configurator from within a `paster pshell` session, you were required to pass a `package` argument although `package` is not actually required. If you didn't pass `package`, you would receive an error something like `KeyError: '__name__'` emanating from the `pyramid.path.caller_module` function. This has now been fixed.
- The `pyramid_routesalchemy` `paster` template's unit tests failed (`AssertionError: 'SomeProject' != 'someproject'`). This is fixed.
- Make default renderer work (renderer factory registered with no name, which is active for every view unless the view names a specific renderer).
- The Mako renderer did not properly turn the `mako.imports`, `mako.default_filters`, and `mako.imports` settings into lists.
- The Mako renderer did not properly convert the `mako.error_handler` setting from a dotted name to a callable.

Documentation

- Merged many wording, readability, and correctness changes to narrative documentation chapters from <https://github.com/caseman/pyramid> (up to and including “Models” narrative chapter).
- “Sample Applications” section of docs changed to note existence of Cluegun, Shootout and Virginia sample applications, ported from their `repoze.bfg` origin packages.
- `SQLAlchemy+URLDispatch` tutorial updated to integrate changes to `pyramid_routesalchemy` template.
- Add `pyramid.interfaces.ITemplateRenderer` interface to Interfaces API chapter (has `implementation()` method, required to be used when getting at Chameleon macros).
- Add a “Modifying Package Structure” section to the project narrative documentation chapter (explain turning a module into a package).
- Documentation was added for the new `handler ZCML` directive in the ZCML section.

Deprecations

- `pyramid.configuration.Configurator` is now deprecated. Use `pyramid.config.Configurator`, passing its constructor `autocommit=True` instead. The `pyramid.configuration.Configurator` alias will live for a long time, as every application uses it, but its import now issues a deprecation warning. The `pyramid.config.Configurator` class has the same API as `pyramid.configuration.Configurator` class, which it means to replace, except by default it is a *non-autocommitting* configurator. The now-deprecated `pyramid.configuration.Configurator` will autocommit every time a configuration method is called.

The `pyramid.configuration` module remains, but it is deprecated. Use `pyramid.config` instead.

1.0a4 (2010-11-21)

Features

- URL Dispatch now allows for replacement markers to be located anywhere in the pattern, instead of immediately following a `/`.
- URL Dispatch now uses the form `{marker}` to denote a replace marker in the route pattern instead of `:marker`. The old colon-style marker syntax is still accepted for backwards compatibility. The new format allows a regular expression for that marker location to be used instead of the default `[^/]+`, for example `{marker:\d+}` is now valid to require the marker to be digits.
- Add a `pyramid.url.route_path` API, allowing folks to generate relative URLs. Calling `route_path` is the same as calling `pyramid.url.route_url` with the argument `_app_url` equal to the empty string.
- Add a `pyramid.request.Request.route_path` API. This is a convenience method of the request which calls `pyramid.url.route_url`.
- Make test suite pass on Jython (requires PasteScript trunk, presumably to be 1.7.4).
- Make test suite pass on PyPy (Chameleon doesn't work).
- Surrounding application configuration with `config.begin()` and `config.end()` is no longer necessary. All paster templates have been changed to no longer call these functions.
- Fix configurator to not convert `ImportError` to `ConfigurationError` if the import that failed was unrelated to the import requested via a dotted name when resolving dotted names (such as view dotted names).

Documentation

- SQLAlchemy+URLDispatch and ZODB+Traversal tutorials have been updated to not call `config.begin()` or `config.end()`.

Bug Fixes

- Add deprecation warnings to import of `pyramid.chameleon_text` and `pyramid.chameleon_zpt` of `get_renderer`, `get_template`, `render_template`, and `render_template_to_response`.
- Add deprecation warning for import of `pyramid.zcml.zcml_configure` and `pyramid.zcml.file_configure`.
- The `pyramid_alchemy` paster template had a typo, preventing an import from working.
- Fix apparent failures when calling `pyramid.traversal.find_model(root, path)` or `pyramid.traversal.traverse(path)` when `path` is (erroneously) a Unicode object. The user is meant to pass these APIs a string object, never a Unicode object. In practice, however, users indeed pass Unicode. Because the string that is passed must be ASCII encodeable, now, if they pass a Unicode object, its data is eagerly converted to an ASCII string rather than being passed along to downstream code as a convenience to the user and to prevent puzzling second-order failures from cropping up (all failures will occur within `pyramid.traversal.traverse` rather than later down the line as the result of calling e.g. `traversal_path`).

Backwards Incompatibilities

- The `pyramid.testing.zcml_configure` API has been removed. It had been advertised as removed since repoze.bfg 1.2a1, but hadn't actually been.

Deprecations

- The `pyramid.settings.get_settings` API is now deprecated. Use `pyramid.threadlocals.get_current_registry().settings` instead or use the `settings` attribute of the registry available from the request (`request.registry.settings`).

Documentation

- Removed `zodbsessions` tutorial chapter. It's still useful, but we now have a `SessionFactory` abstraction which competes with it, and maintaining documentation on both ways to do it is a distraction.

Internal

- Replace `Twill` with `WebTest` in internal integration tests (avoid deprecation warnings generated by `Twill`).

1.0a3 (2010-11-16)

Features

- Added Mako `TemplateLookup` settings for `mako.error_handler`, `mako.default_filters`, and `mako.imports`.
- Normalized all paster templates: each now uses the name `main` to represent the function that returns a WSGI application, each now uses `WebError`, each now has roughly the same shape of `development.ini` style.
- Added class vars `matchdict` and `matched_route` to `pyramid.request.Request`. Each is set to `None`.
- New API method: `pyramid.settings.asbool`.
- New API methods for `pyramid.request.Request`: `model_url`, `route_url`, and `static_url`. These are simple passthroughs for their respective functions in `pyramid.url`.
- The `settings` object which used to be available only when `request.settings.get_settings` was called is now available as `registry.settings` (e.g. `request.registry.settings` in view code).

Bug Fixes

- The pylons_* paster templates erroneously used the {squiggly} routing syntax as the pattern supplied to `add_route`. This style of routing is not supported. They were replaced with `:colon` style route patterns.
- The pylons_* paster template used the same string (`your_app_secret_string`) for the `session.secret` setting in the generated `development.ini`. This was a security risk if left unchanged in a project that used one of the templates to produce production applications. It now uses a randomly generated string.

Documentation

- ZODB+traversal wiki (wiki) tutorial updated due to changes to `pyramid_zodb` paster template.
- SQLAlchemy+urldispatch wiki (wiki2) tutorial updated due to changes to `pyramid_routesalchemy` paster template.
- Documented the `matchdict` and `matched_route` attributes of the request object in the Request API documentation.

Deprecations

- Obtaining the settings object via `registry.{get|query}Utility(ISettings)` is now deprecated. Instead, obtain the settings object via the `registry.settings` attribute. A backwards compatibility shim was added to the registry object to register the settings object as an `ISettings` utility when `setattr(registry, 'settings', foo)` is called, but it will be removed in a later release.
- Obtaining the settings object via `pyramid.settings.get_settings` is now deprecated. Obtain it as the `settings` attribute of the registry now (obtain the registry via `pyramid.threadlocal.get_registry` or as `request.registry`).

Behavior Differences

- Internal: ZCML directives no longer call `get_current_registry()` if there's a `registry` attribute on the ZCML context (kill off use of threadlocals).
- Internal: Chameleon template renderers now accept two arguments: `path` and `lookup`. `Lookup` will be an instance of a lookup class which supplies (late-bound) arguments for `debug`, `reload`, and `translate`. Any third-party renderers which use (the non-API) function `pyramid.renderers.template_renderer_factory` will need to adjust their implementations to obey the new callback argument list. This change was to kill off inappropriate use of threadlocals.

1.0a2 (2010-11-09)

Documentation

- All references to events by interface (e.g. `pyramid.interfaces.INewRequest`) have been changed to reference their concrete classes (e.g. `pyramid.events.NewRequest`) in documentation about making subscriptions.
- All references to Pyramid-the-application were changed from *mod-pyramid* to *app-Pyramid*. A custom role setting was added to `docs/conf.py` to allow for this. (internal)

1.0a1 (2010-11-05)

Features (delta from BFG 1.3)

- Mako templating renderer supports resource specification format for template lookups and within Mako templates. Absolute filenames must be used in Pyramid to avoid this lookup process.
- Add `pyramid.httpexceptions` module, which is a facade for the `webob.exc` module.
- Direct built-in support for the Mako templating language.
- A new configurator method exists: `add_handler`. This method adds a Pylons-style “view handler” (such a thing used to be called a “controller” in Pylons 1.0).
- New argument to configurator: `session_factory`.
- New method on configurator: `set_session_factory`
- Using `request.session` now returns a (dictionary-like) session object if a session factory has been configured.
- The request now has a new attribute: `tmpl_context` for benefit of Pylons users.
- The decorator previously known as `pyramid.view.bfg_view` is now known most formally as `pyramid.view.view_config` in docs and paster templates. An import of `pyramid.view.bfg_view`, however, will continue to work “forever”.
- New API methods in `pyramid.session`: `signed_serialize` and `signed_deserialize`.

- New interface: `pyramid.interfaces.IRendererInfo`. An object of this type is passed to renderer factory constructors (see “Backwards Incompatibilities”).
- New event type: `pyramid.interfaces.IBeforeRender`. An object of this type is sent as an event before a renderer is invoked (but after the application-level renderer globals factory added via `pyramid.configurator.configuration.set_renderer_globals_factory`, if any, has injected its own keys). Applications may now subscribe to the `IBeforeRender` event type in order to introspect the and modify the set of renderer globals before they are passed to a renderer. The event object itself has a dictionary-like interface that can be used for this purpose. For example:

```
from repoze.events import subscriber
from pyramid.interfaces import IRendererGlobalsEvent

@subscriber(IRendererGlobalsEvent)
def add_global(event):
    event['mykey'] = 'foo'
```

If a subscriber attempts to add a key that already exist in the renderer globals dictionary, a `KeyError` is raised. This limitation is due to the fact that subscribers cannot be ordered relative to each other. The set of keys added to the renderer globals dictionary by all subscribers and app-level globals factories must be unique.

- New class: `pyramid.response.Response`. This is a pure facade for `webob.Response` (old code need not change to use this facade, it’s existence is mostly for vanity and documentation-generation purposes).
- All preexisting paster templates (except `zodb`) now use “imperative” configuration (`starter`, `routesalchemy`, `alchemy`).
- A new paster template named `pyramid_starter_zcml` exists, which uses declarative configuration.

Documentation (delta from BFG 1.3)

- Added a `pyramid.httpexceptions` API documentation chapter.
- Added a `pyramid.session` API documentation chapter.
- Added a `Session Objects` narrative documentation chapter.

- Added an API chapter for the `pyramid.personality` module.
- Added an API chapter for the `pyramid.response` module.
- All documentation which previously referred to `webob.Response` now uses `pyramid.response.Response` instead.
- The documentation has been overhauled to use imperative configuration, moving declarative configuration (ZCML) explanations to a separate narrative chapter `declarative.rst`.
- The ZODB Wiki tutorial was updated to take into account changes to the `pyramid_zodb` paster template.
- The SQL Wiki tutorial was updated to take into account changes to the `pyramid_routesalchemy` paster template.

Backwards Incompatibilities (with BFG 1.3)

- There is no longer an `IDebugLogger` registered as a named utility with the name `repoze.bfg.debug`.
- The logger which used to have the name of `repoze.bfg.debug` now has the name `pyramid.debug`.
- The deprecated API `pyramid.testing.registerViewPermission` has been removed.
- The deprecated API named `pyramid.testing.registerRoutesMapper` has been removed.
- The deprecated API named `pyramid.request.get_request` was removed.
- The deprecated API named `pyramid.security.Unauthorized` was removed.
- The deprecated API named `pyramid.view.view_execution_permitted` was removed.
- The deprecated API named `pyramid.view.NotFound` was removed.
- The `bfgshell` paster command is now named `pshell`.
- The Venesian “category” for all built-in Venesian decorators (e.g. `subscriber` and `view_config/bfg_view`) is now `pyramid` instead of `bfg`.

- `pyramid.renderers.rendered_response` function removed; use `render_pyramid.renderers.render_to_response` instead.
- Renderer factories now accept a *renderer info object* rather than an absolute resource specification or an absolute path. The object has the following attributes: `name` (the `renderer=` value), `package` (the ‘current package’ when the renderer configuration statement was found), `type`: the renderer type, `registry`: the current registry, and `settings`: the deployment settings dictionary.

Third-party `repoze.bfg` renderer implementations that must be ported to Pyramid will need to account for this.

This change was made primarily to support more flexible Mako template rendering.

- The presence of the key `repoze.bfg.message` in the WSGI environment when an exception occurs is now deprecated. Instead, code which relies on this environ value should use the `exception` attribute of the request (e.g. `request.exception[0]`) to retrieve the message.
- The values `bfg_localizer` and `bfg_locale_name` kept on the request during internationalization for caching purposes were never APIs. These however have changed to `localizer` and `locale_name`, respectively.
- The default `cookie_name` value of the `authkttauthenticationpolicy` ZCML now defaults to `auth_tkt` (it used to default to `repoze.bfg.auth_tkt`).
- The default `cookie_name` value of the `pyramid.authentication.AuthTktAuthenticationPolicy` constructor now defaults to `auth_tkt` (it used to default to `repoze.bfg.auth_tkt`).
- The `request_type` argument to the `view` ZCML directive, the `pyramid.configuration.Configurator.add_view` method, or the `pyramid.view.view_config` decorator (nee `bfg_view`) is no longer permitted to be one of the strings `GET`, `HEAD`, `PUT`, `POST` or `DELETE`, and now must always be an interface. Accepting the method-strings as `request_type` was a backwards compatibility strategy servicing `repoze.bfg` 1.0 applications. Use the `request_method` parameter instead to specify that a view a string request-method predicate.

repoze.bfg Change History (previous name for Pyramid)

1.3b1 (2010-10-25)

Features

- The `paster` template named `bfg_routesalchemy` has been updated to use SQLAlchemy declarative syntax. Thanks to Ergo^.

Bug Fixes

- When a renderer factory could not be found, a misleading error message was raised if the renderer name was not a string.

Documentation

- The “bfgwiki2” (SQLAlchemy + url dispatch) tutorial has been updated slightly. In particular, the source packages no longer attempt to use a private index, and the recommended Python version is now 2.6. It was also updated to take into account the changes to the `bfg_routesalchemy` template used to set up an environment.
- The “bfgwiki” (ZODB + traversal) tutorial has been updated slightly. In particular, the source packages no longer attempt to use a private index, and the recommended Python version is now 2.6.

1.3a15 (2010-09-30)

Features

- The `repoze.bfg.traversal.traversal_path` API now eagerly attempts to encode a Unicode path into ASCII before attempting to split it and decode its segments. This is for convenience, effectively to allow a (stored-as-Unicode-in-a-database, or retrieved-as-Unicode-from-a-request-parameter) Unicode path to be passed to `find_model`, which eventually internally uses the `traversal_path` function under the hood. In version 1.2 and prior, if the path was Unicode, that Unicode was split on slashes and each resulting segment value was Unicode. An inappropriate call to the `decode()` method of a resulting Unicode path segment could cause a `UnicodeDecodeError` to occur even if the Unicode representation of the path contained no ‘high order’ characters (it effectively did a “double decode”). By converting the Unicode path argument to ASCII before we attempt to decode and split, genuine errors will occur in a more obvious place while also allowing us to handle (for convenience) the case that it’s a Unicode representation formed entirely from ASCII-compatible characters.

1.3a14 (2010-09-14)

Bug Fixes

- If an exception view was registered through the legacy `set_notfound_view` or `set_forbidden_view` APIs, the context sent to the view was incorrect (could be `None` inappropriately).

Features

- Compatibility with WebOb 1.0.

Requirements

- Now requires WebOb \geq 1.0.

Backwards Incompatibilities

- Due to changes introduced WebOb 1.0, the `repoze.bfg.request.make_request_ascii` event subscriber no longer works, so it has been removed. This subscriber was meant to be used in a deployment so that code written before BFG 0.7.0 could run unchanged. At this point, such code will need to be rewritten to expect Unicode from `request.GET`, `request.POST` and `request.params` or it will need to be changed to use `request.str_POST`, `request.str_GET` and/or `request.str_params` instead of the non-str versions of same, as the non-str versions of the same APIs always now perform decoding to Unicode.

Errata

- A prior changelog entry asserted that the `INewResponse` event was not sent to listeners if the response was not “valid” (if a view or renderer returned a response object that did not have a `status/headers/app_iter`). This is not true in this release, nor was it true in 1.3a13.

1.3a13 (2010-09-14)

Bug Fixes

- The `traverse` route predicate could not successfully generate a traversal path.

Features

- In support of making it easier to configure applications which are “secure by default”, a default permission feature was added. If supplied, the default permission is used as the permission string to all view registrations which don’t otherwise name a permission. These APIs are in support of that:
 - A new constructor argument was added to the Configurator: `default_permission`.
 - A new method was added to the Configurator: `set_default_permission`.
 - A new ZCML directive was added: `default_permission`.
- Add a new request API: `request.add_finished_callback`. Finished callbacks are called by the router unconditionally near the very end of request processing. See the “Using Finished Callbacks” section of the “Hooks” narrative chapter of the documentation for more information.
- A `request.matched_route` attribute is now added to the request when a route has matched. Its value is the “route” object that matched (see the `IRoute` interface within `repoze.bfg.interfaces` API documentation for the API of a route object).
- The `exception` attribute of the request is now set slightly earlier and in a slightly different set of scenarios, for benefit of “finished callbacks” and “response callbacks”. In previous versions, the `exception` attribute of the request was not set at all if an exception view was not found. In this version, the `request.exception` attribute is set immediately when an exception is caught by the router, even if an exception view could not be found.
- The `add_route` method of a Configurator now accepts a `pregenerator` argument. The pregenerator for the resulting route is called by `route_url` in order to adjust the set of arguments passed to it by the user for special purposes, such as Pylons ‘subdomain’ support. It will influence the URL returned by `route_url`. See the `repoze.bfg.interfaces.IRoutePregenerator` interface for more information.

Backwards Incompatibilities

- The router no longer sets the value `wsgiorg.routing_args` into the environ when a route matches. The value used to be something like `((), matchdict)`. This functionality was only ever obliquely referred to in change logs; it was never documented as an API.
- The `exception` attribute of the request now defaults to `None`. In prior versions, the `request.exception` attribute did not exist if an exception was not raised by user code during request processing; it only began existence once an exception view was found.

Deprecations

- The `repoze.bfg.interfaces.IWSGIApplicationCreatedEvent` event interface was renamed to `repoze.bfg.interfaces.IApplicationCreated`. Likewise, the `repoze.bfg.events.WSGIApplicationCreatedEvent` class was renamed to `repoze.bfg.events.ApplicationCreated`. The older aliases will continue to work indefinitely.
- The `repoze.bfg.interfaces.IAfterTraversal` event interface was renamed to `repoze.bfg.interfaces.IContextFound`. Likewise, the `repoze.bfg.events.AfterTraversal` class was renamed to `repoze.bfg.events.ContextFound`. The older aliases will continue to work indefinitely.
- References to the WSGI environment values `bfg.routes.matchdict` and `bfg.routes.route` were removed from documentation. These will stick around internally for several more releases, but it is `request.matchdict` and `request.matched_route` are now the “official” way to obtain the matchdict and the route object which resulted in the match.

Documentation

- Added documentation for the `default_permission` ZCML directive.
- Added documentation for the `default_permission` constructor value and the `set_default_permission` method in the Configurator API documentation.
- Added a new section to the “security” chapter named “Setting a Default Permission”.
- Document `renderer_globals_factory` and `request_factory` arguments to Configurator constructor.
- Added two sections to the “Hooks” chapter of the documentation: “Using Response Callbacks” and “Using Finished Callbacks”.
- Added documentation of the `request.exception` attribute to the `repoze.bfg.request.Request` API documentation.
- Added glossary entries for “response callback” and “finished callback”.
- The “Request Processing” narrative chapter has been updated to note finished and response callback steps.
- New interface in interfaces API documentation: `IRoutePregenerator`.
- Added a “The Matched Route” section to the URL Dispatch narrative docs chapter, detailing the `matched_route` attribute.

1.3a12 (2010-09-08)

Bug Fixes

- Fix a bug in `repoze.bfg.url.static_url` URL generation: if two resource specifications were used to create two separate static views, but they shared a common prefix, it was possible that `static_url` would generate an incorrect URL.
- Fix another bug in `repoze.bfg.static_url` URL generation: too many slashes in generated URL.
- Prevent a race condition which could result in a `RuntimeError` when rendering a Chameleon template that has not already been rendered once. This would usually occur directly after a restart, when more than one person or thread is trying to execute the same view at the same time: <https://bugs.launchpad.net/karl3/+bug/621364>

Features

- The argument to `repoze.bfg.configuration.Configurator.add_route` which was previously called `path` is now called `pattern` for better explicability. For backwards compatibility purposes, passing a keyword argument named `path` to `add_route` will still work indefinitely.
- The `path` attribute to the ZCML `route` directive is now named `pattern` for better explicability. The older `path` attribute will continue to work indefinitely.

Documentation

- All narrative, API, and tutorial docs which referred to a route pattern as a `path` have now been updated to refer to them as a `pattern`.
- The `repoze.bfg.interfaces` API documentation page is now rendered via `repoze.sphinx.autointerface`.
- The URL Dispatch narrative chapter now refers to the `interfaces` chapter to explain the API of an `IRoute` object.

Paster Templates

- The routesalchemy template has been updated to use `pattern` in its route declarations rather than `path`.

Dependencies

- `tests_require` now includes `repoze.sphinx.autointerface` as a dependency.

Internal

- Add an API to the Configurator named `get_routes_mapper`. This returns an object implementing the `IRoutesMapper` interface.
- The `repoze.bfg.urldispatch.RoutesMapper` object now has a `get_route` method which returns a single `Route` object or `None`.
- A new interface `repoze.bfg.interfaces.IRoute` was added. The `repoze.bfg.urldispatch.Route` object implements this interface.
- The canonical attribute for accessing the routing pattern from a route object is now `pattern` rather than `path`.
- Use `hash()` rather than `id()` when computing the “phash” of a custom route/view predicate in order to allow the custom predicate some control over which predicates are “equal”.
- Use `response.headerlist.append` instead of `response.headers.add` in `repoze.bfg.request.add_global_response_headers` in case the response is not a `WebOb` response.
- The `repoze.bfg.urldispatch.Route` constructor (not an API) now accepts a different ordering of arguments. Previously it was `(pattern, name, factory=None, predicates=())`. It is now `(name, pattern, factory=None, predicates=())`. This is in support of consistency with `configurator.add_route`.
- The `repoze.bfg.urldispatch.RoutesMapper.connect` method (not an API) now accepts a different ordering of arguments. Previously it was `(pattern, name, factory=None, predicates=())`. It is now `(name, pattern, factory=None, predicates=())`. This is in support of consistency with `configurator.add_route`.

1.3a11 (2010-09-05)

Bug Fixes

- Process the response callbacks and the `NewResponse` event earlier, to enable mutations to the response to take effect.

1.3a10 (2010-09-05)

Features

- A new `repoze.bfg.request.Request.add_response_callback` API has been added. This method is documented in the new `repoze.bfg.request` API chapter. It can be used to influence response values before a concrete response object has been created.
- The `repoze.bfg.interfaces.INewResponse` interface now includes a `request` attribute; as a result, a handler for `INewResponse` now has access to the request which caused the response.
- Each of the follow methods of the `Configurator` now allow the below-named arguments to be passed as “dotted name strings” (e.g. “foo.bar.baz”) rather than as actual implementation objects that must be imported:

setup_registry `root_factory`, `authentication_policy`, `authorization_policy`, `debug_logger`, `locale_negotiator`, `request_factory`, `renderer_globals_factory`

add_subscriber `subscriber`, `iface`

derive_view `view`

add_view `view`, `for_`, `context`, `request_type`, `containment`

add_route() `view`, `view_for`, `factory`, `for_`, `view_context`

scan `package`

add_renderer `factory`

set_forbidden_view `view`

set_notfound_view `view`

set_request_factory `factory`

set_renderer_globals_factory() `factory`

set_locale_negotiator `negotiator`

testing_add_subscriber `event_iface`

Bug Fixes

- The route pattern registered internally for a local “static view” (either via the `static` ZCML directive or via the `add_static_view` method of the configurator) was incorrect. It was registered for e.g. `static*traverse`, while it should have been registered for `static/*traverse`. Symptom: two static views could not reliably be added to a system when they both shared the same path prefix (e.g. `/static` and `/static2`).

Backwards Incompatibilities

- The `INewResponse` event is now not sent to listeners if the response returned by view code (or a renderer) is not a “real” response (e.g. if it does not have `.status`, `.headerlist` and `.app_iter` attributes).

Documentation

- Add an API chapter for the `repoze.bfg.request` module, which includes documentation for the `repoze.bfg.request.Request` class (the “request object”).
- Modify the “Request and Response” narrative chapter to reference the new `repoze.bfg.request` API chapter. Some content was moved from this chapter into the API documentation itself.
- Various changes to denote that Python dotted names are now allowed as input to Configurator methods.

Internal

- The (internal) feature which made it possible to attach a `global_response_headers` attribute to the request (which was assumed to contain a sequence of header key/value pairs which would later be added to the response by the router), has been removed. The functionality of `repoze.bfg.request.Request.add_response_callback` takes its place.
- The `repoze.bfg.events.NewResponse` class’s construct has changed: it now must be created with `(request, response)` rather than simply `(response)`.

1.3a9 (2010-08-22)

Features

- The Configurator now accepts a dotted name *string* to a package as a package constructor argument. The package argument was previously required to be a package *object* (not a dotted name string).
- The `repoze.bfg.configuration.Configurator.with_package` method was added. This method returns a new Configurator using the same application registry as the configurator object it is called upon. The new configurator is created afresh with its package constructor argument set to the value passed to `with_package`. This feature will make it easier for future BFG versions to allow dotted names as arguments in places where currently only object references are allowed (the work to allow dotted names instead of object references everywhere has not yet been done, however).
- The new `repoze.bfg.configuration.Configurator.maybe_dotted` method resolves a Python dotted name string supplied as its `dotted` argument to a global Python object. If the value cannot be resolved, a `repoze.bfg.configuration.ConfigurationError` is raised. If the value supplied as `dotted` is not a string, the value is returned unconditionally without any resolution attempted.
- The new `repoze.bfg.configuration.Configurator.absolute_resource_spec` method resolves a potentially relative “resource specification” string into an absolute version. If the value supplied as `relative_spec` is not a string, the value is returned unconditionally without any resolution attempted.

Backwards Incompatibilities

- The functions in `repoze.bfg.renderers` named `render` and `render_to_response` introduced in 1.3a6 previously took a set of `**values` arguments for the values to be passed to the renderer. This was wrong, as renderers don’t need to accept only dictionaries (they can accept any type of object). Now, the value sent to the renderer must be supplied as a positional argument named `value`. The `request` argument is still a keyword argument, however.
- The functions in `repoze.bfg.renderers` named `render` and `render_to_response` now accept an additional keyword argument named `package`.
- The `get_renderer` API in `repoze.bfg.renderers` now accepts a `package` argument.

Documentation

- The ZCML `include` directive docs were incorrect: they specified `filename` rather than (the correct) `file` as an allowable attribute.

Internal

- The `repoze.bfg.resource.resolve_resource_spec` function can now accept a package object as its `pname` argument instead of just a package name.
- The `_renderer_factory_from_name` and `_renderer_from_name` methods of the `Configurator` were removed. These were never APIs.
- The `_render`, `_render_to_response` and `_make_response` functions with `repoze.bfg.render` (added in 1.3a6) have been removed.
- A new helper class `repoze.bfg.renderers.RendererHelper` was added.
- The `_map_view` function of `repoze.bfg.configuration` now takes only a `renderer_name` argument instead of both a `renderer` and `renderer`_name` argument. It also takes a ``package` argument now.
- Use `imp.get_suffixes` indirection in `repoze.bfg.path.package_name` instead of hardcoded `.py` `.pyc` and `.pyo` to use for comparison when attempting to decide if a directory is a package.
- Make tests runnable again under Jython (although they do not all pass currently).
- The `reify` decorator now maintains the docstring of the function it wraps.

1.3a8 (2010-08-08)

Features

- New public interface: `repoze.bfg.exceptions.IExceptionResponse`. This interface is provided by all internal exception classes (such as `repoze.bfg.exceptions.NotFound` and `repoze.bfg.exceptions.Forbidden`), instances of which are both exception objects and can behave as WSGI response objects. This interface is made public so that exception classes which are also valid WSGI response factories can be configured to implement them or exception instances which are also or response instances can be configured to provide them.

- New API class: `repoze.bfg.view.AppendSlashNotFoundViewFactory`.

There can only be one Not Found view in any `repoze.bfg` application. Even if you use `repoze.bfg.view.append_slash_notfound_view` as the Not Found view, `repoze.bfg` still must generate a 404 Not Found response when it cannot redirect to a slash-appended URL; this not found response will be visible to site users.

If you don't care what this 404 response looks like, and you only need redirections to slash-appended route URLs, you may use the `repoze.bfg.view.append_slash_notfound_view` object as the Not Found view. However, if you wish to use a *custom* notfound view callable when a URL cannot be redirected to a slash-appended URL, you may wish to use an instance of the `repoze.bfg.view.AppendSlashNotFoundViewFactory` class as the Not Found view, supplying the notfound view callable as the first argument to its constructor. For instance:

```
from repoze.bfg.exceptions import NotFound
from repoze.bfg.view import AppendSlashNotFoundViewFactory

def notfound_view(context, request):
    return HTTPNotFound('It aint there, stop trying!')

custom_append_slash = AppendSlashNotFoundViewFactory(notfound_view)
config.add_view(custom_append_slash, context=NotFound)
```

The `notfound_view` supplied must adhere to the two-argument view callable calling convention of `(context, request)` (`context` will be the exception object).

Documentation

- Expanded the “Cleaning Up After a Request” section of the URL Dispatch narrative chapter.
- Expanded the “Redirecting to Slash-Appended Routes” section of the URL Dispatch narrative chapter.

Internal

- Previously, two default view functions were registered at Configurator setup (one for `repoze.bfg.exceptions.NotFound` named `default_notfound_view` and one for `repoze.bfg.exceptions.Forbidden` named `default_forbidden_view`) to render internal exception responses. Those default view functions have been removed, replaced with a

generic default view function which is registered at Configurator setup for the `repoze.bfg.interfaces.IExceptionResponse` interface that simply returns the exception instance; the `NotFound` and `Forbidden` classes are now still exception factories but they are also response factories which generate instances that implement the new `repoze.bfg.interfaces.IExceptionResponse` interface.

1.3a7 (2010-08-01)

Features

- The `repoze.bfg.configuration.Configurator.add_route` API now returns the route object that was added.
- A `repoze.bfg.events.subscriber` decorator was added. This decorator decorates module-scope functions, which are then treated as event listeners after a `scan()` is performed. See the Events narrative documentation chapter and the `repoze.bfg.events` module documentation for more information.

Bug Fixes

- When adding a view for a route which did not yet exist (“did not yet exist” meaning, temporally, a view was added with a route name for a route which had not yet been added via `add_route`), the value of the `custom_predicate` argument to `add_view` was lost. Symptom: wrong view matches when using URL dispatch and custom view predicates together.
- Pattern matches for a `:segment` marker in a URL dispatch route pattern now always match at least one character. See “Backwards Incompatibilities” below in this changelog.

Backwards Incompatibilities

- A bug existed in the regular expression to do URL matching. As an example, the URL matching machinery would cause the pattern `{foo}` to match the root URL `/` resulting in a match dictionary of `{'foo':u''}` or the pattern `{fud}/edit` might match the URL `//edit` resulting in a match dictionary of `{'fud':u''}`. It was always the intent that `:segment` markers in the pattern would need to match *at least one* character, and never match the empty string. This, however, means that in certain circumstances, a routing match which your application inadvertently depended upon may no longer happen.

Documentation

- Added description of the `repoze.bfg.events.subscriber` decorator to the Events narrative chapter.
- Added `repoze.bfg.events.subscriber` API documentation to `repoze.bfg.events` API docs.
- Added a section named “Zope 3 Enforces ‘TTW’ Authorization Checks By Default; BFG Does Not” to the “Design Defense” chapter.

1.3a6 (2010-07-25)

Features

- New argument to `repoze.bfg.configuration.Configurator.add_route` and the route ZCML directive: `traverse`. If you would like to cause the context to be something other than the root object when this route matches, you can spell a traversal pattern as the `traverse` argument. This traversal pattern will be used as the traversal path: traversal will begin at the root object implied by this route (either the global root, or the object returned by the factory associated with this route).

The syntax of the `traverse` argument is the same as it is for `path`. For example, if the `path` provided is `articles/:article/edit`, and the `traverse` argument provided is `/:article`, when a request comes in that causes the route to match in such a way that the `article` match value is ‘1’ (when the request URI is `/articles/1/edit`), the traversal path will be generated as `/1`. This means that the root object’s `__getitem__` will be called with the name `1` during the traversal phase. If the `1` object exists, it will become the `context` of the request. The Traversal narrative has more information about traversal.

If the traversal path contains segment marker names which are not present in the `path` argument, a runtime error will occur. The `traverse` pattern should not contain segment markers that do not exist in the `path`.

A similar combining of routing and traversal is available when a route is matched which contains a `*traverse` remainder marker in its path. The `traverse` argument allows you to associate route patterns with an arbitrary traversal path without using a `*traverse` remainder marker; instead you can use other match information.

Note that the `traverse` argument is ignored when attached to a route that has a `*traverse` remainder marker in its path.

- A new method of the `Configurator` exists: `set_request_factory`. If used, this method will set the factory used by the `repoze.bfg` router to create all request objects.
- The `Configurator` constructor takes an additional argument: `request_factory`. If used, this argument will set the factory used by the `repoze.bfg` router to create all request objects.
- The `Configurator` constructor takes an additional argument: `request_factory`. If used, this argument will set the factory used by the `repoze.bfg` router to create all request objects.
- A new method of the `Configurator` exists: `set_renderer_globals_factory`. If used, this method will set the factory used by the `repoze.bfg` router to create renderer globals.
- A new method of the `Configurator` exists: `get_settings`. If used, this method will return the current settings object (performs the same job as the `repoze.bfg.settings.get_settings` API).
- The `Configurator` constructor takes an additional argument: `renderer_globals_factory`. If used, this argument will set the factory used by the `repoze.bfg` router to create renderer globals.
- Add `repoze.bfg.renderers.render`, `repoze.bfg.renderers.render_to_response` and `repoze.bfg.renderers.get_renderer` functions. These are imperative APIs which will use the same rendering machinery used by view configurations with a `renderer=` attribute/argument to produce a rendering or renderer. Because these APIs provide a central API for all rendering, they now form the preferred way to perform imperative template rendering. Using functions named `render_*` from modules such as `repoze.bfg.chameleon_zpt` and `repoze.bfg.chameleon_text` is now discouraged (although not deprecated). The code the backing older templating-system-specific APIs now calls into the newer `repoze.bfg.renderer` code.
- The `repoze.bfg.configuration.Configurator.testing_add_template` has been renamed to `testing_add_renderer`. A backwards compatibility alias is present using the old name.

Documentation

- The `Hybrid` narrative chapter now contains a description of the `traverse` route argument.
- The `Hooks` narrative chapter now contains sections about changing the request factory and adding a renderer globals factory.
- The API documentation includes a new module: `repoze.bfg.renderers`.
- The `Templates` chapter was updated; all narrative that used templating-specific APIs within examples to perform rendering (such as the `repoze.bfg.chameleon_zpt.render_template_to_response` method) was changed to use `repoze.bfg.renderers.render_*` functions.

Bug Fixes

- The header predicate (when used as either a view predicate or a route predicate) had a problem when specified with a name/regex pair. When the header did not exist in the headers dictionary, the regex match could be fed `None`, causing it to throw a `TypeError: expected string or buffer` exception. Now, the predicate returns `False` as intended.

Deprecations

- The `repoze.bfg.renderers.rendered_response` function was never an official API, but may have been imported by extensions in the wild. It is officially deprecated in this release. Use `repoze.bfg.renderers.render_to_response` instead.
- The following APIs are *documentation* deprecated (meaning they are officially deprecated in documentation but do not raise a deprecation error upon their usage, and may continue to work for an indefinite period of time):

In the `repoze.bfg.chameleon_zpt` module: `get_renderer`, `get_template`, `render_template`, `render_template_to_response`. The suggested alternatives are documented within the docstrings of those methods (which are still present in the documentation).

In the `repoze.bfg.chameleon_text` module: `get_renderer`, `get_template`, `render_template`, `render_template_to_response`. The suggested alternatives are documented within the docstrings of those methods (which are still present in the documentation).

In general, to perform template-related functions, one should now use the various methods in the `repoze.bfg.renderers` module.

Backwards Incompatibilities

- A new internal exception class (*not* an API) named `repoze.bfg.exceptions.PredicateMismatch` now exists. This exception is currently raised when no constituent view of a multiview can be called (due to no predicate match). Previously, in this situation, a `repoze.bfg.exceptions.NotFound` was raised. We provide backwards compatibility for code that expected a `NotFound` to be raised when no predicates match by causing `repoze.bfg.exceptions.PredicateMismatch` to inherit from `NotFound`. This will cause any exception view registered for `NotFound` to be called when a predicate mismatch occurs, as was the previous behavior.

There is however, one perverse case that will expose a backwards incompatibility. If 1) you had a view that was registered as a member of a multiview 2) this view explicitly raised a `NotFound` exception *in order to* proceed to the next predicate check in the multiview, that code will now behave differently: rather than skipping to the next view match, a `NotFound` will be raised to the top-level exception handling machinery instead. For code to be depending upon the behavior of a view raising `NotFound` to proceed to the next predicate match, would be tragic, but not impossible, given that `NotFound` is a public interface. `repoze.bfg.exceptions.PredicateMismatch` is not a public API and cannot be depended upon by application code, so you should not change your view code to raise `PredicateMismatch`. Instead, move the logic which raised the `NotFound` exception in the view out into a custom view predicate.

- If, when you run your application’s unit test suite under BFG 1.3, a `KeyError` naming a template or a `ValueError` indicating that a ‘renderer factory’ is not registered may be raised (e.g. `ValueError: No factory for renderer named '.pt' when looking up karl.views:templates/snippets.pt`), you may need to perform some extra setup in your test code.

The best solution is to use the `repoze.bfg.configuration.Configurator.testing_add_renderer` (or, alternately the deprecated `repoze.bfg.testing.registerTemplateRenderer` or `registerDummyRenderer`) API within the code comprising each individual unit test suite to register a “dummy” renderer for each of the templates and renderers used by code under test. For example:

```
config = Configurator()
config.testing_add_renderer('karl.views:templates/snippets.pt')
```

This will register a basic dummy renderer for this particular missing template. The `testing_add_renderer` API actually *returns* the renderer, but if you don’t care about how the render is used, you don’t care about having a reference to it either.

A more rough way to solve the issue exists. It causes the “real” template implementations to be used while the system is under test, which is suboptimal, because tests will run slower, and unit tests won’t actually *be* unit tests, but it is easier. Always ensure you call the `setup_registry()` method of the `Configurator`. Eg:

```
reg = MyRegistry()
config = Configurator(registry=reg)
config.setup_registry()
```

Calling `setup_registry` only has an effect if you’re *passing in* a `registry` argument to the `Configurator` constructor. `setup_registry` is called by the course of normal operations anyway if you do not pass in a `registry`.

If your test suite isn't using a Configurator yet, and is still using the older `repoze.bfg.testing` APIs name `setUp` or `cleanUp`, these will register the renderers on your behalf.

A variant on the symptom for this theme exists: you may already be dutifully registering a dummy template or renderer for a template used by the code you're testing using `testing_register_renderer` or `registerTemplateRenderer`, but (perhaps unbeknownst to you) the code under test expects to be able to use a "real" template renderer implementation to retrieve or render *another* template that you forgot was being rendered as a side effect of calling the code you're testing. This happened to work because it found the *real* template while the system was under test previously, and now it cannot. The solution is the same.

It may also help reduce confusion to use a *resource specification* to specify the template path in the test suite and code rather than a relative path in either. A resource specification is unambiguous, while a relative path needs to be relative to "here", where "here" isn't always well-defined ("here" in a test suite may or may not be the same as "here" in the code under test).

1.3a5 (2010-07-14)

Features

- New internal exception: `repoze.bfg.exceptions.URLDecodeError`. This URL is a subclass of the built-in Python exception named `UnicodeDecodeError`.
- When decoding a URL segment to Unicode fails, the exception raised is now `repoze.bfg.exceptions.URLDecodeError` instead of `UnicodeDecodeError`. This makes it possible to register an exception view invoked specifically when `repoze.bfg` cannot decode a URL.

Bug Fixes

- Fix regression in `repoze.bfg.configuration.Configurator.add_static_view`. Before 1.3a4, view names that contained a slash were supported as route prefixes. 1.3a4 broke this by trying to treat them as full URLs.

Documentation

- The `repoze.bfg.exceptions.URLDecodeError` exception was added to the exceptions chapter of the API documentation.

Backwards Incompatibilities

- in previous releases, when a URL could not be decoded from UTF-8 during traversal, a `TypeError` was raised. Now the error which is raised is a `repoze.bfg.exceptions.URLDecodeError`.

1.3a4 (2010-07-03)

Features

- Undocumented hook: make `get_app` and `get_root` of the `repoze.bfg.paster.BFGShellCommand` hookable in cases where endware may interfere with the default versions.
- In earlier versions, a custom route predicate associated with a url dispatch route (each of the predicate functions fed to the `custom_predicates` argument of `repoze.bfg.configuration.Configurator.add_route`) has always required a 2-positional argument signature, e.g. `(context, request)`. Before this release, the `context` argument was always `None`.

As of this release, the first argument passed to a predicate is now a dictionary conventionally named `info` consisting of `route`, and `match`. `match` is a dictionary: it represents the arguments matched in the URL by the route. `route` is an object representing the route which was matched.

This is useful when predicates need access to the route match. For example:

```
def any_of(segment_name, *args):
    def predicate(info, request):
        if info['match'][segment_name] in args:
            return True
    return predicate

num_one_two_or_three = any_of('num', 'one', 'two', 'three')

add_route('num', '/*:num', custom_predicates=(num_one_two_or_three,))
```

The `route` object is an object that has two useful attributes: `name` and `path`. The `name` attribute is the route name. The `path` attribute is the route pattern. An example of using the route in a set of route predicates:

```
def twenty_ten(info, request):
    if info['route'].name in ('ymd', 'ym', 'y'):
        return info['match']['year'] == '2010'

add_route('y', '/:year', custom_predicates=(twenty_ten,))
add_route('ym', '/:year/:month', custom_predicates=(twenty_ten,))
add_route('ymd', '/:year/:month/:day', custom_predicates=(twenty_ten,))
```

- The `repoze.bfg.url.route_url` API has changed. If a keyword `_app_url` is present in the arguments passed to `route_url`, this value will be used as the protocol/hostname/port/leading path prefix of the generated URL. For example, using an `_app_url` of `http://example.com:8080/foo` would cause the URL `http://example.com:8080/foo/fleeb/flub` to be returned from this function if the expansion of the route pattern associated with the `route_name` expanded to `/fleeb/flub`.
- It is now possible to use a URL as the name argument fed to `repoze.bfg.configuration.Configurator.add_static_view`. When the name argument is a URL, the `repoze.bfg.url.static_url` API will generate join this URL (as a prefix) to a path including the static file name. This makes it more possible to put static media on a separate webserver for production, while keeping static media package-internal and served by the development webserver during development.

Documentation

- The authorization chapter of the ZODB Wiki Tutorial (`docs/tutorials/bfgwiki`) was changed to demonstrate authorization via a group rather than via a direct username (thanks to Alex Marandon).
- The authorization chapter of the SQLAlchemy Wiki Tutorial (`docs/tutorials/bfgwiki2`) was changed to demonstrate authorization via a group rather than via a direct username.
- Redirect requests for tutorial sources to `http://docs.repoze.org/bfgwiki-1.3` and `http://docs.repoze.org/bfgwiki2-1.3/` respectively.
- A section named `Custom Route Predicates` was added to the URL Dispatch narrative chapter.
- The Static Resources chapter has been updated to mention using `static_url` to generate URLs to external webserver.

Internal

- Removed `repoze.bfg.static.StaticURLFactory` in favor of a new abstraction revolving around the (still-internal) `repoze.bfg.static.StaticURLInfo` helper class.

1.3a3 (2010-05-01)

Paster Templates

- The `bfg_alchemy` and `bfg_routesalchemy` templates no longer register a `handle_teardown` event listener which calls `DBSession.remove`. This was found by Chris Withers to be unnecessary.

Documentation

- The “bfgwiki2” (URL dispatch wiki) tutorial code and documentation was changed to remove the `handle_teardown` event listener which calls `DBSession.remove`.
- Any mention of the `handle_teardown` event listener as used by the paster templates was removed from the URL Dispatch narrative chapter.
- A section entitled Detecting Available Languages was added to the `il8n` narrative docs chapter.

1.3a2 (2010-04-28)

Features

- A locale negotiator no longer needs to be registered explicitly. The default locale negotiator at `repoze.bfg.il8n.default_locale_negotiator` is now used unconditionally as...um, the default locale negotiator.
- The default locale negotiator has become more complex.
 - First, the negotiator looks for the `__LOCALE__` attribute of the request object (possibly set by a view or an event listener).
 - Then it looks for the `request.params['_LOCALE_']` value.
 - Then it looks for the `request.cookies['_LOCALE_']` value.

Backwards Incompatibilities

- The default locale negotiator now looks for the parameter named `_LOCALE_` rather than a parameter named `locale` in `request.params`.

Behavior Changes

- A locale negotiator may now return `None`, signifying that the default locale should be used.

Documentation

- Documentation concerning locale negotiation in the Internationalization and Localization chapter was updated.
- Expanded portion of i18n narrative chapter docs which discuss working with gettext files.

1.3a1 (2010-04-26)

Features

- Added “exception views”. When you use an exception (anything that inherits from the Python `Exception` builtin) as view context argument, e.g.:

```
from repoze.bfg.view import bfg_view
from repoze.bfg.exceptions import NotFound
from webob.exc import HTTPNotFound

@bfg_view(context=NotFound)
def notfound_view(request):
    return HTTPNotFound()
```

For the above example, when the `repoze.bfg.exceptions.NotFound` exception is raised by any view or any root factory, the `notfound_view` view callable will be invoked and its response returned.

Other normal view predicates can also be used in combination with an exception view registration:

```
from repoze.bfg.view import bfg_view
from repoze.bfg.exceptions import NotFound
from webob.exc import HTTPNotFound

@bfg_view(context=NotFound, route_name='home')
def notfound_view(request):
    return HTTPNotFound()
```

The above exception view names the `route_name` of `home`, meaning that it will only be called when the route matched has a name of `home`. You can therefore have more than one exception view for any given exception in the system: the “most specific” one will be called when the set of request circumstances which match the view registration. The only predicate that cannot be not be used successfully is `name`. The name used to look up an exception view is always the empty string.

Existing (pre-1.3) normal views registered against objects inheriting from `Exception` will continue to work. Exception views used for user-defined exceptions and system exceptions used as contexts will also work.

The feature can be used with any view registration mechanism (`@bfg_view` decorator, ZCML, or imperative `config.add_view` styles).

This feature was kindly contributed by Andrey Popp.

- Use “Venusian” (<http://docs.repoze.org/venusian>) to perform `bfg_view` decorator scanning rather than relying on a BFG-internal decorator scanner. (Truth be told, Venusian is really just a generalization of the BFG-internal decorator scanner).
- Internationalization and localization features as documented in the narrative documentation chapter entitled `Internationalization and Localization`.
- A new deployment setting named `default_locale_name` was added. If this string is present as a Paster `.ini` file option, it will be considered the default locale name. The default locale name is used during locale-related operations such as language translation.
- It is now possible to turn on Chameleon template “debugging mode” for all Chameleon BFG templates by setting a BFG-related Paster `.ini` file setting named `debug_templates`. The exceptions raised by Chameleon templates when a rendering fails are sometimes less than helpful. `debug_templates` allows you to configure your application development environment so that exceptions generated by Chameleon during template compilation and execution will contain more helpful debugging information. This mode is on by default in all new projects.
- Add a new method of the Configurator named `derive_view` which can be used to generate a BFG view callable from a user-supplied function, instance, or class. This useful for external framework and plugin authors wishing to wrap callables supplied by their users which follow the same calling conventions and response conventions as objects that can be supplied directly to BFG as a view callable. See the `derive_view` method in the `repoze.bfg.configuration.Configurator` docs.

ZCML

- Add a `translationdir` ZCML directive to support localization.
- Add a `localnegotiator` ZCML directive to support localization.

Deprecations

- The `exception views` feature replaces the need for the `set_notfound_view` and `set_forbidden_view` methods of the `Configurator` as well as the `notfound` and `forbidden` ZCML directives. Those methods and directives will continue to work for the foreseeable future, but they are deprecated in the documentation.

Dependencies

- A new install-time dependency on the `venusian` distribution was added.
- A new install-time dependency on the `translationstring` distribution was added.
- Chameleon 1.2.3 or better is now required (internationalization and per-template debug settings).

Internal

- View registrations and lookups are now done with three “requires” arguments instead of two to accomodate orthogonality of exception views.
- The `repoze.bfg.interfaces.IForbiddenView` and `repoze.bfg.interfaces.INotFoundView` interfaces were removed; they weren’t APIs and they became vestigial with the addition of exception views.
- Remove `repoze.bfg.compat.pkgutil_26.py` and `import alias repoze.bfg.compat.walk_packages`. These were only required by internal scanning machinery; Venusian replaced the internal scanning machinery, so these are no longer required.

Documentation

- Exception view documentation was added to the `Hooks` narrative chapter.
- A new narrative chapter entitled `Internationalization and Localization` was added.
- The “Environment Variables and ini File Settings” chapter was changed: documentation about the `default_locale_name` setting was added.
- A new API chapter for the `repoze.bfg.i18n` module was added.
- Documentation for the new `translationdir` and `localenegotiator` ZCML directives were added.
- A section was added to the `Templates` chapter entitled “Nicer Exceptions in Templates” describing the result of setting `debug_templates = true`.

Paster Templates

- All paster templates now create a `setup.cfg` which includes commands related to nose testing and Babel message catalog extraction/compilation.
- A `default_locale_name = en` setting was added to each existing paster template.
- A `debug_templates = true` setting was added to each existing paster template.

Licensing

- The Edgewall (BSD) license was added to the `LICENSES.txt` file, as some code in the `repoze.bfg.i18n` derives from Babel source.

1.2 (2010-02-10)

- No changes from 1.2b6.

1.2b6 (2010-02-06)

Backwards Incompatibilities

- Remove magical feature of `repoze.bfg.url.model_url` which prepended a fully-expanded `urldispatch` route URL before a the model's path if it was noticed that the request had matched a route. This feature was ill-conceived, and didn't work in all scenarios.

Bug Fixes

- More correct conversion of provided `renderer` values to resource specification values (internal).

1.2b5 (2010-02-04)

Bug Fixes

- 1.2b4 introduced a bug whereby views added via a route configuration that named a view callable and also a `view_attr` became broken. Symptom: `MyViewClass` is not callable or the `__call__` of a class was being called instead of the method named via `view_attr`.
- Fix a bug whereby a `renderer` argument to the `@bfg_view` decorator that provided a package-relative template filename might not have been resolved properly. Symptom: inappropriate `Missing template resource errors`.

1.2b4 (2010-02-03)

Documentation

- Update GAE tutorial to use Chameleon instead of Jinja2 (now that it's possible).

Bug Fixes

- Ensure that `secure` flag for `AuthTktAuthenticationPolicy` constructor does what it's documented to do (merge Daniel Holth's fancy-cookies-2 branch).

Features

- Add `path` and `http_only` options to `AuthTktAuthenticationPolicy` constructor (merge Daniel Holth's fancy-cookies-2 branch).

Backwards Incompatibilities

- Remove `view_header`, `view_accept`, `view_xhr`, `view_path_info`, `view_request_method`, `view_request_param`, and `view_containment` predicate arguments from the `Configurator.add_route` argument list. These arguments were speculative. If you need the features exposed by these arguments, add a view associated with a route using the `route_name` argument to the `add_view` method instead.
- Remove `view_header`, `view_accept`, `view_xhr`, `view_path_info`, `view_request_method`, `view_request_param`, and `view_containment` predicate arguments from the route ZCML directive attribute set. These attributes were speculative. If you need the features exposed by these attributes, add a view associated with a route using the `route_name` attribute of the `view` ZCML directive instead.

Dependencies

- Remove dependency on `sourcecodegen` (not depended upon by Chameleon 1.1.1+).

1.2b3 (2010-01-24)

Bug Fixes

- When “hybrid mode” (both `traversal` and `urldispatch`) is in use, default to finding route-related views even if a non-route-related view registration has been made with a more specific context. The default used to be to find views with a more specific context first. Use the new `use_global_views` argument to the route definition to get back the older behavior.

Features

- Add `use_global_views` argument to `add_route` method of `Configurator`. When this argument is true, views registered for *no* route will be found if no more specific view related to the route is found.
- Add `use_global_views` attribute to ZCML `<route>` directive (see above).

Internal

- When registering a view, register the view adapter with the “requires” interfaces as `(request_type, context_type)` rather than `(context_type, request_type)`. This provides for saner lookup, because the registration will always be made with a specific request interface, but registration may not be made with a specific context interface. In general, when creating multiadapters, you want to order the requires interfaces so that the elements which are more likely to be registered using specific interfaces are ordered before those which are less likely.

1.2b2 (2010-01-21)

Bug Fixes

- When the `Configurator` is passed an instance of `zope.component.registry.Components` as a `registry` constructor argument, fix the instance up to have the attributes we expect of an instance of `repoze.bfg.registry.Registry` when `setup_registry` is called. This makes it possible to use the global Zope component registry as a BFG application registry.
- When WebOb 0.9.7.1 was used, a deprecation warning was issued for the class attribute named `charset` within `repoze.bfg.request.Request`. BFG now *requires* WebOb \geq 0.9.7, and code was added so that this deprecation warning has disappeared.
- Fix a view lookup ordering bug whereby a view with a larger number of predicates registered first (literally first, not “earlier”) for a triad would lose during view lookup to one registered with fewer.
- Make sure views with exactly N custom predicates are always called before views with exactly N non-custom predicates given all else is equal in the view configuration.

Documentation

- Change renderings of ZCML directive documentation.
- Add a narrative documentation chapter: “Using the Zope Component Architecture in repoze.bfg”.

Dependencies

- Require WebOb >= 0.9.7

1.2b1 (2010-01-18)

Bug Fixes

- In `bfg_routesalchemy`, `bfg_alchemy` paster templates and the `bfgwiki2` tutorial, clean up the SQLAlchemy connection by registering a `repoze.tm.after_end` callback instead of relying on a `__del__` method of a `Cleanup` class added to the WSGI environment. The `__del__` strategy was fragile and caused problems in the wild. Thanks to Daniel Holth for testing.

Features

- Read logging configuration from PasteDeploy config file `loggers` section (and related) when `paster bfgshell` is invoked.

Documentation

- Major rework in preparation for book publication.

1.2a11 (2010-01-05)

Bug Fixes

- Make `paster bfgshell` and `paster create -t bfg_xxx` work on Jython (fix minor incompatibility with treatment of `__doc__` at the class level).
- Updated dependency on WebOb to require a version which supports features now used in tests.

Features

- Jython compatibility (at least when `repoze.bfg.jinja2` is used as the templating engine; Chameleon does not work under Jython).
- Show the derived abspath of template resource specifications in the traceback when a renderer template cannot be found.
- Show the original traceback when a Chameleon template cannot be rendered due to a platform incompatibility.

1.2a10 (2010-01-04)

Features

- The `Configurator.add_view` method now accepts an argument named `context`. This is an alias for the older argument named `for_`; it is preferred over `for_`, but `for_` will continue to be supported “forever”.
- The `view` ZCML directive now accepts an attribute named `context`. This is an alias for the older attribute named `for`; it is preferred over `for`, but `for` will continue to be supported “forever”.
- The `Configurator.add_route` method now accepts an argument named `view_context`. This is an alias for the older argument named `view_for`; it is preferred over `view_for`, but `view_for` will continue to be supported “forever”.
- The `route` ZCML directive now accepts an attribute named `view_context`. This is an alias for the older attribute named `view_for`; it is preferred over `view_for`, but `view_for` will continue to be supported “forever”.

Documentation and Paster Templates

- LaTeX rendering tweaks.
- All uses of the `Configurator.add_view` method that used its `for_` argument now use the `context` argument instead.
- All uses of the `Configurator.add_route` method that used its `view_for` argument now use the `view_context` argument instead.
- All uses of the `view` ZCML directive that used its `for` attribute now use the `context` attribute instead.
- All uses of the `route` ZCML directive that used its `view_for` attribute now use the `view_context` attribute instead.
- Add a (minimal) tutorial dealing with use of `repoze.catalog` in a `repoze.bfg` application.

Documentation Licensing

- Loosen the documentation licensing to allow derivative works: it is now offered under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. This is only a documentation licensing change; the `repoze.bfg` software continues to be offered under the Repoze Public License at <http://repoze.org/license.html> (BSD-like).

1.2a9 (2009-12-27)

Documentation Licensing

- The *documentation* (the result of `make <html|latex|htmlhelp>` within the `docs` directory) in this release is now offered under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License as described by <http://creativecommons.org/licenses/by-nc-nd/3.0/us/> . This is only a licensing change for the documentation; the `repoze.bfg` software continues to be offered under the Repoze Public License at <http://repoze.org/license.html> (BSD-like).

Documentation

- Added manual index entries to generated index.
- Document the previously existing (but non-API) `repoze.bfg.configuration.Configurator.setup_registry` method as an official API of a `Configurator`.
- Fix syntax errors in various documentation code blocks.
- Created new top-level documentation section: “ZCML Directives”. This section contains detailed ZCML directive information, some of which was removed from various narrative chapters.
- The LaTeX rendering of the documentation has been improved.
- Added a “Fore-Matter” section with author, copyright, and licensing information.

1.2a8 (2009-12-24)

Features

- Add a `**kw` arg to the `Configurator.add_settings` API.
- Add `hook_zca` and `unhook_zca` methods to the `Configurator` API.
- The `repoze.bfg.testing.setUp` method now returns a `Configurator` instance which can be used to do further configuration during unit tests.

Bug Fixes

- The `json` renderer failed to set the response content type to `application/json`. It now does, by setting `request.response_content_type` unless this attribute is already set.
- The `string` renderer failed to set the response content type to `text/plain`. It now does, by setting `request.response_content_type` unless this attribute is already set.

Documentation

- General documentation improvements by using better Sphinx roles such as “`class`”, “`func`”, “`meth`”, and so on. This means that there are many more hyperlinks pointing to API documentation for API definitions in all narrative, tutorial, and API documentation elements.
- Added a description of imperative configuration in various places which only described ZCML configuration.
- A syntactical refreshing of various tutorials.
- Added the `repoze.bfg.authentication`, `repoze.bfg.authorization`, and `repoze.bfg.interfaces` modules to API documentation.

Deprecations

- The `repoze.bfg.testing.registerRoutesMapper` API (added in an early 1.2 alpha) was deprecated. Its import now generates a deprecation warning.

1.2a7 (2009-12-20)

Features

- Add four new testing-related APIs to the `repoze.bfg.configuration.Configurator` class: `testing_securitypolicy`, `testing_models`, `testing_add_subscriber`, and `testing_add_template`. These were added in order to provide more direct access to the functionality of the `repoze.bfg.testing` APIs named `registerDummySecurityPolicy`, `registerModels`, `registerEventListener`, and `registerTemplateRenderer` when a configurator is used. The testing APIs named are nominally deprecated (although they will likely remain around “forever”, as they are in heavy use in the wild).
- Add a new API to the `repoze.bfg.configuration.Configurator` class: `add_settings`. This API can be used to add “settings” (information returned within via the `repoze.bfg.settings.get_settings` API) after the configurator has been initially set up. This is most useful for testing purposes.
- Add a `custom_predicates` argument to the `Configurator add_view` method, the `bfg_view` decorator and the attribute list of the ZCML view directive. If `custom_predicates` is specified, it must be a sequence of predicate callables (a predicate callable accepts two arguments: `context` and `request` and returns `True` or `False`). The associated view callable will only be invoked if all custom predicates return `True`. Use one or more custom predicates when no existing predefined predicate is useful. Predefined and custom predicates can be mixed freely.
- Add a `custom_predicates` argument to the `Configurator add_route` and the attribute list of the ZCML route directive. If `custom_predicates` is specified, it must be a sequence of predicate callables (a predicate callable accepts two arguments: `context` and `request` and returns `True` or `False`). The associated route will match will only be invoked if all custom predicates return `True`, else route matching continues. Note that the value `context` will always be `None` when passed to a custom route predicate. Use one or more custom predicates when no existing predefined predicate is useful. Predefined and custom predicates can be mixed freely.

Internal

- Remove the `repoze.bfg.testing.registerTraverser` function. This function was never an API.

Documentation

- Doc-deprecated most helper functions in the `repoze.bfg.testing` module. These helper functions likely won't be removed any time soon, nor will they generate a warning any time soon, due to their heavy use in the wild, but equivalent behavior exists in methods of a `Configurator`.

1.2a6 (2009-12-18)

Features

- The `Configurator` object now has two new methods: `begin` and `end`. The `begin` method is meant to be called before any “configuration” begins (e.g. before `add_view`, et. al are called). The `end` method is meant to be called after all “configuration” is complete.

Previously, before there was imperative configuration at all (1.1 and prior), configuration `begin` and `end` was invariably implied by the process of loading a ZCML file. When a ZCML load happened, the threadlocal data structure containing the request and registry was modified before the load, and torn down after the load, making sure that all framework code that needed `get_current_registry` for the duration of the ZCML load was satisfied.

Some API methods called during imperative configuration, (such as `Configurator.add_view` when a renderer is involved) end up for historical reasons calling `get_current_registry`. However, in 1.2a5 and below, the `Configurator` supplied no functionality that allowed people to make sure that `get_current_registry` returned the registry implied by the configurator being used. `begin` now serves this purpose. Inversely, `end` pops the thread local stack, undoing the actions of `begin`.

We make this boundary explicit to reduce the potential for confusion when the configurator is used in different circumstances (e.g. in unit tests and app code vs. just in initial app setup).

Existing code written for 1.2a1-1.2a5 which does not call `begin` or `end` continues to work in the same manner it did before. It is however suggested that this code be changed to call `begin` and `end` to reduce the potential for confusion in the future.

- All `paster` templates which generate an application skeleton now make use of the new `begin` and `end` methods of the `Configurator` they use in their respective copies of `run.py` and `tests.py`.

Documentation

- All documentation that makes use of a `Configurator` object to do application setup and test setup now makes use of the new `begin` and `end` methods of the configurator.

Bug Fixes

- When a `repoze.bfg.exceptions.NotFound` or `repoze.bfg.exceptions.Forbidden` *class* (as opposed to instance) was raised as an exception within a root factory (or route root factory), the exception would not be caught properly by the `repoze.bfg.Router` and it would propagate to up the call stack, as opposed to rendering the not found view or the forbidden view as would have been expected.
- When Chameleon page or text templates used as renderers were added imperatively (via `Configurator.add_view` or some derivative), they too-eagerly attempted to look up the `reload_templates` setting via `get_settings`, meaning they were always registered in non-auto-reload-mode (the default). Each now waits until its respective `template` attribute is accessed to look up the value.
- When a route with the same name as a previously registered route was added, the old route was not removed from the mapper's routelist. Symptom: the old registered route would be used (and possibly matched) during route lookup when it should not have had a chance to ever be used.

1.2a5 (2009-12-10)

Features

- When the `repoze.bfg.exceptions.NotFound` or `repoze.bfg.exceptions.Forbidden` error is raised from within a custom root factory or the factory of a route, the appropriate response is now sent to the requesting user agent (the result of the notfound view or the forbidden view, respectively). When these errors are raised from within a root factory, the context passed to the notfound or forbidden view will be `None`. Also, the request will not be decorated with `view_name`, `subpath`, `context`, etc. as would normally be the case if traversal had been allowed to take place.

Internals

- The exception class representing the error raised by various methods of a `Configurator` is now importable as `repoze.bfg.exceptions.ConfigurationError`.

Documentation

- General documentation freshening which takes imperative configuration into account in more places and uses glossary references more liberally.
- Remove explanation of changing the request type in a new request event subscriber, as other predicates are now usually an easier way to get this done.
- Added “Thread Locals” narrative chapter to documentation, and added a API chapter documenting the `repoze.bfg.threadlocals` module.
- Added a “Special Exceptions” section to the “Views” narrative documentation chapter explaining the effect of raising `repoze.bfg.exceptions.NotFound` and `repoze.bfg.exceptions.Forbidden` from within view code.

Dependencies

- A new dependency on the `twill` package was added to the `setup.py tests_require` argument (Twill will only be downloaded when `repoze.bfg setup.py test` or `setup.py nosetests` is invoked).

1.2a4 (2009-12-07)

Features

- `repoze.bfg.testing.DummyModel` now accepts a new constructor keyword argument: `__provides__`. If this constructor argument is provided, it should be an interface or a tuple of interfaces. The resulting model will then provide these interfaces (they will be attached to the constructed model via `zope.interface.alsoProvides`).

Bug Fixes

- Operation on GAE was broken, presumably because the `repoze.bfg.configuration` module began to attempt to import the `repoze.bfg.chameleon_zpt` and `repoze.bfg.chameleon_text` modules, and these cannot be used on non-CPython platforms. It now tolerates startup time import failures for these modules, and only raise an import error when a template from one of these packages is actually used.

1.2a3 (2009-12-02)

Bug Fixes

- The `repoze.bfg.url.route_url` function inappropriately passed along `_query` and/or `_anchor` arguments to the `mapper.generate` function, resulting in blowups.
- When two views were registered with differering `for` interfaces or classes, and the `for` of first view registered was a superclass of the second, the `repoze.bfg` view machinery would incorrectly associate the two views with the same “multiview”. Multiviews are meant to be collections of views that have *exactly* the same `for/request/viewname` values, without taking inheritance into account. Symptom: wrong view callable found even when you had correctly specified a `for_` interface/class during view configuration for one or both view configurations.

Backwards Incompatibilities

- The `repoze.bfg.templating` module has been removed; it had been deprecated in 1.1 and never actually had any APIs in it.

1.2a2 (2009-11-29)

Bug Fixes

- The long description of this package (as shown on PyPI) was not valid reStructuredText, and so was not renderable.
- Trying to use an HTTP method name string such as `GET` as a `request_type` predicate argument caused a startup time failure when it was encountered in imperative configuration or in a decorator (symptom: `Type Error: Required specification must be a specification`). This now works again, although `request_method` is now the preferred predicate argument for associating a view configuration with an HTTP request method.

Documentation

- Fixed “Startup” narrative documentation chapter; it was explaining “the old way” an application constructor worked.

1.2a1 (2009-11-28)

Features

- An imperative configuration mode.

A `repoze.bfg` application can now begin its life as a single Python file. Later, the application might evolve into a set of Python files in a package. Even later, it might start making use of other configuration features, such as ZCML. But neither the use of a package nor the use of non-imperative configuration is required to create a simple `repoze.bfg` application any longer.

Imperative configuration makes `repoze.bfg` competitive with “microframeworks” such as Bottle and Tornado. `repoze.bfg` has a good deal of functionality that most microframeworks lack, so this is hopefully a “best of both worlds” feature.

The simplest possible `repoze.bfg` application is now:

```
from webob import Response
from wsgiref import simple_server
from repoze.bfg.configuration import Configurator

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_view(hello_world)
    app = config.make_wsgi_app()
    simple_server.make_server(' ', 8080, app).serve_forever()
```

- A new class now exists: `repoze.bfg.configuration.Configurator`. This class forms the basis for sharing machinery between “imperatively” configured applications and traditional declaratively-configured applications.
- The `repoze.bfg.testing.setUp` function now accepts three extra optional keyword arguments: `registry`, `request` and `hook_zca`.

If the `registry` argument is not `None`, the argument will be treated as the registry that is set as the “current registry” (it will be returned by `repoze.bfg.threadlocal.get_current_registry`) for the duration of the test. If the `registry` argument is `None` (the default), a new registry is created and used for the duration of the test.

The value of the `request` argument is used as the “current request” (it will be returned by `repoze.bfg.threadlocal.get_current_request`) for the duration of the test; it defaults to `None`.

If `hook_zca` is `True` (the default), the `zope.component.getSiteManager` function will be hooked with a function that returns the value of `registry` (or the default-created `registry` if `registry` is `None`) instead of the `registry` returned by `zope.component.getGlobalSiteManager`, causing the Zope Component Architecture API (`getSiteManager`, `getAdapter`, `getUtility`, and so on) to use the testing `registry` instead of the global ZCA `registry`.

- The `repoze.bfg.testing.tearDown` function now accepts an `unhook_zca` argument. If this argument is `True` (the default), `zope.component.getSiteManager.reset()` will be called. This will cause the result of the `zope.component.getSiteManager` function to be the global ZCA `registry` (the result of `zope.component.getGlobalSiteManager`) once again.
- The `run.py` module in various `repoze.bfg` paster templates now use a `repoze.bfg.configuration.Configurator` class instead of the (now-legacy) `repoze.bfg.router.make_app` function to produce a WSGI application.

Documentation

- The documentation now uses the “request-only” view calling convention in most examples (as opposed to the `context`, `request` convention). This is a documentation-only change; the `context`, `request` convention is also supported and documented, and will be “forever”.
- `repoze.bfg.configuration` API documentation has been added.
- A narrative documentation chapter entitled “Creating Your First `repoze.bfg` Application” has been added. This chapter details usage of the new `repoze.bfg.configuration.Configurator` class, and demonstrates a simplified “imperative-mode” configuration; doing `repoze.bfg` application configuration imperatively was previously much more difficult.
- A narrative documentation chapter entitled “Configuration, Decorations and Code Scanning” explaining ZCML- vs. imperative- vs. decorator-based configuration equivalence.
- The “ZCML Hooks” chapter has been renamed to “Hooks”; it documents how to override hooks now via imperative configuration and ZCML.
- The explanation about how to supply an alternate “response factory” has been removed from the “Hooks” chapter. This feature may be removed in a later release (it still works now, it’s just not documented).
- Add a section entitled “Test Set Up and Tear Down” to the unittesting chapter.

Bug Fixes

- The ACL authorization policy debugging output when `debug_authorization` console debugging output was turned on wasn't as clear as it could have been when a view execution was denied due to an authorization failure resulting from the set of principals passed never having matched any ACE in any ACL in the lineage. Now in this case, we report `<default deny>` as the ACE value and either the root ACL or `<No ACL found on any object in model lineage>` if no ACL was found.
- When two views were registered with the same `accept` argument, but were otherwise registered with the same arguments, if a request entered the application which had an `Accept` header that accepted *either* of the media types defined by the set of views registered with predicates that otherwise matched, a more or less “random” one view would “win”. Now, we try harder to use the view callable associated with the view configuration that has the most specific `accept` argument. Thanks to Alberto Valverde for an initial patch.

Internals

- The routes mapper is no longer a root factory wrapper. It is now consulted directly by the router.
- The `repoze.bfg.registry.make_registry` callable has been removed.
- The `repoze.bfg.view.map_view` callable has been removed.
- The `repoze.bfg.view.owrap_view` callable has been removed.
- The `repoze.bfg.view.predicate_wrap` callable has been removed.
- The `repoze.bfg.view.secure_view` callable has been removed.
- The `repoze.bfg.view.authdebug_view` callable has been removed.
- The `repoze.bfg.view.renderer_from_name` callable has been removed. Use `repoze.bfg.configuration.Configurator.renderer_from_name` instead (still not an API, however).
- The `repoze.bfg.view.derive_view` callable has been removed. Use `repoze.bfg.configuration.Configurator.derive_view` instead (still not an API, however).
- The `repoze.bfg.settings.get_options` callable has been removed. Its job has been subsumed by the `repoze.bfg.settings.Settings` class constructor.

- The `repoze.bfg.view.requestonly` function has been moved to `repoze.bfg.configuration.requestonly`.
- The `repoze.bfg.view.rendered_response` function has been moved to `repoze.bfg.configuration.rendered_response`.
- The `repoze.bfg.view.decorate_view` function has been moved to `repoze.bfg.configuration.decorate_view`.
- The `repoze.bfg.view.MultiView` class has been moved to `repoze.bfg.configuration.MultiView`.
- The `repoze.bfg.zcml.Uncacheable` class has been removed.
- The `repoze.bfg.resource.resource_spec` function has been removed.
- All ZCML directives which deal with attributes which are paths now use the `path` method of the ZCML context to resolve a relative name to an absolute one (imperative configuration requirement).
- The `repoze.bfg.scripting.get_root` API now uses a ‘real’ WebOb request rather than a FakeRequest when it sets up the request as a threadlocal.
- The `repoze.bfg.traversal.traverse` API now uses a ‘real’ WebOb request rather than a FakeRequest when it calls the traverser.
- The `repoze.bfg.request.FakeRequest` class has been removed.
- Most uses of the ZCA threadlocal API (the `getSiteManager`, `getUtility`, `getAdapter`, `getMultiAdapter` threadlocal API) have been removed from the core. Instead, when a threadlocal is necessary, the core uses the `repoze.bfg.threadlocal.get_current_registry` API to obtain the registry.
- The internal ILogger utility named `repoze.bfg.debug` is now just an IDebugger unnamed utility. A named utility with the old name is registered for b/w compat.
- The `repoze.bfg.interfaces.ITemplateRendererFactory` interface was removed; it has become unused.
- Instead of depending on the `martian` package to do code scanning, we now just use our own scanning routines.
- We now no longer have a dependency on `repoze.zcml` package; instead, the `repoze.bfg` package includes implementations of the `adapter`, `subscriber` and `utility` directives.

- Relating to the following functions:

```
repoze.bfg.view.render_view
repoze.bfg.view.render_view_to_iterable
repoze.bfg.view.render_view_to_response
repoze.bfg.view.append_slash_notfound_view
repoze.bfg.view.default_notfound_view
repoze.bfg.view.default_forbidden_view
repoze.bfg.configuration.rendered_response
repoze.bfg.security.has_permission
repoze.bfg.security.authenticated_userid
repoze.bfg.security.effective_principals
repoze.bfg.security.view_execution_permitted
repoze.bfg.security.remember
repoze.bfg.security.forget
repoze.bfg.url.route_url
repoze.bfg.url.model_url
repoze.bfg.url.static_url
repoze.bfg.traversal.virtual_root
```

Each of these functions now expects to be called with a request object that has a `registry` attribute which represents the current `repoze.bfg` registry. They fall back to obtaining the registry from the `threadlocal` API.

Backwards Incompatibilites

- Unit tests which use `zope.testing.cleanup.cleanUp` for the purpose of isolating tests from one another may now begin to fail due to lack of isolation between tests.

Here's why: In `repoze.bfg` 1.1 and prior, the registry returned by `repoze.bfg.threadlocal.get_current_registry` when no other registry had been pushed on to the threadlocal stack was the `zope.component.globalregistry.base` global registry (aka the result of `zope.component.getGlobalSiteManager()`). In `repoze.bfg` 1.2+, however, the registry returned in this situation is the new module-scope `repoze.bfg.registry.global_registry` object. The `zope.testing.cleanup.cleanUp` function clears the `zope.component.globalregistry.base` global registry unconditionally. However, it does not know about the `repoze.bfg.registry.global_registry` object, so it does not clear it.

If you use the `zope.testing.cleanup.cleanUp` function in the `setUp` of test cases in your unit test suite instead of using the (more correct as of 1.1) `repoze.bfg.testing.setUp`, you will need to replace all calls to `zope.testing.cleanup.cleanUp` with a call to `repoze.bfg.testing.setUp`.

If replacing all calls to `zope.testing.cleanup.cleanUp` with a call to `repoze.bfg.testing.setUp` is infeasible, you can put this bit of code somewhere that is executed exactly **once** (not for each test in a test suite; in the “`__init__.py`” of your package or your package's tests subpackage would be a reasonable place):

```
import zope.testing.cleanup
from repoze.bfg.testing import setUp
zope.testing.cleanup.addCleanUp(setUp)
```

- When there is no “current registry” in the `repoze.bfg.threadlocal.manager` threadlocal data structure (this is the case when there is no “current request” or we're not in the midst of a `r.b.testing.setUp`-bounded unit test), the `.get` method of the manager returns a data structure containing a *global* registry. In previous releases, this function returned the global Zope “base” registry: the result of `zope.component.getGlobalSiteManager`, which is an instance of the `zope.component.registry.Component` class. In this release, however, the global registry returns a globally importable instance of the `repoze.bfg.registry.Registry` class. This registry instance can always be imported as `repoze.bfg.registry.global_registry`.

Effectively, this means that when you call `repoze.bfg.threadlocal.get_current_registry` when no request or `setUp` bounded unit test is in effect, you will always get back the global registry that lives in `repoze.bfg.registry.global_registry`. It also means that `repoze.bfg` APIs that *call* `get_current_registry` will use this registry.

This change was made because `repoze.bfg` now expects the registry it uses to have a slightly different API than a bare instance of `zope.component.registry.Components`.

- View registration no longer registers a `repoze.bfg.interfaces.IViewPermission` adapter (it is no longer checked by the framework; since 1.1, views have been responsible for providing their own security).
- The `repoze.bfg.router.make_app` callable no longer accepts the `authentication_policy` nor the `authorization_policy` arguments. This feature was deprecated in version 1.0 and has been removed.
- Obscure: the machinery which configured views with a `request_type` *and* a `route_name` would ignore the request interface implied by `route_name` registering a view only for the interface implied by `request_type`. In the unlikely event that you were trying to use these two features together, the symptom would have been that views that named a `request_type` but which were also associated with routes were not found when the route matched. Now if a view is configured with both a `request_type` and a `route_name`, an error is raised.
- The `route` ZCML directive now no longer accepts the `request_type` or `view_request_type` attributes. These attributes didn't actually work in any useful way (see entry above this one).
- Because the `repoze.bfg` package now includes implementations of the adapter, subscriber and utility ZCML directives, it is now an error to have `<include package="repoze.zcml" file="meta.zcml"/>` in the ZCML of a `repoze.bfg` application. A ZCML conflict error will be raised if your ZCML does so. This shouldn't be an issue for "normal" installations; it has always been the responsibility of the `repoze.bfg.includes` ZCML to include this file in the past; it now just doesn't.
- The `repoze.bfg.testing.zcml_configure` API was removed. Use the `Configurator.load_zcml` API instead.

Deprecations

- The `repoze.bfg.router.make_app` function is now nominally deprecated. Its import and usage does not throw a warning, nor will it probably ever disappear. However, using a `repoze.bfg.configuration.Configurator` class is now the preferred way to generate a WSGI application.

Note that `make_app` calls `zope.component.getSiteManager.sethook(repoze.bfg.threadlocal.get_current_registry)` on the caller's behalf, hooking ZCA global API lookups, for backwards compatibility purposes. If you disuse `make_app`, your calling code will need to perform this call itself, at least if your application uses the ZCA global API (`getSiteManager`, `getAdapter`, etc).

Dependencies

- A dependency on the `martian` package has been removed (its functionality is replaced internally).
- A dependency on the `repoze.zcml` package has been removed (its functionality is replaced internally).

1.1.1 (2009-11-21)

Bug Fixes

- “Hybrid mode” applications (applications which explicitly used traversal *after* url dispatch via `<route>` paths containing the `*traverse` element) were broken in 1.1-final and all 1.1 alpha and beta releases. Views registered without a `route_name` route shadowed views registered with a `route_name` inappropriately.

1.1 (2009-11-15)

Internals

- Remove dead `IRouteRequirement` interface from `repoze.bfg.zcml` module.

Documentation

- Improve the “Extending an Existing Application” narrative chapter.
- Add more sections to the “Defending Design” chapter.

1.1b4 (2009-11-12)

Bug Fixes

- Use `alsoProvides` in the `urldispatch` module to attach an interface to the request rather than `directlyProvides` to avoid disturbing interfaces set in a `NewRequest` event handler.

Documentation

- Move 1.0.1 and previous changelog to HISTORY.txt.
- Add examples to `repoze.bfg.url.model_url` docstring.
- Add “Defending BFG Design” chapter to frontpage docs.

Templates

- Remove `ez_setup.py` and its import from all paster templates, samples, and tutorials for `distribute` compatibility. The documentation already explains how to install `virtualenv` (which will include some `setuptools` package), so these files, imports and usages were superfluous.

Deprecations

- The `options` kw arg to the `repoze.bfg.router.make_app` function is deprecated. In its place is the keyword argument `settings`. The `options` keyword continues to work, and a deprecation warning is not emitted when it is detected. However, the paster templates, code samples, and documentation now make reference to `settings` rather than `options`. This change/deprecation was mainly made for purposes of clarity and symmetry with the `get_settings()` API and discussions of “settings” in various places in the docs: we want to use the same name to refer to the same thing everywhere.

1.1b3 (2009-11-06)

Features

- `repoze.bfg.testing.registerRoutesMapper` testing facility added. This testing function registers a routes “mapper” object in the registry, for tests which require its presence. This function is documented in the `repoze.bfg.testing` API documentation.

Bug Fixes

- Compound statements that used an assignment entered into in an interactive IPython session invoked via `paster bfgshell` no longer fail to mutate the shell namespace correctly. For example, this set of statements used to fail:

```
In [2]: def bar(x): return x
...:
In [3]: list(bar(x) for x in 'abc')
Out[3]: NameError: 'bar'
```

In this release, the `bar` function is found and the correct output is now sent to the console. Thanks to Daniel Holth for the patch.

- The `bfgshell` command did not function properly; it was still expecting to be able to call the root factory with a bare `environ` rather than a request object.

Backwards Incompatibilities

- The `repoze.bfg.scripting.get_root` function now expects a request object as its second argument rather than an `environ`.

1.1b2 (2009-11-02)

Bug Fixes

- Prevent PyPI installation failure due to `easy_install` trying way too hard to guess the best version of Paste. When `easy_install` pulls from PyPI it reads links off various pages to determine “more up to date” versions. It incorrectly picks up a link for an ancient version of a package named “Paste-Deploy-0.1” (note the dash) when trying to find the “Paste” distribution and somehow believes it’s the latest version of “Paste”. It also somehow “helpfully” decides to check out a version of this package from SVN. We pin the Paste dependency version to a version greater than 1.7 to work around this `easy_install` bug.

Documentation

- Fix “Hybrid” narrative chapter: stop claiming that `<view>` statements that mention a `route_name` need to come after (in XML order) the `<route>` statement which creates the route. This hasn’t been true since 1.1a1.
- “What’s New in `repoze.bfg` 1.1” document added to narrative documentation.

Features

- Add a new event type: `repoze.bfg.events.AfterTraversal`. Events of this type will be sent after traversal is completed, but before any view code is invoked. Like `repoze.bfg.events.NewRequest`, This event will have a single attribute: `request` representing the current request. Unlike the request attribute of `repoze.bfg.events.NewRequest` however, during an `AfterTraversal` event, the request object will possess attributes set by the traverser, most notably `context`, which will be the context used when a view is found and invoked. The interface `repoze.bfg.events.IAfterTraversal` can be used to subscribe to the event. For example:

```
<subscriber for="repoze.bfg.interfaces.IAfterTraversal"
            handler="my.app.handle_after_traverse"/>
```

Like any framework event, a subscriber function should expect one parameter: `event`.

Dependencies

- Rather than depending on `chameleon.core` and `chameleon.zpt` distributions individually, depend on Malthe's repackaged Chameleon distribution (which includes both `chameleon.core` and `chameleon.zpt`).

1.1b1 (2009-11-01)

Bug Fixes

- The routes root factory called route factories and the default route factory with an `environ` rather than a `request`. One of the symptoms of this bug: applications generated using the `bfg_zodb` pasteur template in 1.1a9 did not work properly.
- Reinstate `renderer` alias for `view_renderer` in the `<route>` ZCML directive (in-the-wild 1.1a bw compat).
- `bfg_routesalchemy` pasteur template: change `<route>` declarations: rename `renderer` attribute to `view_renderer`.
- Header values returned by the `authktauthenticationpolicy` `remember` and `forget` methods would be of type `unicode`. This violated the WSGI spec, causing a `TypeError` to be raised when these headers were used under `mod_wsgi`.

- If a BFG app that had a route matching the root URL was mounted under a path in mod-wsgi, ala `WSGIScriptAlias /myapp /Users/chris/projects/modwsgi/env/bfg.wsgi`, the home route (a route with the path of `'/'` or `' '`) would not match when the path `/myapp` was visited (only when the path `/myapp/` was visited). This is now fixed: if the `urldispatch` root factory notes that the `PATH_INFO` is empty, it converts it to a single slash before trying to do matching.

Documentation

- In `<route>` declarations in tutorial ZCML, rename `renderer` attribute to `view_renderer` (fwd compat).
- Fix various tutorials broken by 1.1a9 `<route>` directive changes.

Internal

- Deal with a potential circref in the traversal module.

1.1a9 (2009-10-31)

Bug Fixes

- An incorrect ZCML conflict would be encountered when the `request_param` predicate attribute was used on the ZCML `view` directive if any two otherwise same-predicated views had the combination of a predicate value with an `=` sign and one without (e.g. `a` vs. `a=123`).

Features

- In previous versions of BFG, the “root factory” (the `get_root` callable passed to `make_app` or a function pointed to by the `factory` attribute of a route) was called with a “bare” WSGI environment. In this version, and going forward, it will be called with a `request` object. The request object passed to the factory implements dictionary-like methods in such a way that existing root factory code which expects to be passed an `environ` will continue to work.

- The `__call__` of a plugin “traverser” implementation (registered as an adapter for `ITraverser` or `ITraverserFactory`) will now receive a *request* as the single argument to its `__call__` method. In previous versions it was passed a `WSGI environ` object. The request object passed to the factory implements dictionary-like methods in such a way that existing traverser code which expects to be passed an `environ` will continue to work.
- The `ZCML route` directive’s attributes `xhr`, `request_method`, `path_info`, `request_param`, `header` and `accept` are now *route* predicates rather than *view* predicates. If one or more of these predicates is specified in the route configuration, all of the predicates must return true for the route to match a request. If one or more of the route predicates associated with a route returns `False` when checked during a request, the route match fails, and the next match in the routelist is tried. This differs from the previous behavior, where no route predicates existed and all predicates were considered view predicates, because in that scenario, the next route was not tried.

Documentation

- Various changes were made to narrative and API documentation supporting the change from passing a request rather than an `environ` to root factories and traversers.

Internal

- The request implements dictionary-like methods that mutate and query the `WSGI environ`. This is only for the purpose of backwards compatibility with root factories which expect an `environ` rather than a request.
- The `repoze.bfg.request.create_route_request_factory` function, which returned a request factory was removed in favor of a `repoze.bfg.request.route_request_interface` function, which returns an interface.
- The `repoze.bfg.request.Request` class, which is a subclass of `webob.Request` now defines its own `__setattr__`, `__getattr__` and `__delattr__` methods, which override the default `WebOb` behavior. The default `WebOb` behavior stores attributes of the request in `self.environ['webob.adhoc_attrs']`, and retrieves them from that dictionary during a `__getattr__`. This behavior was undesirable for speed and “expectation” reasons. Now attributes of the request are stored in `request.__dict__` (as you otherwise might expect from an object that did not override these methods).
- The router no longer calls `repoze.bfg.traversal._traverse` and does its work “inline” (speed).

- Reverse the order in which the router calls the request factory and the root factory. The request factory is now called first; the resulting request is passed to the root factory.
- The `repoze.bfg.request.request_factory` function has been removed. Its functionality is no longer required.
- The “routes root factory” that wraps the default root factory when there are routes mentioned in the configuration now attaches an interface to the request via `zope.interface.directlyProvides`. This replaces logic in the (now-gone) `repoze.bfg.request.request_factory` function.
- The `route` and `view` ZCML directives now register an interface as a named utility (retrieved from `repoze.bfg.request.route_request_interface`) rather than a request factory (the previous return value of the now-missing `repoze.bfg.request.create_route_request_factory`).
- The `repoze.bfg.functional` module was renamed to `repoze.bfg.compat`.

Backwards Incompatibilities

- Explicitly revert the feature introduced in 1.1a8: where the name `root` is available as an attribute of the request before a `NewRequest` event is emitted. This makes some potential future features impossible, or at least awkward (such as grouping traversal and view lookup into a single adapter lookup).
- The `containment`, `attr` and `renderer` attributes of the `route` ZCML directive were removed.

1.1a8 (2009-10-27)

Features

- Add `path_info` view configuration predicate.
- `paster bfgshell` now supports IPython if it’s available for import. Thanks to Daniel Holth for the initial patch.
- Add `repoze.bfg.testing.registerSettings` API, which is documented in the “`repoze.bfg.testing`” API chapter. This allows for registration of “settings” values obtained via `repoze.bfg.settings.get_settings()` for use in unit tests.
- The name `root` is available as an attribute of the request slightly earlier now (before a `NewRequest` event is emitted). `root` is the result of the application “root factory”.
- Added `max_age` parameter to `authktauthenticationpolicy` ZCML directive. If this value is set, it must be an integer representing the number of seconds which the auth tkt cookie will survive. Mainly, its existence allows the `auth_tkt` cookie to survive across browser sessions.

Bug Fixes

- Fix bug encountered during “scan” (when `<scan .>` directive is used in ZCML) introduced in 1.1a7. Symptom: `AttributeError: object has no attribute __provides__` raised at startup time.
- The `reissue_time` argument to the `authktauthenticationpolicy` ZCML directive now actually works. When it is set to an integer value, an `authticket` set-cookie header is appended to the response whenever a request requires authentication and ‘now’ minus the `authticket`’s timestamp is greater than `reissue_time` seconds.

Documentation

- Add a chapter titled “Request and Response” to the narrative documentation, content cribbed from the WebOb documentation.
- Call out predicate attributes of ZCML directive within “Views” chapter.
- Fix `route_url` documentation (`_query` argument documented as `query` and `_anchor` argument documented as `anchor`).

Backwards Incompatibilities

- The `authkt` authentication policy `remember` method now no longer honors `token` or `userdata` keyword arguments.

Internal

- Change how `bfg_view` decorator works when used as a class method decorator. In 1.1a7, the “scan” directive actually tried to grope every class in scanned package at startup time, calling `dir` against each found class, and subsequently invoking `getattr` against each thing found by `dir` to see if it was a method. This led to some strange symptoms (e.g. `AttributeError: object has no attribute __provides__`), and was generally just a bad idea. Now, instead of groping classes for methods at startup time, we just cause the `bfg_view` decorator itself to populate the method’s class’ `__dict__` when it is used as a method decorator. This also requires a nasty `_getframe` thing but it’s slightly less nasty than the startup time groping behavior. This is essentially a reversion back to 1.1a6 “grokking” behavior plus some special magic for using the `bfg_view` decorator as method decorator inside the `bfg_view` class itself.

- The router now checks for a `global_response_headers` attribute of the request object before returning a response. If this value exists, it is presumed to be a sequence of two-tuples, representing a set of headers to append to the ‘normal’ response headers. This feature is internal, rather than exposed externally, because it’s unclear whether it will stay around in the long term. It was added to support the `reissue_time` feature of the `authkt` authentication policy.
- The interface `ITraverserFactory` is now just an alias for `ITraverser`.

1.1a7 (2009-10-18)

Features

- More than one `@bfg_view` decorator may now be stacked on top of any number of others. Each invocation of the decorator registers a single view configuration. For instance, the following combination of decorators and a function will register two view configurations for the same view callable:

```
from repoze.bfg.view import bfg_view

@bfg_view(name='edit')
@bfg_view(name='change')
def edit(context, request):
    pass
```

This makes it possible to associate more than one view configuration with a single callable without requiring any ZCML.

- The `@bfg_view` decorator can now be used against a class method:

```
from webob import Response
from repoze.bfg.view import bfg_view

class MyView(object):
    def __init__(self, context, request):
        self.context = context
        self.request = request

    @bfg_view(name='hello')
    def amethod(self):
        return Response('hello from %s!' % self.context)
```

When the `bfg_view` decorator is used against a class method, a view is registered for the *class* (it’s a “class view” where the “attr” happens to be the name of the method it is attached to), so the class it’s defined within must have a suitable constructor: one that accepts `context`, `request` or just `request`.

Documentation

- Added `Changing the Traverser` and `Changing How :mod:`repoze.bfg.url.model_url` Generates a URL` to the “Hooks” narrative chapter of the docs.

Internal

- Remove `ez_setup.py` and imports of it within `setup.py`. In the new world, and as per virtualenv setup instructions, people will already have either `setuptools` or `distribute`.

1.1a6 (2009-10-15)

Features

- Add `xhr`, `accept`, and `header` view configuration predicates to ZCML view declaration, ZCML route declaration, and `bfg_view` decorator. See the `Views` narrative documentation chapter for more information about these predicates.
- Add `setUp` and `tearDown` functions to the `repoze.bfg.testing` module. Using `setUp` in a test setup and `tearDown` in a test teardown is now the recommended way to do component registry setup and teardown. Previously, it was recommended that a single function named `repoze.bfg.testing.cleanUp` be called in both the test setup and tear down. `repoze.bfg.testing.cleanUp` still exists (and will exist “forever” due to its widespread use); it is now just an alias for `repoze.bfg.testing.setUp` and is nominally deprecated.
- The BFG component registry is now available in view and event subscriber code as an attribute of the request ie. `request.registry`. This fact is currently undocumented except for this note, because BFG developers never need to interact with the registry directly anywhere else.
- The BFG component registry now inherits from `dict`, meaning that it can optionally be used as a simple dictionary. *Component* registrations performed against it via e.g. `registerUtility`, `registerAdapter`, and similar API methods are kept in a completely separate namespace than its `dict` members, so using the its component API methods won’t effect the keys and values in the dictionary namespace. Likewise, though the component registry “happens to be” a dictionary, use of mutating dictionary methods such as `__setitem__` will have no influence on any component registrations made against it. In other words, the registry object you obtain via e.g. `repoze.bfg.threadlocal.get_current_registry` or `request.registry` happens to be both a component registry and a dictionary, but using its component-registry API won’t impact data added to it via its dictionary API and vice versa. This is a forward compatibility move based on the goals of “marco”.
- Expose and document `repoze.bfg.testing.zcml_configure` API. This function populates a component registry from a ZCML file for testing purposes. It is documented in the “Unit and Integration Testing” chapter.

Documentation

- Virtual hosting narrative docs chapter updated with info about `mod_wsgi`.
- Point all index URLs at the literal 1.1 index (this alpha cycle may go on a while).
- Various tutorial test modules updated to use `repoze.bfg.testing.setUp` and `repoze.bfg.testing.tearDown` methods in order to encourage this as best practice going forward.
- Added “Creating Integration Tests” section to unit testing narrative documentation chapter. As a result, the name of the unittesting chapter is now “Unit and Integration Testing”.

Backwards Incompatibilities

- Importing `getSiteManager` and `get_registry` from `repoze.bfg.registry` is no longer supported. These imports were deprecated in `repoze.bfg 1.0`. Import of `getSiteManager` should be done as `from zope.component import getSiteManager`. Import of `get_registry` should be done as `from repoze.bfg.threadlocal import get_current_registry`. This was done to prevent a circular import dependency.
- Code bases which alternately invoke both `zope.testing.cleanup.cleanUp` and `repoze.bfg.testing.cleanUp` (treating them equivalently, using them interchangeably) in the `setUp/tearDown` of unit tests will begin to experience test failures due to lack of test isolation. The “right” mechanism is `repoze.bfg.testing.cleanUp` (or the combination of `repoze.bfg.testing.setUp` and `repoze.bfg.testing.tearDown`). but a good number of legacy codebases will use `zope.testing.cleanup.cleanUp` instead. We support `zope.testing.cleanup.cleanUp` but not in combination with `repoze.bfg.testing.cleanUp` in the same codebase. You should use one or the other test cleanup function in a single codebase, but not both.

Internal

- Created new `repoze.bfg.configuration` module which assumes responsibilities previously held by the `repoze.bfg.registry` and `repoze.bfg.router` modules (avoid a circular import dependency).
- The result of the `zope.component.getSiteManager` function in unit tests set up with `repoze.bfg.testing.cleanUp` or `repoze.bfg.testing.setUp` will be an instance of `repoze.bfg.registry.Registry` instead of the global `zope.component.globalregistry.base.registry`. This also means that the threadlocal ZCA API functions such as `getAdapter` and `getUtility` as well as internal BFG machinery (such as `model_url` and `route_url`) will consult this registry within unit tests. This is a forward compatibility move based on the goals of “marco”.
- Removed `repoze.bfg.testing.addCleanUp` function and associated module-scope globals. This was never an API.

1.1a5 (2009-10-10)

Documentation

- Change “Traversal + ZODB” and “URL Dispatch + SQLAlchemy” Wiki tutorials to make use of the new-to-1.1 “renderer” feature (return dictionaries from all views).
- Add tests to the “URL Dispatch + SQLAlchemy” tutorial after the “view” step.
- Added a diagram of model graph traversal to the “Traversal” narrative chapter of the documentation.
- An `exceptions` API chapter was added, documenting the new `repoze.bfg.exceptions` module.
- Describe “request-only” view calling conventions inside the `urldispatch` narrative chapter, where it’s most helpful.
- Add a diagram which explains the operation of the BFG router to the “Router” narrative chapter.

Features

- Add a new `repoze.bfg.testing` API: `registerRoute`, for registering routes to satisfy calls to e.g. `repoze.bfg.url.route_url` in unit tests.
- The `notfound` and `forbidden` ZCML directives now accept the following additional attributes: `attr`, `renderer`, and `wrapper`. These have the same meaning as they do in the context of a ZCML `view` directive.
- For behavior like Django’s `APPEND_SLASH=True`, use the `repoze.bfg.view.append_slash_notfound_view` view as the Not Found view in your application. When this view is the Not Found view (indicating that no view was found), and any routes have been defined in the configuration of your application, if the value of `PATH_INFO` does not already end in a slash, and if the value of `PATH_INFO` *plus* a slash matches any route’s path, do an HTTP redirect to the slash-appended `PATH_INFO`. Note that this will *lose* POST data information (turning it into a GET), so you shouldn’t rely on this to redirect POST requests.
- Speed up `repoze.bfg.location.lineage` slightly.
- Speed up `repoze.bfg.encode.urlencode` (nee’ `repoze.bfg.url.urlencode`) slightly.

- Speed up `repoze.bfg.traversal.model_path`.
- Speed up `repoze.bfg.traversal.model_path_tuple` slightly.
- Speed up `repoze.bfg.traversal.traverse` slightly.
- Speed up `repoze.bfg.url.model_url` slightly.
- Speed up `repoze.bfg.url.route_url` slightly.
- Speed up `repoze.bfg.traversal.ModelGraphTraverser.__call__` slightly.
- Minor speedup of `repoze.bfg.router.Router.__call__`.
- New `repoze.bfg.exceptions` module was created to house exceptions that were previously sprinkled through various modules.

Internal

- Move `repoze.bfg.traversal._url_quote` into `repoze.bfg.encode` as `url_quote`.

Deprecations

- The import of `repoze.bfg.view.NotFound` is deprecated in favor of `repoze.bfg.exceptions.NotFound`. The old location still functions, but emits a deprecation warning.
- The import of `repoze.bfg.security.Unauthorized` is deprecated in favor of `repoze.bfg.exceptions.Forbidden`. The old location still functions but emits a deprecation warning. The rename from `Unauthorized` to `Forbidden` brings parity to the name of the exception and the system view it invokes when raised.

Backwards Incompatibilities

- We previously had a Unicode-aware wrapper for the `urllib.urlencode` function named `repoze.bfg.url.urlencode` which delegated to the `stdlib` function, but which marshalled all unicode values to utf-8 strings before calling the `stdlib` version. A newer replacement now lives in `repoze.bfg.encode`. The replacement does not delegate to the `stdlib`.

The replacement diverges from the `stdlib` implementation and the previous `repoze.bfg.url.urlencode` implementation inasmuch as its `doseq` argument is now a decoy: it always behaves in the `doseq=True` way (which is the only sane behavior) for speed purposes.

The old import location (`repoze.bfg.url.urlencode`) still functions and has not been deprecated.

- In 0.8a7, the return value expected from an object implementing `ITraverserFactory` was changed from a sequence of values to a dictionary containing the keys `context`, `view_name`, `subpath`, `traversed`, `virtual_root`, `virtual_root_path`, and `root`. Until now, old-style traversers which returned a sequence have continued to work but have generated a deprecation warning. In this release, traversers which return a sequence instead of a dictionary will no longer work.

1.1a4 (2009-09-23)

Bug Fixes

- On 64-bit Linux systems, views that were members of a multiview (orderings of views with predicates) were not evaluated in the proper order. Symptom: in a configuration that had two views with the same name but one with a `request_method=POST` predicate and one without, the one without the predicate would be called unconditionally (even if the request was a POST request). Thanks much to Sebastien Douche for providing the buildbots that pointed this out.

Documentation

- Added a tutorial which explains how to use `repoze.session` (ZODB-based sessions) in a ZODB-based `repoze.bfg` app.
- Added a tutorial which explains how to add ZEO to a ZODB-based `repoze.bfg` application.
- Added a tutorial which explains how to run a `repoze.bfg` application under `mod_wsgi`. See “Running a `repoze.bfg` Application under `mod_wsgi`” in the tutorials section of the documentation.

Features

- Add a `repoze.bfg.url.static_url` API which is capable of generating URLs to static resources defined by the `<static>` ZCML directive. See the “Views” narrative chapter’s section titled “Generating Static Resource URLs” for more information.
- Add a `string` renderer. This renderer converts a non-Response return value of any view callable into a string. It is documented in the “Views” narrative chapter.
- Give the `route` ZCML directive the `view_attr` and `view_renderer` parameters (bring up to speed with 1.1a3 features). These can also be spelled as `attr` and `renderer`.

Backwards Incompatibilities

- An object implementing the `IRenderer` interface (and `ITemplateRenderer``, which is a subclass of `IRenderer`) must now accept an extra `system` argument in its `__call__` method implementation. Values computed by the system (as opposed to by the view) are passed by the system in the `system` parameter, which will always be a dictionary. Keys in the dictionary include: `view` (the view object that returned the value), `renderer_name` (the template name or simple name of the renderer), `context` (the context object passed to the view), and `request` (the request object passed to the view). Previously only `ITemplateRenderers` received system arguments as elements inside the main `value` dictionary.

Internal

- The way `bfg_view` declarations are scanned for has been modified. This should have no external effects.
- Speed: do not register an `ITraverserFactory` in `configure.zcml`; instead rely on `queryAdapter` and a manual default to `ModelGraphTraverser`.
- Speed: do not register an `IContextURL` in `configure.zcml`; instead rely on `queryAdapter` and a manual default to `TraversalContextURL`.
- General speed microimprovements for helloworld benchmark: replace `try/excepts` with statements which use `‘in’` keyword.

1.1a3 (2009-09-16)

Documentation

- The “Views” narrative chapter in the documentation has been updated extensively to discuss “renderers”.

Features

- A `renderer` attribute has been added to view configurations, replacing the previous (1.1a2) version’s `template` attribute. A “renderer” is an object which accepts the return value of a view and converts it to a string. This includes, but is not limited to, templating systems.
- A new interface named `IRenderer` was added. The existing interface, `ITemplateRenderer` now derives from this new interface. This interface is internal.
- A new interface named `IRendererFactory` was added. An existing interface named `ITemplateRendererFactory` now derives from this interface. This interface is internal.
- The `view` attribute of the `view` ZCML directive is no longer required if the ZCML directive also has a `renderer` attribute. This is useful when the renderer is a template renderer and no names need be passed to the template at render time.
- A new zcml directive `renderer` has been added. It is documented in the “Views” narrative chapter of the documentation.
- A ZCML `view` directive (and the associated `bfg_view` decorator) can now accept a “wrapper” value. If a “wrapper” value is supplied, it is the value of a separate view’s `name` attribute. When a view with a `wrapper` attribute is rendered, the “inner” view is first rendered normally. Its body is then attached to the request as “wrapped_body”, and then a wrapper view name is looked up and rendered (using `repoze.bfg.render_view_to_response`), passed the request and the context. The wrapper view is assumed to do something sensible with `request.wrapped_body`, usually inserting its structure into some other rendered template. This feature makes it possible to specify (potentially nested) “owrap” relationships between views using only ZCML or decorators (as opposed always using ZPT METAL and analogues to wrap view renderings in outer wrappers).

Dependencies

- When used under Python < 2.6, BFG now has an installation time dependency on the `simplejson` package.

Deprecations

- The `repoze.bfg.testing.registerDummyRenderer` API has been deprecated in favor of `repoze.bfg.testing.registerTemplateRenderer`. A deprecation warning is *not* issued at import time for the former name; it will exist “forever”; its existence has been removed from the documentation, however.
- The `repoze.bfg.templating.renderer_from_cache` function has been moved to `repoze.bfg.renderer.template_renderer_factory`. This was never an API, but code in the wild was spotted that used it. A deprecation warning is issued at import time for the former.

Backwards Incompatibilities

- The `ITemplateRenderer` interface has been changed. Previously its `__call__` method accepted `**kw`. It now accepts a single positional parameter named `kw` (REVISED: it accepts two positional parameters as of 1.1a4: `value` and `system`). This is mostly an internal change, but it was exposed in APIs in one place: if you’ve used the `repoze.bfg.testing.registerDummyRenderer` API in your tests with a custom “renderer” argument with your own renderer implementation, you will need to change that renderer implementation to accept `kw` instead of `**kw` in its `__call__` method (REVISED: make it accept `value` and `system` positional arguments as of 1.1a4).
- The `ITemplateRendererFactory` interface has been changed. Previously its `__call__` method accepted an `auto_reload` keyword parameter. Now its `__call__` method accepts no keyword parameters. Renderers are now themselves responsible for determining details of auto-reload. This is purely an internal change. This interface was never external.
- The `template_renderer` ZCML directive introduced in 1.1a2 has been removed. It has been replaced by the `renderer` directive.
- The previous release (1.1a2) added a view configuration attribute named `template`. In this release, the attribute has been renamed to `renderer`. This signifies that the attribute is more generic: it can now be not just a template name but any renderer name (ala `json`).
- In the previous release (1.1a2), the Chameleon text template renderer was used if the system didn’t associate the `template` view configuration value with a filename with a “known” extension. In this release, you must use a `renderer` attribute which is a path that ends with a `.txt` extension (e.g. `templates/foo.txt`) to use the Chameleon text renderer.

1.1a2 (2009-09-14)

Features

- A ZCML `view` directive (and the associated `bfg_view` decorator) can now accept an “`attr`” value. If an “`attr`” value is supplied, it is considered a method named of the view object to be called when the response is required. This is typically only good for views that are classes or instances (not so useful for functions, as functions typically have no methods other than `__call__`).
- A ZCML `view` directive (and the associated `bfg_view` decorator) can now accept a “`template`” value. If a “`template`” value is supplied, and the view callable returns a dictionary, the associated template is rendered with the dictionary as keyword arguments. See the section named “Views That Have a `template`” in the “Views” narrative documentation chapter for more information.

1.1a1 (2009-09-06)

Bug Fixes

- “`tests`” module removed from the `bfg_alchemy` pasteur template; these tests didn’t work.
- Bugfix: the `discriminator` for the ZCML “`route`” directive was incorrect. It was possible to register two routes that collided without the system spitting out a `ConfigurationConflictError` at startup time.

Features

- Feature addition: view predicates. These are exposed as the `request_method`, `request_param`, and `containment` attributes of a ZCML `view` declaration, or the respective arguments to a `@bfg_view` decorator. View predicates can be used to register a view for a more precise set of environment parameters than was previously possible. For example, you can register two views with the same name with different `request_param` attributes. If the `request.params` dict contains ‘`foo`’ (`request_param=“foo”`), one view might be called; if it contains ‘`bar`’ (`request_param=“bar”`), another view might be called. `request_param` can also name a key/value pair ala `foo=123`. This will match only when the `foo` key is in the `request.params` dict and it has the value ‘`123`’. This particular example makes it possible to write separate view functions for different form submissions. The other predicates, `containment` and `request_method` work similarly. `containment` is a view predicate that will match only when the context’s graph lineage has an object possessing a particular class or interface, for example. `request_method` is a view predicate that will match when the HTTP `REQUEST_METHOD` equals some string (eg. ‘`POST`’).

- The `@bfg_view` decorator now accepts three additional arguments: `request_method`, `request_param`, and `containment`. `request_method` is used when you'd like the view to match only a request with a particular HTTP `REQUEST_METHOD`; a string naming the `REQUEST_METHOD` can also be supplied as `request_type` for backwards compatibility. `request_param` is used when you'd like a view to match only a request that contains a particular `request.params` key (with or without a value). `containment` is used when you'd like to match a request that has a context that has some class or interface in its graph lineage. These are collectively known as “view predicates”.
- The route ZCML directive now honors `view_request_method`, `view_request_param` and `view_containment` attributes, which pass along these values to the associated view if any is provided. Additionally, the `request_type` attribute can now be spelled as `view_request_type`, and `permission` can be spelled as `view_permission`. Any attribute which starts with `view_` can now be spelled without the `view_` prefix, so `view_for` can be spelled as `for` now, etc. Both forms are documented in the `urldispatch` narrative documentation chapter.
- The `request_param` ZCML view directive attribute (and its `bfg_view` decorator cousin) can now specify both a key and a value. For example, `request_param="foo=123"` means that the `foo` key must have a value of 123 for the view to “match”.
- Allow `repoze.bfg.traversal.find_interface` API to use a class object as the argument to compare against the model passed in. This means you can now do `find_interface(model, SomeClass)` and the first object which is found in the lineage which has `SomeClass` as its class (or the first object found which has `SomeClass` as any of its superclasses) will be returned.
- Added `static` ZCML directive which registers a route for a view that serves up files in a directory. See the “Views” narrative documentation chapter’s “Serving Static Resources Using a ZCML Directive” section for more information.
- The `repoze.bfg.view.static` class now accepts a string as its first argument (“`root_dir`”) that represents a package-relative name e.g. `somepackage:foo/bar/static`. This is now the preferred mechanism for spelling package-relative static paths using this class. A `package_name` keyword argument has been left around for backwards compatibility. If it is supplied, it will be honored.
- The API `repoze.bfg.testing.registerView` now takes a `permission` argument. Use this instead of using `repoze.bfg.testing.registerViewPermission`.
- The ordering of route declarations vs. the ordering of view declarations that use a “`route_name`” in ZCML no longer matters. Previously it had been impossible to use a `route_name` from a route that had not yet been defined in ZCML (order-wise) within a “view” declaration.
- The `repoze.bfg` router now catches both `repoze.bfg.security.Unauthorized` and `repoze.bfg.view.NotFound` exceptions while rendering a view. When the router catches an `Unauthorized`, it returns the registered forbidden view. When the router catches a `NotFound`, it returns the registered notfound view.

Internal

- Change `urldispatch` internals: `Route` object is now constructed using a path, a name, and a factory instead of a name, a matcher, a generator, and a factory.
- Move (non-API) `default_view`, `default_forbidden_view`, and `default_notfound_view` functions into the `repoze.bfg.view` module (moved from `repoze.bfg.router`).
- Removed `ViewPermissionFactory` from `repoze.bfg.security`. View permission checking is now done by registering and looking up an `ISecuredView`.
- The `static` `ZCML` directive now uses a custom root factory when constructing a route.
- The interface `IRequestFactories` was removed from the `repoze.bfg.interfaces` module. This interface was never an API.
- The function named `named_request_factories` and the data structure named `DEFAULT_REQUEST_FACTORIES` have been removed from the `repoze.bfg.request` module. These were never APIs.
- The `IViewPermissionFactory` interface has been removed. This was never an API.

Documentation

- Request-only-convention examples in the “Views” narrative documentation were broken.
- Fixed documentation bugs related to `forget` and `remember` in security API docs.
- Fixed documentation for `repoze.bfg.view.static` (in narrative `Views` chapter).

Deprecations

- The API `repoze.bfg.testing.registerViewPermission` has been deprecated.

Backwards Incompatibilities

- The interfaces `IPOSTRequest`, `IGETRequest`, `IPUTRequest`, `IDeleteRequest`, and `IHEADRequest` have been removed from the `repoze.bfg.interfaces` module. These were not documented as APIs post-1.0. Instead of using one of these, use a `request_method` ZCML attribute or `request_method bfg_view` decorator parameter containing an HTTP method name (one of `GET`, `POST`, `HEAD`, `PUT`, `DELETE`) instead of one of these interfaces if you were using one explicitly. Passing a string in the set (`GET`, `HEAD`, `PUT`, `POST`, `DELETE`) as a `request_type` argument will work too. Rationale: instead of relying on interfaces attached to the request object, BFG now uses a “view predicate” to determine the request type.
- Views registered without the help of the ZCML `view` directive are now responsible for performing their own authorization checking.
- The `registry_manager` backwards compatibility alias importable from “`repoze.bfg.registry`”, deprecated since `repoze.bfg` 0.9 has been removed. If you are trying to use the registry manager within a debug script of your own, use a combination of the “`repoze.bfg.paster.get_app`” and “`repoze.bfg.scripts.get_root`” APIs instead.
- The `INotFoundAppFactory` interface has been removed; it has been deprecated since `repoze.bfg` 0.9. If you have something like the following in your `configure.zcml`:

```
<utility provides="repoze.bfg.interfaces.INotFoundAppFactory"
              component="helloworld.factories.notfound_app_factory"/>
```

Replace it with something like:

```
<notfound
    view="helloworld.views.notfound_view"/>
```

See “Changing the Not Found View” in the “Hooks” chapter of the documentation for more information.

- The `IUnauthorizedAppFactory` interface has been removed; it has been deprecated since `repoze.bfg` 0.9. If you have something like the following in your `configure.zcml`:

```
<utility provides="repoze.bfg.interfaces.IUnauthorizedAppFactory"
              component="helloworld.factories.unauthorized_app_factory"/>
```

Replace it with something like:

```
<forbidden
  view="helloworld.views.forbidden_view"/>
```

See “Changing the Forbidden View” in the “Hooks” chapter of the documentation for more information.

- `ISecurityPolicy`-based security policies, deprecated since `repoze.bfg` 0.9, have been removed. If you have something like this in your `configure.zcml`, it will no longer work:

```
<utility
  provides="repoze.bfg.interfaces.ISecurityPolicy"
  factory="repoze.bfg.security.RemoteUserInheritingACLSecurityPolicy"
/>
```

If ZCML like the above exists in your application, you will receive an error at startup time. Instead of the above, you’ll need something like:

```
<remoteuserauthenticationpolicy/>
<aclauthorizationpolicy/>
```

This is just an example. See the “Security” chapter of the `repoze.bfg` documentation for more information about configuring security policies.

- Custom ZCML directives which register an authentication or authorization policy (ala “`authktauthenticationpolicy`” or “`aclauthorizationpolicy`”) should register the policy “eagerly” in the ZCML directive instead of from within a ZCML action. If an authentication or authorization policy is not found in the component registry by the view machinery during deferred ZCML processing, view security will not work as expected.

1.0.1 (2009-07-22)

- Added support for `has_resource`, `resource_isdir`, and `resource_listdir` to the resource “`OverrideProvider`”; this fixes a bug with a symptom that a file could not be overridden in a resource directory unless a file with the same name existed in the original directory being overridden.
- Fixed documentation bug showing invalid test for values from the `matchdict`: they are stored as attributes of the `Article`, rather than subitems.

- Fixed documentation bug showing wrong environment key for the `matchdict` produced by the matching route.
- Added a workaround for a bug in Python 2.6, 2.6.1, and 2.6.2 having to do with a recursion error in the `mimetypes` module when trying to serve static files from Paste's `FileApp`: <http://bugs.python.org/issue5853>. Symptom: File `"/usr/lib/python2.6/mimetypes.py"`, line 244, in `guess_type` return `guess_type(url, strict)` `RuntimeError: maximum recursion depth exceeded`. Thanks to Armin Ronacher for identifying the symptom and pointing out a fix.
- Minor edits to tutorials for accuracy based on feedback.
- Declared Paste and PasteDeploy dependencies.

1.0 (2009-07-05)

- Retested and added some content to GAE tutorial.
- Edited "Extending" narrative docs chapter.
- Added "Deleting the Database" section to the "Defining Models" chapter of the traversal wiki tutorial.
- Spell checking of narratives and tutorials.

1.0b2 (2009-07-03)

- `remoteuserauthenticationpolicy` ZCML directive didn't work without an `environ_key` directive (didn't match docs).
- Fix `configure_zcml` filespec check on Windows. Previously if an absolute filesystem path including a drive letter was passed as `filename` (or as `configure_zcml` in the options dict) to `repoze.bfg.router.make_app`, it would be treated as a `package:resource_name` specification.
- Fix inaccuracies and import errors in `bfgwiki` (traversal+ZODB) and `bfgwiki2` (urldispatch+SA) tutorials.
- Use `bfgsite` index for all tutorial `setup.cfg` files.
- Full documentation grammar/style/spelling audit.

1.0b1 (2009-07-02)

Features

- Allow a Paste config file (`configure_zcml`) value or an environment variable (`BFG_CONFIGURE_ZCML`) to name a ZCML file (optionally package-relative) that will be used to bootstrap the application. Previously, the integrator could not influence which ZCML file was used to do the bootstrapping (only the original application developer could do so).

Documentation

- Added a “Resources” chapter to the narrative documentation which explains how to override resources within one package from another package.
- Added an “Extending” chapter to the narrative documentation which explains how to extend or modify an existing BFG application using another Python package and ZCML.

1.0a9 (2009-07-01)

Features

- Make it possible to pass strings in the form “package_name:relative/path” to APIs like `render_template`, `render_template_to_response`, and `get_template`. Sometimes the package in which a caller lives is a direct namespace package, so the module which is returned is semi-useless for navigating from. In this way, the caller can control the horizontal and vertical of where things get looked up from.

1.0a8 (2009-07-01)

Deprecations

- Deprecate the `authentication_policy` and `authorization_policy` arguments to `repoze.bfg.router.make_app`. Instead, developers should use the various authentication policy ZCML directives (`repozewholauthenticationpolicy`, `remoteuserauthenticationpolicy` and `authktauthenticationpolicy`) and the `aclauthorizationpolicy` authorization policy directive as described in the changes to the “Security” narrative documentation chapter and the wiki tutorials.

Features

- Add three new ZCML directives which configure authentication policies:
 - `repozewholauthenticationpolicy`
 - `remoteuserauthenticationpolicy`
 - `authtktauthenticationpolicy`
- Add a new ZCML directive which configures an ACL authorization policy named `aclauthorizationpolicy`.

Bug Fixes

- Bug fix: when a `repoze.bfg.resource.PackageOverrides` class was instantiated, and the package it was overriding already had a `__loader__` attribute, it would fail at startup time, even if the `__loader__` attribute was another `PackageOverrides` instance. We now replace any `__loader__` that is also a `PackageOverrides` instance. Symptom: `ConfigurationExecutionError: <type 'exceptions.TypeError': Package <module 'karl.views' from '/Users/chris/projects/osi/bfgenv/src/karl/karl/views/__init__.pyc'> already has a __loader__ (probably a module in a zipped egg).`

1.0a7 (2009-06-30)

Features

- Add a `reload_resources` configuration file setting (aka the `BFG_RELOAD_RESOURCES` environment variable). When this is set to true, the server never needs to be restarted when moving files between directory resource overrides (esp. for templates currently).
- Add a `reload_all` configuration file setting (aka the `BFG_RELOAD_ALL` environment variable) that implies both `reload_resources` and `reload_templates`.
- The static helper view class now uses a `PackageURLParser` in order to allow for the overriding of static resources (CSS / logo files, etc) using the `resource` ZCML directive. The `PackageURLParser` class was added to a (new) static module in BFG; it is a subclass of the `StaticURLParser` class in `paste.urlparser`.
- The `repoze.bfg.templating.renderer_from_cache` function now checks for the `reload_resources` setting; if it's true, it does not register a template renderer (it won't use the registry as a template renderer cache).

Documentation

- Add `pkg_resources` to the glossary.
- Update the “Environment” docs to note the existence of `reload_resources` and `reload_all`.
- Updated the `bfg_alchemy` paster template to include two views: the view on the root shows a list of links to records; the view on a record shows the details for that object.

Internal

- Use a colon instead of a tab as the separator between package name and relpath to form the “spec” when register a `ITemplateRenderer`.
- Register a `repoze.bfg.resource.OverrideProvider` as a `pkg_resources` provider only for modules which are known to have overrides, instead of globally, when a `<resource>` directive is used (performance).

1.0a6 (2009-06-29)

Bug Fixes

- Use `caller_package` function instead of `caller_module` function within templating to avoid needing to name the caller module in resource overrides (actually match docs).
- Make it possible to override templates stored directly in a module with templates in a subdirectory of the same module, stored directly within another module, or stored in a subdirectory of another module (actually match docs).

1.0a5 (2009-06-28)

Features

- A new ZCML directive exists named “resource”. This ZCML directive allows you to override Chameleon templates within a package (both directories full of templates and individual template files) with other templates in the same package or within another package. This allows you to “fake out” a view’s use of a template, causing it to retrieve a different template than the one actually named by a relative path to a call like `render_template_to_response('templates/mytemplate.pt')`. For example, you can override a template file by doing:

```
<resource
  to_override="some.package:templates/mytemplate.pt"
  override_with="another.package:othertemplates/anothertemplate.pt"
/>
```

The string passed to “to_override” and “override_with” is named a “specification”. The colon separator in a specification separates the package name from a package-relative directory name. The colon and the following relative path are optional. If they are not specified, the override attempts to resolve every lookup into a package from the directory of another package. For example:

```
<resource
  to_override="some.package"
  override_with="another.package"
/>
```

Individual subdirectories within a package can also be overridden:

```
<resource
  to_override="some.package:templates/"
  override_with="another.package:othertemplates/"
/>
```

If you wish to override a directory with another directory, you must make sure to attach the slash to the end of both the `to_override` specification and the `override_with` specification. If you fail to attach a slash to the end of a specification that points a directory, you will get unexpected results. You cannot override a directory specification with a file specification, and vice versa (a startup error will occur if you try).

You cannot override a resource with itself (a startup error will occur if you try).

Only individual *package* resources may be overridden. Overrides will not traverse through sub-packages within an overridden package. This means that if you want to override resources for both `some.package:templates`, and `some.package.views:templates`, you will need to register two overrides.

The package name in a specification may start with a dot, meaning that the package is relative to the package in which the ZCML file resides. For example:

```
<resource
  to_override=".subpackage:templates/"
  override_with="another.package:templates/"
/>
```

Overrides for the same `to_overrides` specification can be named multiple times within ZCML. Each `override_with` path will be consulted in the order defined within ZCML, forming an override search path.

Resource overrides can actually override resources other than templates. Any software which uses the `pkg_resources` `get_resource_filename`, `get_resource_stream` or `get_resource_string` APIs will obtain an overridden file when an override is used. However, the only built-in facility which uses the `pkg_resources` API within BFG is the templating stuff, so we only call out template overrides here.

- Use the `pkg_resources` API to locate template filenames instead of dead-reckoning using the `os.path` module.
- The `repoze.bfg.templating` module now uses `pkg_resources` to locate and register template files instead of using an absolute path name.

1.0a4 (2009-06-25)

Features

- Cause `:segment` matches in route paths to put a Unicode-decoded and URL-dequoted value in the `matchdict` for the value matched. Previously a non-decoded non-URL-dequoted string was placed in the `matchdict` as the value.
- Cause `*remainder` matches in route paths to put a *tuple* in the `matchdict` dictionary in order to be able to present Unicode-decoded and URL-dequoted values for the traversal path. Previously a non-decoded non-URL-dequoted string was placed in the `matchdict` as the value.
- Add optional `max_age` keyword value to the `remember` method of `repoze.bfg.authentication.AuthTktAuthenticationPolicy`; if this value is passed to `remember`, the generated cookie will have a corresponding Max-Age value.

Documentation

- Add information to the URL Dispatch narrative documentation about path pattern matching syntax.

Bug Fixes

- Make `route_url` URL-quote segment replacements during generation. Remainder segments are not quoted.

1.0a3 (2009-06-24)

Implementation Changes

- `repoze.bfg` no longer relies on the `Routes` package to interpret URL paths. All known existing path patterns will continue to work with the reimplemented logic, which lives in `repoze.bfg.urldispatch`. `<route>` ZCML directives which use certain attributes (uncommon ones) may not work (see “Backwards Incompatibilities” below).

Bug Fixes

- `model_url` when passed a request that was generated as a result of a route match would fail in a call to `route.generate`.
- BFG-on-GAE didn’t work due to a corner case bug in the fallback Python implementation of `threading.local` (symptom: “Initialization arguments are not supported”). Thanks to Michael Bernstein for the bug report.

Documentation

- Added a “corner case” explanation to the “Hybrid Apps” chapter explaining what to do when “the wrong” view is matched.
- Use `repoze.bfg.url.route_url` API in tutorials rather than `Routes url_for` API.

Features

- Added the `repoze.bfg.url.route_url` API. This API allows you to generate URLs based on `<route>` declarations. See the URL Dispatch narrative chapter and the “`repoze.bfg.url`” module API documentation for more information.

Backwards Incompatibilities

- As a result of disusing Routes, using the Routes `url_for` API inside a BFG application (as was suggested by previous iterations of tutorials) will no longer work. Use the `repoze.bfg.url.route_url` method instead.
- The following attributes on the `<route>` ZCML directive no longer work: `encoding`, `static`, `filter`, `condition_method`, `condition_subdomain`, `condition_function`, `explicit`, or `subdomains`. These were all Routes features.
- The `<route>` ZCML directive no longer supports the `<requirement>` subdirective. This was a Routes feature.

1.0a2 (2009-06-23)

Bug Fixes

- The `bfg_routesalchemy` pasteur template app tests failed due to a mismatch between test and view signatures.

Features

- Add a `view_for` attribute to the `route` ZCML directive. This attribute should refer to an interface or a class (ala the `for` attribute of the `view` ZCML directive).

Documentation

- Conditional documentation in installation section (“how to install a Python interpreter”).

Backwards Incompatibilities

- The `callback` argument of the `repoze.bfg.authentication` authentication policies named `RepozeWho1AuthenticationPolicy`, `RemoteUserAuthenticationPolicy`, and `AuthTktAuthenticationPolicy` now must accept two positional arguments: the original argument accepted by each (userid or identity) plus a second argument, which will be the current request. Apologies, this is required to service finding groups when there is no “global” database connection.

1.0a1 (2009-06-22)

Features

- A new ZCML directive was added named `notfound`. This ZCML directive can be used to name a view that should be invoked when the request can't otherwise be resolved to a view callable. For example:

```
<notfound
  view="helloworld.views.notfound_view"/>
```

- A new ZCML directive was added named `forbidden`. This ZCML directive can be used to name a view that should be invoked when a view callable for a request is found, but cannot be invoked due to an authorization failure. For example:

```
<forbidden
  view="helloworld.views.forbidden_view"/>
```

- Allow views to be *optionally* defined as callables that accept only a request object, instead of both a context and a request (which still works, and always will). The following types work as views in this style:

- functions that accept a single argument `request`, e.g.:

```
def aview(request):
    pass
```

- new and old-style classes that have an `__init__` method that accepts `self`, `request`, e.g.:

```
def View(object):
    __init__(self, request):
        pass
```

- Arbitrary callables that have a `__call__` method that accepts `self`, `request`, e.g.:

```
def AView(object):
    def __call__(self, request):
        pass
    view = AView()
```

This likely should have been the calling convention all along, as the request has `context` as an attribute already, and with views called as a result of URL dispatch, having the context in the arguments is not very useful. C’est la vie.

- Cache the absolute path in the caller’s package globals within `repoze.bfg.path` to get rid of repeated (expensive) calls to `os.path.abspath`.
- Add `reissue_time` and `timeout` parameters to `repoze.bfg.authentication.AuthTktAuthenticationPolicy` constructor. If these are passed, cookies will be reset every so often (cadged from the same change to `repoze.who` lately).
- The `matchdict` related to the matching of a Routes route is available on the request as the `matchdict` attribute: `request.matchdict`. If no route matched, this attribute will be `None`.
- Make 404 responses slightly cheaper by showing `environ["PATH_INFO"]` on the notfound result page rather than the fully computed URL.
- Move LRU cache implementation into a separate package (`repoze.lru`).
- The concepts of traversal and URL dispatch have been unified. It is now possible to use the same sort of factory as both a traversal “root factory” and what used to be referred to as a `urldispatch` “context factory”.
- When the root factory argument (as a first argument) passed to `repoze.bfg.router.make_app` is `None`, a *default* root factory is used. This is in support of using routes as “root finders”; it supplants the idea that there is a default `IRoutesContextFactory`.
- The `view` ZCML statement and the `repoze.bfg.view.bfg_view` decorator now accept an extra argument: `route_name`. If a `route_name` is specified, it must match the name of a previously defined `route` statement. When it is specified, the view will only be called when that route matches during a request.
- It is now possible to perform traversal *after* a route has matched. Use the pattern `*traverse` in a `<route>` `path` attribute within ZCML, and the path remainder which it matches will be used as a traversal path.
- When any route defined matches, the WSGI environment will now contain a key `bfg.routes.route` (the Route object which matched), and a key `bfg.routes.matchdict` (the result of calling `route.match`).

Deprecations

- Utility registrations against `repoze.bfg.interfaces.INotFoundView` and `repoze.bfg.interfaces.IForbiddenView` are now deprecated. Use the `notfound` and `forbidden` ZCML directives instead (see the “Hooks” chapter for more information). Such registrations will continue to work, but the `notfound` and `forbidden` directives do “extra work” to ensure that the callable named by the directive can be called by the router even if it’s a class or request-argument-only view.

Removals

- The `IRoutesContext`, `IRoutesContextFactory`, and `IContextNotFound` interfaces were removed from `repoze.bfg.interfaces`. These were never APIs.
- The `repoze.bfg.urldispatch.RoutesContextNotFound`, `repoze.bfg.urldispatch.RoutesModelTraverser` and `repoze.bfg.urldispatch.RoutesContextURL` classes were removed. These were also never APIs.

Backwards Incompatibilities

- Moved the `repoze.bfg.push` module, which implemented the `pushpage` decorator, into a separate distribution, `repoze.bfg.pushpage`. Applications which used this decorator should continue to work after adding that distribution to their installation requirements.
- Changing the default request factory via an `IRequestFactory` utility registration (as used to be documented in the “Hooks” chapter’s “Changing the request factory” section) is no longer supported. The dance to manufacture a request is complicated as a result of unifying traversal and url dispatch, making it highly unlikely for anyone to be able to override it properly. For those who just want to decorate or modify a request, use a `NewRequestEvent` subscriber (see the Events chapter in the documentation).
- The `repoze.bfg.IRequestFactory` interface was removed. See the bullet above for why.
- Routes “context factories” (spelled as the factory argument to a route statement in ZCML) must now expect the WSGI `environ` as a single argument rather than a set of keyword arguments. They can obtain the match dictionary by asking for `environ['bfg.routes.matchdict']`. This is the same set of keywords that used to be passed to `urldispatch` “context factories” in BFG 0.9 and below.

- Using the `@zope.component.adapter` decorator on a bfg view function no longer works. Use the `@repoze.bfg.view.bfg_view` decorator instead to mark a function (or a class) as a view.
- The name under which the matching route object is found in the environ was changed from `bfg.route` to `bfg.routes.route`.
- Finding the root is now done *before* manufacturing a request object (and sending a new request event) within the router (it used to be performed afterwards).
- Adding `*path_info` to a route no longer changes the `PATH_INFO` for a request that matches using URL dispatch. This feature was only there to service the `repoze.bfg.wsgi.wsgiapp2` decorator and it did it wrong; use `*subpath` instead now.
- The values of `subpath`, `traversed`, and `virtual_root_path` attached to the request object are always now tuples instead of lists (performance).

Bug Fixes

- The `bfg_alchemy` Paster template named “repoze.tm” in its pipeline rather than “repoze.tm2”, causing the startup to fail.
- Move BBB logic for registering an `IAAuthenticationPolicy/IForbiddenView/INotFoundView` based on older concepts from the router module’s `make_app` function into the `repoze.bfg.zcml.zcml_configure` callable, to service compatibility with scripts that use “zope.configuration.xmlconfig” (replace with `repoze.bfg.zml.zcml_configure` as necessary to get BBB logic)

Documentation

- Add interface docs related to how to create authentication policies and authorization policies to the “Security” narrative chapter.
- Added a (fairly sad) “Combining Traversal and URL Dispatch” chapter to the narrative documentation. This explains the usage of `*traverse` and `*subpath` in routes URL patters.
- A “router” chapter explaining the request/response lifecycle at a high level was added.
- Replaced all mentions and explanations of a routes “context factory” with equivalent explanations of a “root factory” (context factories have been disused).
- Updated Routes bfgwiki2 tutorial to reflect the fact that context factories are now no longer used.

0.9.1 (2009-06-02)

Features

- Add API named `repoze.bfg.settings.get_settings` which retrieves a derivation of values passed as the `options` value of `repoze.bfg.router.make_app`. This API should be preferred instead of using `getUtility(ISettings)`. I added a new `repoze.bfg.settings` API document as well.

Bug Fixes

- Restored missing entry point declaration for `bfg_alchemy` paster template, which was accidentally removed in 0.9.

Documentation

- Fix a reference to `wsgiapp` in the `wsgiapp2` API documentation within the `repoze.bfg.wsgi` module.

API Removals

- The `repoze.bfg.location.locate` API was removed: it didn't do enough to be very helpful and had a misleading name.

0.9 (2009-06-01)

Bug Fixes

- It was not possible to register a custom `IRoutesContextFactory` for use as a default context factory as documented in the “Hooks” chapter.

Features

- The `request_type` argument of ZCML view declarations and `bfg_view` decorators can now be one of the strings GET, POST, PUT, DELETE, or HEAD instead of a reference to the respective interface type imported from `repoze.bfg.interfaces`.
- The `route` ZCML directive now accepts `request_type` as an alias for its `condition_method` argument for symmetry with the `view` directive.
- The `bfg_routesalchemy` paster template now provides a unit test and actually uses the database during a view rendering.

Removals

- Remove `repoze.bfg.threadlocal.setManager`. It was only used in unit tests.
- Remove `repoze.bfg.wsgi.HTTPException`, `repoze.bfg.wsgi.NotFound`, and `repoze.bfg.wsgi.Unauthorized`. These classes were disused with the introduction of the `IUnauthorizedView` and `INotFoundView` machinery.

Documentation

- Add description to narrative templating chapter about how to use Chameleon text templates.
- Changed Views narrative chapter to use method strings rather than interface types, and moved advanced interface type usage to Events narrative chapter.
- Added a Routes+SQLAlchemy wiki tutorial.

0.9a8 (2009-05-31)

Features

- It is now possible to register a custom `repoze.bfg.interfaces.INotFoundView` for a given application. This feature replaces the `repoze.bfg.interfaces.INotFoundAppFactory` feature previously described in the Hooks chapter. The `INotFoundView` will be called when the framework detects that a view lookup done as a result of a request fails; it should accept a context object and a request object; it should return an `IResponse` object (a webob response, basically). See the Hooks narrative chapter of the BFG docs for more info.
- The error presented when a view invoked by the router returns a non-response object now includes the view's name for troubleshooting purposes.

Bug Fixes

- A “new response” event is emitted for forbidden and notfound views.

Deprecations

- The `repoze.bfg.interfaces.INotFoundAppFactory` interface has been deprecated in favor of using the new `repoze.bfg.interfaces.INotFoundView` mechanism.

Renames

- Renamed `repoze.bfg.interfaces.IForbiddenResponseFactory` to `repoze.bfg.interfaces.IForbiddenView`.

0.9a7 (2009-05-30)

Features

- Remove “context” argument from `effective_principals` and `authenticated_userid` function APIs in `repoze.bfg.security`, effectively a doing reversion to 0.8 and before behavior. Both functions now again accept only the `request` parameter.

0.9a6 (2009-05-29)

Documentation

- Changed “BFG Wiki” tutorial to use `AuthTktAuthenticationPolicy` rather than `repoze.who`.

Features

- Add an `AuthTktAuthenticationPolicy`. This policy retrieves credentials from an `auth_tkt` cookie managed by the application itself (instead of relying on an upstream data source for authentication data). See the Security API chapter of the documentation for more info.
- Allow `RemoteUserAuthenticationPolicy` and `RepozeWho1AuthenticationPolicy` to accept various constructor arguments. See the Security API chapter of the documentation for more info.

0.9a5 (2009-05-28)**Features**

- Add a `get_app` API functions to the `paster` module. This obtains a WSGI application from a config file given a config file name and a section name. See the `repoze.bfg.paster` API docs for more information.
- Add a new module named `scripting`. It contains a `get_root` API function, which, provided a Router instance, returns a traversal root object and a “closer”. See the `repoze.bfg.scripting` API docs for more info.

0.9a4 (2009-05-27)**Bug Fixes**

- Try checking for an “old style” security policy *after* we parse ZCML (thinko).

0.9a3 (2009-05-27)**Features**

- Allow `IAuthenticationPolicy` and `IAuthorizationPolicy` to be overridden via ZCML registrations (do ZCML parsing after registering these in `router.py`).

Documentation

- Added “BFG Wiki” tutorial to documentation; it describes step-by-step how to create a traversal-based ZODB application with authentication.

Deprecations

- Added deprecations for imports of `ACLSecurityPolicy`, `InheritingACLSecurityPolicy`, `RemoteUserACLSecurityPolicy`, `RemoteUserInheritingACLSecurityPolicy`, `WhoACLSecurityPolicy`, and `WhoInheritingACLSecurityPolicy` from the `repoze.bfg.security` module; for the meantime (for backwards compatibility purposes) these live in the `repoze.bfg.secpols` module. Note however, that the entire concept of a “security policy” is deprecated in BFG in favor of separate authentication and authorization policies, so any use of a security policy will generate additional deprecation warnings even if you do start using `repoze.bfg.secpols`. `repoze.bfg.secpols` will disappear in a future release of `repoze.bfg`.

Deprecated Import Alias Removals

- Remove `repoze.bfg.template` module. All imports from this package have been deprecated since 0.3.8. Instead, import `get_template`, `render_template`, and `render_template_to_response` from the `repoze.bfg.chameleon_zpt` module.
- Remove backwards compatibility import alias for `repoze.bfg.traversal.split_path` (deprecated since 0.6.5). This must now be imported as `repoze.bfg.traversal.traversal_path`.
- Remove backwards compatibility import alias for `repoze.bfg.urldispatch.RoutesContext` (deprecated since 0.6.5). This must now be imported as `repoze.bfg.urldispatch.DefaultRoutesContext`.
- Removed backwards compatibility import aliases for `repoze.bfg.router.get_options` and `repoze.bfg.router.Settings` (deprecated since 0.6.2). These both must now be imported from `repoze.bfg.settings`.
- Removed backwards compatibility import alias for `repoze.bfg.interfaces.IRootPolicy` (deprecated since 0.6.2). It must be imported as `repoze.bfg.interfaces.IRootFactory` now.
- Removed backwards compatibility import alias for `repoze.bfg.interfaces.ITemplate` (deprecated since 0.4.4). It must be imported as `repoze.bfg.interfaces.ITemplateRenderer` now.
- Removed backwards compatibility import alias for `repoze.bfg.interfaces.ITemplateFactory` (deprecated since 0.4.4). It must be imported as `repoze.bfg.interfaces.ITemplateRendererFactory` now.
- Removed backwards compatibility import alias for `repoze.bfg.chameleon_zpt.ZPTTemplateFactory` (deprecated since 0.4.4). This must be imported as `repoze.bfg.ZPTTemplateRenderer` now.

0.9a2 (2009-05-27)

Features

- A paster command has been added named “bfgshell”. This command can be used to get an interactive prompt with your BFG root object in the global namespace. E.g.:

```
bin/paster bfgshell /path/to/myapp.ini myapp
```

See the `Project` chapter in the BFG documentation for more information.

Deprecations

- The name `repoze.bfg.registry.registry_manager` was never an API, but scripts in the wild were using it to set up an environment for use under a debug shell. A backwards compatibility shim has been added for this purpose, but the feature is deprecated.

0.9a1 (2009-5-27)

Features

- New API functions named `forget` and `remember` are available in the `security` module. The `forget` function returns headers which will cause the currently authenticated user to be logged out when set in a response. The `remember` function (when passed the proper arguments) will return headers which will cause a principal to be “logged in” when set in a response. See the Security API chapter of the docs for more info.
- New keyword arguments to the `repoze.bfg.router.make_app` call have been added: `authentication_policy` and `authorization_policy`. These should, respectively, be an implementation of an authentication policy (an object implementing the `repoze.bfg.interfaces.IAuthenticationPolicy` interface) and an implementation of an authorization policy (an object implementing `repoze.bfg.interfaces.IAuthorizationPolicy`). Concrete implementations of authentication policies exist in `repoze.bfg.authentication`. Concrete implementations of authorization policies exist in `repoze.bfg.authorization`.

Both `authentication_policy` and `authorization_policy` default to `None`.

If `authentication_policy` is `None`, but `authorization_policy` is *not* `None`, then `authorization_policy` is ignored (the ability to do authorization depends on authentication).

If the `authentication_policy` argument is *not* `None`, and the `authorization_policy` argument *is* `None`, the authorization policy defaults to an authorization implementation that uses ACLs (`repoze.bfg.authorization.ACLAuthorizationPolicy`).

We no longer encourage configuration of “security policies” using ZCML, as previously we did for `ISecurityPolicy`. This is because it’s not uncommon to need to configure settings for concrete authorization or authentication policies using paste `.ini` parameters; the app entry point for your application is the natural place to do this.

- Two new abstractions have been added in the way of adapters used by the system: an `IAuthorizationPolicy` and an `IAuthenticationPolicy`. A combination of these (as registered by the `securitypolicy` ZCML directive) take the place of the `ISecurityPolicy` abstraction in previous releases of `repoze.who`. The API functions in `repoze.who.security` (such as `authentication_userid`, `effective_principals`, `has_permission`, and so on) have been changed to try to make use of these new adapters. If you’re using an older `ISecurityPolicy` adapter, the system will still work, but it will print deprecation warnings when such a policy is used.
- The way the (internal) `IViewPermission` utilities registered via ZCML are invoked has changed. They are purely adapters now, returning a boolean result, rather than returning a callable. You shouldn’t have been using these anyway. ;-)
- New concrete implementations of `IAuthenticationPolicy` have been added to the `repoze.bfg.authentication` module: `RepozeWho1AuthenticationPolicy` which uses `repoze.who` identity to retrieve authentication data from and `RemoteUserAuthenticationPolicy`, which uses the `REMOTE_USER` value in the WSGI environment to retrieve authentication data.
- A new concrete implementation of `IAuthorizationPolicy` has been added to the `repoze.bfg.authorization` module: `ACLAuthorizationPolicy` which uses ACL inheritance to do authorization.
- It is now possible to register a custom `repoze.bfg.interfaces.IForbiddenResponseFactory` for a given application. This feature replaces the `repoze.bfg.interfaces.IUnauthorizedAppFactory` feature previously described in the Hooks chapter. The `IForbiddenResponseFactory` will be called when the framework detects an authorization failure; it should accept a context object and a request object; it should return an `IResponse` object (a webob response, basically). Read the below point for more info and see the Hooks narrative chapter of the BFG docs for more info.

Backwards Incompatibilities

- Custom `NotFound` and `Forbidden` (nee' `Unauthorized`) WSGI applications (registered as a utility for `INotFoundAppFactory` and `IUnauthorizedAppFactory`) could rely on an environment key named `message` describing the circumstance of the response. This key has been renamed to `repoze.bfg.message` (as per the WSGI spec, which requires environment extensions to contain dots).

Deprecations

- The `repoze.bfg.interfaces.IUnauthorizedAppFactory` interface has been deprecated in favor of using the new `repoze.bfg.interfaces.IForbiddenResponseFactory` mechanism.
- The `view_execution_permitted` API should now be imported from the `repoze.bfg.security` module instead of the `repoze.bfg.view` module.
- The `authenticated_userid` and `effective_principals` APIs in `repoze.bfg.security` used to only take a single argument (`request`). They now accept two arguments (`context` and `request`). Calling them with a single argument is still supported but issues a deprecation warning. (NOTE: this change was reverted in 0.9a7; meaning the 0.9 versions of these functions again accept `request` only, just like 0.8 and before).
- Use of “old-style” security policies (those base on `ISecurityPolicy`) is now deprecated. See the “Security” chapter of the docs for info about activating an authorization policy and an authentication policy.

0.8.1 (2009-05-21)

Features

- Class objects may now be used as view callables (both via ZCML and via use of the `bfg_view` decorator in Python 2.6 as a class decorator). The calling semantics when using a class as a view callable is similar to that of using a class as a Zope “browser view”: the class' `__init__` must accept two positional parameters (conventionally named `context`, and `request`). The resulting instance must be callable (it must have a `__call__` method). When called, the instance should return a response. For example:

```
from webob import Response

class MyView(object):
    def __init__(self, context, request):
        self.context = context
        self.request = request

    def __call__(self):
        return Response('hello from %s!' % self.context)

See the "Views" chapter in the documentation and the
``repoze.bfg.view`` API documentation for more information.
```

- Removed the pickling of ZCML actions (the code that wrote `configure.zcml.cache` next to `configure.zcml` files in projects). The code which managed writing and reading of the cache file was a source of subtle bugs when users switched between imperative (e.g. `@bfg_view`) registrations and declarative registrations (e.g. the `view` directive in ZCML) on the same project. On a moderately-sized project (535 ZCML actions and 15 ZCML files), executing actions read from the pickle was saving us only about 200ms (2.5 sec vs 2.7 sec average). On very small projects (1 ZCML file and 4 actions), startup time was comparable, and sometimes even slower when reading from the pickle, and both ways were so fast that it really just didn't matter anyway.

0.8 (2009-05-18)

Features

- Added a `traverse` function to the `repoze.bfg.traversal` module. This function may be used to retrieve certain values computed during path resolution. See the Traversal API chapter of the documentation for more information about this function.

Deprecations

- Internal: `ITraverser` callables should now return a dictionary rather than a tuple. Up until 0.7.0, all `ITraversers` were assumed to return a 3-tuple. In 0.7.1, `ITraversers` were assumed to return a 6-tuple. As (by evidence) it's likely we'll need to add further information to the return value of an `ITraverser` callable, 0.8 assumes that an `ITraverser` return a dictionary with certain elements in it. See the `repoze.bfg.interfaces.ITraverser` interface for the list of keys that should be present in the dictionary. `ITraversers` which return tuples will still work, although a deprecation warning will be issued.

Backwards Incompatibilities

- If your code used the `ITraverser` interface directly (not via an API function such as `find_model`) via an adapter lookup, you'll need to change your code to expect a dictionary rather than a 3- or 6-tuple if your code ever gets return values from the default `ModelGraphTraverser` or `RoutesModelTraverser` adapters.

0.8a7 (2009-05-16)

Backwards Incompatibilities

- The `RoutesMapper` class in `repoze.bfg.urldispatch` has been removed, as well as its documentation. It had been deprecated since 0.6.3. Code in `repoze.bfg.urldispatch.RoutesModelTraverser` which catered to it has also been removed.
- The semantics of the `route` ZCML directive have been simplified. Previously, it was assumed that to use a route, you wanted to map a route to an externally registered view. The new `route` directive instead has a `view` attribute which is required, specifying the dotted path to a view callable. When a route directive is processed, a view is *registered* using the name attribute of the route directive as its name and the callable as its value. The `view_name` and `provides` attributes of the `route` directive are therefore no longer used. Effectively, if you were previously using the `route` directive, it means you must change a pair of ZCML directives that look like this:

```
<route
  name="home"
  path=""
  view_name="login"
  factory=".models.root.Root"
/>

<view
  for=".models.root.Root"
  name="login"
  view=".views.login_view"
/>
```

To a ZCML directive that looks like this:

```
<route
  name="home"
  path=""
  view=".views.login_view"
  factory=".models.root.Root"
/>
```

In other words, to make old code work, remove the `view` directives that were only there to serve the purpose of backing `route` directives, and move their `view=` attribute into the `route` directive itself.

This change also necessitated that the `name` attribute of the `route` directive is now required. If you were previously using `route` directives without a `name` attribute, you'll need to add one (the name is arbitrary, but must be unique among all `route` and `view` statements).

The `provides` attribute of the `route` directive has also been removed. This directive specified a sequence of interface types that the generated context would be decorated with. Since route views are always generated now for a single interface (`repoze.bfg.IRoutesContext`) as opposed to being looked up arbitrarily, there is no need to decorate any context to ensure a view is found.

Documentation

- Added API docs for the `repoze.bfg.testing` methods `registerAdapter`, `registerUtility`, `registerSubscriber`, and `cleanUp`.
- Added glossary entry for “root factory”.
- Noted existence of `repoze.bfg.pagetemplate` template bindings in “Available Add On Template System Bindings” in Templates chapter in narrative docs.
- Update “Templates” narrative chapter in docs (expand to show a sample template and correct macro example).

Features

- Courtesy Carlos de la Guardia, added an `alchemy` Paster template. This paster template sets up a BFG project that uses `SQLAlchemy` (with `SQLite`) and uses traversal to resolve URLs. (no Routes are used). This template can be used via `paster create -t bfg_alchemy`.
- The `Routes Route` object used to resolve the match is now put into the environment as `bfg.route` when URL dispatch is used.
- You can now change the default Routes “context factory” globally. See the “ZCML Hooks” chapter of the documentation (in the “Changing the Default Routes Context Factory” section).

0.8a6 (2009-05-11)

Features

- Added a `routesalchemy` Paster template. This paster template sets up a BFG project that uses SQLAlchemy (with SQLite) and uses Routes exclusively to resolve URLs (no traversal root factory is used). This template can be used via `paster create -t bfg_routesalchemy`.

Documentation

- Added documentation to the URL Dispatch chapter about how to catch the root URL using a ZCML `route` directive.
- Added documentation to the URL Dispatch chapter about how to perform a cleanup function at the end of a request (e.g. close the SQL connection).

Bug Fixes

- In version 0.6.3, passing a `get_root` callback (a “root factory”) to `repoze.bfg.router.make_app` became optional if any `route` declaration was made in ZCML. The intent was to make it possible to disuse traversal entirely, instead relying entirely on URL dispatch (Routes) to resolve all contexts. However a compound set of bugs prevented usage of a Routes-based root view (a view which responds to “/”). One bug existed in `repoze.bfg.urldispatch`, another existed in Routes itself.

To resolve this issue, the `urldispatch` module was fixed, and a fork of the Routes trunk was put into the “dev” index named `Routes-1.11dev-chrism-home`. The source for the fork exists at <http://bitbucket.org/chrism/routes-home/> (broken link); its contents have been merged into the Routes trunk (what will be Routes 1.11).

0.8a5 (2009-05-08)

Features

- Two new security policies were added: `RemoteUserInheritingACLSecurityPolicy` and `WhoInheritingACLSecurityPolicy`. These are security policies which take into account *all* ACLs defined in the lineage of a context rather than stopping at the first ACL found in a lineage. See the “Security” chapter of the API documentation for more information.
- The API and narrative documentation dealing with security was changed to introduce the new “inheriting” security policy variants.
- Added glossary entry for “lineage”.

Deprecations

- The security policy previously named `RepozeWhoIdentityACLSecurityPolicy` now has the slightly saner name of `WhoACLSecurityPolicy`. A deprecation warning is emitted when this policy is imported under the “old” name; usually this is due to its use in ZCML within your application. If you’re getting this deprecation warning, change your ZCML to use the new name, e.g. change:

```
<utility
  provides="repoze.bfg.interfaces.ISecurityPolicy"
  factory="repoze.bfg.security.RepozeWhoIdentityACLSecurityPolicy"
/>
```

To:

```
<utility
  provides="repoze.bfg.interfaces.ISecurityPolicy"
  factory="repoze.bfg.security.WhoACLSecurityPolicy"
/>
```

0.8a4 (2009-05-04)

Features

- `zope.testing` is no longer a direct dependency, although our dependencies (such as `zope.interface`, `repoze.zcml`, etc) still depend on it.
- Tested on Google App Engine. Added a tutorial to the documentation explaining how to deploy a BFG app to GAE.

Backwards Incompatibilities

- Applications which rely on `zope.testing.cleanup.cleanUp` in unit tests can still use that function indefinitely. However, for maximum forward compatibility, they should import `cleanUp` from `repoze.bfg.testing` instead of from `zope.testing.cleanup`. The BFG pasteur templates and docs have been changed to use this function instead of the `zope.testing.cleanup` version.

0.8a3 (2009-05-03)**Features**

- Don't require a successful import of `zope.testing` at BFG application runtime. This allows us to get rid of `zope.testing` on platforms like GAE which have file limits.

0.8a2 (2009-05-02)**Features**

- We no longer include the `configure.zcml` of the `chameleon.zpt` package within the `configure.zcml` of the “`repoze.bfg.includes`” package. This has been a no-op for some time now.
- The `repoze.bfg.chameleon_zpt` package no longer imports from `chameleon.zpt` at module scope, deferring the import until later within a method call. The `chameleon.zpt` package can't be imported on platforms like GAE.

0.8a1 (2009-05-02)**Deprecation Warning and Import Alias Removals**

- Since version 0.6.1, a deprecation warning has been emitted when the name `model_url` is imported from the `repoze.bfg.traversal` module. This import alias (and the deprecation warning) has been removed. Any import of the `model_url` function will now need to be done from `repoze.bfg.url`; any import of the name `model_url` from `repoze.bfg.traversal` will now fail. This was done to remove a dependency on `zope.deferredimport`.
- Since version 0.6.5, a deprecation warning has been emitted when the name `RoutesModelTraverser` is imported from the `repoze.bfg.traversal` module. This import alias (and the deprecation warning) has been removed. Any import of the `RoutesModelTraverser` class will now need to be done from `repoze.bfg.urldispatch`; any import of the name `RoutesModelTraverser` from `repoze.bfg.traversal` will now fail. This was done to remove a dependency on `zope.deferredimport`.

Features

- This release of `repoze.bfg` is “C-free”. This means it has no hard dependencies on any software that must be compiled from C source at installation time. In particular, `repoze.bfg` no longer depends on the `lxml` package.

This change has introduced some backwards incompatibilities, described in the “Backwards Incompatibilities” section below.

- This release was tested on Windows XP. It appears to work fine and all the tests pass.

Backwards Incompatibilities

Incompatibilities related to making `repoze.bfg` “C-free”:

- Removed the `repoze.bfg.chameleon_genshi` module, and thus support for Genshi-style chameleon templates. Genshi-style Chameleon templates depend upon `lxml`, which is implemented in C (as opposed to pure Python) and the `repoze.bfg` core is “C-free” as of this release. You may get Genshi-style Chameleon support back by installing the `repoze.bfg.chameleon_genshi` package available from http://svn.repoze.org/repoze.bfg.chameleon_genshi (also available in the index at <http://dist.repoze.org/bfg/0.8/simple>). All existing code that depended on the `chameleon_genshi` module prior to this release of `repoze.bfg` should work without change after this addon is installed.
- Removed the `repoze.bfg.xslt` module and thus support for XSL templates. The `repoze.bfg.xslt` module depended upon `lxml`, which is implemented in C, and the `repoze.bfg` core is “C-free” as of this release. You may get XSL templating back by installing the `repoze.bfg.xslt` package available from <http://svn.repoze.org/repoze.bfg.xslt/> (also available in the index at <http://dist.repoze.org/bfg/0.8/simple>). All existing code that depended upon the `xslt` module prior to this release of `repoze.bfg` should work without modification after this addon is installed.
- Removed the `repoze.bfg.interfaces.INodeTemplateRenderer` interface and the an old b/w compat aliases from that interface to `repoze.bfg.interfaces.INodeTemplate`. This interface must now be imported from the `repoze.bfg.xslt.interfaces` package after installation of the `repoze.bfg.xslt` addon package described above as `repoze.bfg.interfaces.INodeTemplateRenderer`. This interface was never part of any public API.

Other backwards incompatibilities:

- The `render_template` function in `repoze.bfg.chameleon_zpt` returns Unicode instead of a string. Likewise, the individual values returned by the iterable created by the `render_template_to_iterable` function are also each Unicode. This is actually a backwards incompatibility inherited from our new use of the combination of `chameleon.core 1.0b32` (the non-`lxml`-depending version) and `chameleon.zpt 1.0b16+`; the `chameleon.zpt PageTemplateFile` implementation used to return a string, but now returns Unicode.

0.7.1 (2009-05-01)

Index-Related

- The canonical package index location for `repoze.bfg` has changed. The “old” index (<http://dist.repoze.org/lemonade/dev/simple>) has been superseded by a new index location (<http://dist.repoze.org/bfg/current/simple>). The installation documentation has been updated as well as the `setup.cfg` file in this package. The “lemonade” index still exists, but it is not guaranteed to have the latest BFG software in it, nor will it be maintained in the future.

Features

- The “paster create” templates have been modified to use links to the new “bfg.repoze.org” and “docs.repoze.org” websites.
- Added better documentation for virtual hosting at a URL prefix within the virtual hosting docs chapter.
- The interface for `repoze.bfg.interfaces.ITraverser` and the built-in implementations that implement the interface (`repoze.bfg.traversal.ModelGraphTraverser`, and `repoze.bfg.urldispatch.RoutesModelTraverser`) now expect the `__call__` method of an `ITraverser` to return 3 additional arguments: `traversed`, `virtual_root`, and `virtual_root_path` (the old contract was that the `__call__` method of an `ITraverser` returned; three arguments, the contract new is that it returns six). `traversed` will be a sequence of Unicode names that were traversed (including the virtual root path, if any) or `None` if no traversal was performed, `virtual_root` will be a model object representing the virtual root (or the physical root if traversal was not performed), and `virtual_root_path` will be a sequence representing the virtual root path (a sequence of Unicode names) or `None` if traversal was not performed.

Six arguments are now returned from BFG `ITraversers`. They are returned in this order: `context`, `view_name`, `subpath`, `traversed`, `virtual_root`, and `virtual_root_path`.

Places in the BFG code which called an `ITraverser` continue to accept a 3-argument return value, although BFG will generate and log a warning when one is encountered.

- The request object now has the following attributes: `traversed` (the sequence of names traversed or `None` if traversal was not performed), `virtual_root` (the model object representing the virtual root, including the virtual root path if any), and `virtual_root_path` (the sequence of names representing the virtual root path or `None` if traversal was not performed).

- A new decorator named `wsgiapp2` was added to the `repoze.bfg.wsgi` module. This decorator performs the same function as `repoze.bfg.wsgi.wsgiapp` except it fixes up the `SCRIPT_NAME`, and `PATH_INFO` environment values before invoking the WSGI subapplication.
- The `repoze.bfg.testing.DummyRequest` object now has default attributes for `traversed`, `virtual_root`, and `virtual_root_path`.
- The `RoutesModelTraverser` now behaves more like the Routes “`RoutesMiddleware`” object when an element in the match dict is named `path_info` (usually when there’s a pattern like `http://foo/*path_info`). When this is the case, the `PATH_INFO` environment variable is set to the value in the match dict, and the `SCRIPT_NAME` is appended to with the prefix of the original `PATH_INFO` not including the value of the new variable.
- The `notfound` debug now shows the traversed path, the virtual root, and the virtual root path too.
- Speed up / clarify ‘traversal’ module’s ‘`model_path`’, ‘`model_path_tuple`’, and ‘`_model_path_list`’ functions.

Backwards Incompatibilities

- In previous releases, the `repoze.bfg.url.model_url`, `repoze.bfg.traversal.model_path` and `repoze.bfg.traversal.model_path_tuple` functions always ignored the `__name__` argument of the root object in a model graph (effectively replacing it with a leading / in the returned value) when a path or URL was generated. The code required to perform this operation was not efficient. As of this release, the root object in a model graph *must* have a `__name__` attribute that is either `None` or the empty string (‘’) for URLs and paths to be generated properly from these APIs. If your root model object has a `__name__` argument that is not one of these values, you will need to change your code for URLs and paths to be generated properly. If your model graph has a root node with a string `__name__` that is not null, the value of `__name__` will be prepended to every path and URL generated.
- The `repoze.bfg.location.LocationProxy` class and the `repoze.bfg.location.ClassAndInstanceDescr` class have both been removed in order to be able to eventually shed a dependency on `zope.proxy`. Neither of these classes was ever an API.
- In all previous releases, the `repoze.bfg.location.locate` function worked like so: if a model did not explicitly provide the `repoze.bfg.interfaces.ILocation` interface, `locate` returned a `LocationProxy` object representing model with its `__parent__` attribute assigned to parent and a `__name__` attribute assigned to `__name__`. In this release, the `repoze.bfg.location.locate` function simply jams the `__name__` and `__parent__` attributes on to the supplied model unconditionally, no matter if the object implements `ILocation` or not, and it never returns a proxy. This was done because the `LocationProxy` behavior has now moved into an add-on package (`repoze.bfg.traversalwrapper`), in order to eventually be able to shed a dependency on `zope.proxy`.

- In all previous releases, by default, if traversal was used (as opposed to URL-dispatch), and the root object supplied the `repoze.bfg.interfaces.ILocation` interface, but the children returned via its `__getitem__` returned an object that did not implement the same interface, `repoze.bfg` provided some implicit help during traversal. This traversal feature wrapped subobjects from the root (and thereafter) that did not implement `ILocation` in proxies which automatically provided them with a `__name__` and `__parent__` attribute based on the name being traversed and the previous object traversed. This feature has now been removed from the base `repoze.bfg` package for purposes of eventually shedding a dependency on `zope.proxy`.

In order to re-enable the wrapper behavior for older applications which cannot be changed, register the “traversalwrapper” `ModelGraphTraverser` as the traversal policy, rather than the default `ModelGraphTraverser`. To use this feature, you will need to install the `repoze.bfg.traversalwrapper` package (an add-on package, available at <http://svn.repoze.org/repoze.bfg.traversalwrapper>) Then change your application’s `configure.zcml` to include the following stanza:

```
<adapter factory="repoze.bfg.traversalwrapper.ModelGraphTraverser" provides="repoze.bfg.interfaces.ITraverserFactory" for="*" />
```

When this `ITraverserFactory` is used instead of the default, no object in the graph (even the root object) must supply a `__name__` or `__parent__` attribute. Even if subobjects returned from the root *do* implement the `ILocation` interface, these will still be wrapped in proxies that override the object’s “real” `__parent__` and `__name__` attributes.

See also changes to the “Models” chapter of the documentation (in the “Location-Aware Model Instances”) section.

0.7.0 (2009-04-11)

Bug Fixes

- Fix a bug in `repoze.bfg.wsgi.HTTPException`: the content length was returned as an int rather than as a string.
- Add explicit dependencies on `zope.deferredimport`, `zope.deprecation`, and `zope.proxy` for forward compatibility reasons (`zope.component` will stop relying on `zope.deferredimport` soon and although we use it directly, it’s only a transitive dependency, and “`zope.deprecation`” and `zope.proxy` are used directly even though they’re only transitive dependencies as well).

- Using `model_url` or `model_path` against a broken model graph (one with models that had a non-root model with a `__name__` of `None`) caused an inscrutable error to be thrown: (if not `_must_quote[cachekey].search(s): TypeError: expected string or buffer`). Now URLs and paths generated against graphs that have `None` names in intermediate nodes will replace the `None` with the empty string, and, as a result, the error won't be raised. Of course the URL or path will still be bogus.

Features

- Make it possible to have `testing.DummyTemplateRenderer` return some nondefault string representation.
- Added a new `anchor` keyword argument to `model_url`. If `anchor` is present, its string representation will be used as a named anchor in the generated URL (e.g. if `anchor` is passed as `foo` and the model URL is `http://example.com/model/url`, the generated URL will be `http://example.com/model/url#foo`).

Backwards Incompatibilities

- The default request charset encoding is now `utf-8`. As a result, the request machinery will attempt to decode values from the `utf-8` encoding to Unicode automatically when they are obtained via `request.params`, `request.GET`, and `request.POST`. The previous behavior of BFG was to return a bytestring when a value was accessed in this manner. This change will break form handling code in apps that rely on values from those APIs being considered bytestrings. If you are manually decoding values from form submissions in your application, you'll either need to change the code that does that to expect Unicode values from `request.params`, `request.GET` and `request.POST`, or you'll need to explicitly reenale the previous behavior. To reenale the previous behavior, add the following to your application's `configure.zcml`:

```
<subscriber for="repoze.bfg.interfaces.INewRequest"
            handler="repoze.bfg.request.make_request_ascii"/>
```

See also the documentation in the “Views” chapter of the BFG docs entitled “Using Views to Handle Form Submissions (Unicode and Character Set Issues)”.

Documentation

- Add a section to the narrative Views chapter entitled “Using Views to Handle Form Submissions (Unicode and Character Set Issues)” explaining implicit decoding of form data values.

0.6.9 (2009-02-16)

Bug Fixes

- lru cache was unstable under concurrency (big surprise!) when it tried to redelete a key in the cache that had already been deleted. Symptom: line 64 in `put:del data[oldkey]:KeyError: 'some/path'`. Now we just ignore the key error if we can't delete the key (it has already been deleted).
- Empty location names in model paths when generating a URL using `repoze.bfg.model_url` based on a model obtained via traversal are no longer ignored in the generated URL. This means that if a non-root model object has a `__name__` of `' '`, the URL will reflect it (e.g. `model_url` will generate `http://foo/bar//baz` if an object with the `__name__` of `' '` is a child of `bar` and the parent of `baz`). URLs generated with empty path segments are, however, still irresolvable by the model graph traverser on request ingress (the traverser strips empty path segment names).

Features

- Microspeedups of `repoze.bfg.traversal.model_path`, `repoze.bfg.traversal.model_path_tuple`, `repoze.bfg.traversal.quote_path_segment`, and `repoze.bfg.url.urlencode`.
- add `zip_safe = false` to `setup.cfg`.

Documentation

- Add a note to the `repoze.bfg.traversal.quote_path_segment` API docs about caching of computed values.

Implementation Changes

- Simplification of `repoze.bfg.traversal.TraversalContextURL.__call__` (it now uses `repoze.bfg.traversal.model_path` instead of rolling its own path-generation).

0.6.8 (2009-02-05)

Backwards Incompatibilities

- The `repoze.bfg.traversal.model_path` API now returns a *quoted* string rather than a string represented by series of unquoted elements joined via `/` characters. Previously it returned a string or unicode object representing the model path, with each segment name in the path joined together via `/` characters, e.g. `/foo /bar`. Now it returns a string, where each segment is a UTF-8 encoded and URL-quoted element e.g. `/foo%20/bar`. This change was (as discussed briefly on the `repoze-dev` maillist) necessary to accomodate model objects which themselves have `__name__` attributes that contain the `/` character.

For people that have no models that have high-order Unicode `__name__` attributes or `__name__` attributes with values that require URL-quoting with in their model graphs, this won't cause any issue. However, if you have code that currently expects `model_path` to return an unquoted string, or you have an existing application with data generated via the old method, and you're too lazy to change anything, you may wish replace the BFG-imported `model_path` in your code with this function (this is the code of the "old" `model_path` implementation):

```
from repoze.bfg.location import lineage

def i_am_too_lazy_to_move_to_the_new_model_path(model, *elements):
    rpath = []
    for location in lineage(model):
        if location.__name__:
            rpath.append(location.__name__)
    path = '/' + '/'.join(reversed(rpath))
    if elements:
        suffix = '/'.join(elements)
        path = '/'.join([path, suffix])
    return path
```

- The `repoze.bfg.traversal.find_model` API no longer implicitly converts unicode representations of a full path passed to it as a Unicode object into a UTF-8 string. Callers should either use prequoted path strings returned by `repoze.bfg.traversal.model_path`, or tuple values returned by the result of `repoze.bfg.traversal.model_path_tuple` or they should use the guidelines about passing a string path argument described in the `find_model` API documentation.

Bugfixes

- Each argument contained in `elements` passed to `repoze.bfg.traversal.model_path` will now have any `/` characters contained within quoted to `%2F` in the returned string. Previously, `/` characters in elements were left unquoted (a bug).

Features

- A `repoze.bfg.traversal.model_path_tuple` API was added. This API is an alternative to `model_path` (which returns a string); `model_path_tuple` returns a model path as a tuple (much like Zope's `getPhysicalPath`).
- A `repoze.bfg.traversal.quote_path_segment` API was added. This API will quote an individual path segment (string or unicode object). See the `repoze.bfg.traversal` API documentation for more information.
- The `repoze.bfg.traversal.find_model` API now accepts “path tuples” (see the above note regarding `model_path_tuple`) as well as string path representations (from `repoze.bfg.traversal.model_path`) as a path argument.
- Add ‘*renderer*’ argument (defaulting to `None`) to `repoze.bfg.testing.registerDummyRenderer`. This makes it possible, for instance, to register a custom renderer that raises an exception in a unit test.

Implementation Changes

- Moved `_url_quote` function back to `repoze.bfg.traversal` from `repoze.bfg.url`. This is not an API.

0.6.7 (2009-01-27)

Features

- The `repoze.bfg.url.model_url` API now works against contexts derived from Routes URL dispatch (`Routes.util.url_for` is called under the hood).
- “Virtual root” support for traversal-based applications has been added. Virtual root support is useful when you’d like to host some model in a `repoze.bfg` model graph as an application under a URL pathname that does not include the model path itself. For more information, see the (new) “Virtual Hosting” chapter in the documentation.
- A `repoze.bfg.traversal.virtual_root` API has been added. When called, it returns the virtual root object (or the physical root object if no virtual root has been specified).

Implementation Changes

- `repoze.bfg.traversal.RoutesModelTraverser` has been moved to `repoze.bfg.urldispatch`.
- `model_url` URL generation is now performed via an adapter lookup based on the context and the request.
- ZCML which registers two adapters for the `IContextURL` interface has been added to the `configure.zcml` in `repoze.bfg.includes`.

0.6.6 (2009-01-26)

Implementation Changes

- There is an indirection in `repoze.bfg.url.model_url` now that consults a utility to generate the base model url (without extra elements or a query string). Eventually this will service virtual hosting; for now it's undocumented and should not be hooked.

0.6.5 (2009-01-26)

Features

- You can now override the `NotFound` and `Unauthorized` responses that `repoze.bfg` generates when a view cannot be found or cannot be invoked due to lack of permission. See the “ZCML Hooks” chapter in the docs for more information.
- Added Routes ZCML directive attribute explanations in documentation.
- Added a `traversal_path` API to the traversal module; see the “traversal” API chapter in the docs. This was a function previously known as `split_path` that was not an API but people were using it anyway. Unlike `split_path`, it now returns a tuple instead of a list (as its values are cached).

Behavior Changes

- The `repoze.bfg.view.render_view_to_response` API will no longer raise a `ValueError` if an object returned by a view function it calls does not possess certain attributes (`headerlist`, `app_iter`, `status`). This API used to attempt to perform a check using the `is_response` function in `repoze.bfg.view`, and raised a `ValueError` if the `is_response` check failed. The responsibility is now the caller's to ensure that the return value from a view function is a "real" response.
- WSGI environ dicts passed to `repoze.bfg`'s Router must now contain a `REQUEST_METHOD` key/value; if they do not, a `KeyError` will be raised (speed).
- It is no longer permissible to pass a "nested" list of principals to `repoze.bfg.ACLAuthorizer.permits` (e.g. `['fred', ['larry', 'bob']]`). The principals list must be fully expanded. This feature was never documented, and was never an API, so it's not a backwards incompatibility.
- It is no longer permissible for a security ACE to contain a "nested" list of permissions (e.g. `(Allow, Everyone, ['read', ['view', ['write', 'manage']]])`). The list must instead be fully expanded (e.g. `((Allow, Everyone, ['read', 'view', 'write', 'manage'])))`). This feature was never documented, and was never an API, so it's not a backwards incompatibility.
- The `repoze.bfg.urldispatch.RoutesRootFactory` now injects the `wsgiorg.routing_args` environment variable into the environ when a route matches. This is a tuple of `((), routing_args)` where `routing_args` is the value that comes back from the routes mapper match (the "match dict").
- The `repoze.bfg.traversal.RoutesModelTraverser` class now wants to obtain the `view_name` and `subpath` from the `wsgiorgs.routing_args` environment variable. It falls back to obtaining these from the context for backwards compatibility.

Implementation Changes

- Get rid of `repoze.bfg.security.ACLAuthorizer`: the `ACLSecurityPolicy` now does what it did inline.
- Get rid of `repoze.bfg.interfaces.NoAuthorizationInformation` exception: it was used only by `ACLAuthorizer`.
- Use a homegrown `NotFound` error instead of `webob.exc.HTTPNotFound` (the latter is slow).
- Use a homegrown `Unauthorized` error instead of `webob.exc.Unauthorized` (the latter is slow).
- the `repoze.bfg.lru.lru_cached` decorator now uses `functools.wraps` in order to make documentation of LRU-cached functions possible.
- Various speed micro-tweaks.

Bug Fixes

- `repoze.bfg.testing.DummyModel` did not have a `get` method; it now does.

0.6.4 (2009-01-23)

Backwards Incompatibilities

- The `unicode_path_segments` configuration variable and the `BFG_UNICODE_PATH_SEGMENTS` configuration variable have been removed. Path segments are now always passed to model `__getitem__` methods as unicode. “True” has been the default for this setting since 0.5.4, but changing this configuration setting to false allowed you to go back to passing raw path element strings to model `__getitem__` methods. Removal of this knob services a speed goal (we get about +80 req/s by removing the check), and it’s clearer just to always expect unicode path segments in model `__getitem__` methods.

Implementation Changes

- `repoze.bfg.traversal.split_path` now also handles decoding path segments to unicode (for speed, because its results are cached).
- **`repoze.bfg.traversal.step` was made a method of the `ModelGraphTraverser`.**
- Use “precooked” Request subclasses (e.g. `repoze.bfg.request.GETRequest`) that correspond to HTTP request methods within `router.py` when constructing a request object rather than using `alsoProvides` to attach the proper interface to an un subclassed `webob.Request`. This pattern is purely an optimization (e.g. preventing calls to `alsoProvides` means the difference between 590 r/s and 690 r/s on a MacBook 2GHz).
- Tease out an extra 4% performance boost by changing the Router; instead of using imported ZCA APIs, use the same APIs directly against the registry that is an attribute of the Router.
- The registry used by BFG is now a subclass of `zope.component.registry.Components` (defined as `repoze.bfg.registry.Registry`); it has a `notify` method, a `registerSubscriptionAdapter` and a `registerHandler` method. If no subscribers are registered via `registerHandler` or `registerSubscriptionAdapter`, `notify` is a noop for speed.
- The `Allowed` and `Denied` classes in `repoze.bfg.security` now are lazier about constructing the representation of a reason message for speed; `repoze.bfg.view_execution_permitted` takes advantage of this.
- The `is_response` check was sped up by about half at the expense of making its code slightly uglier.

New Modules

- `repoze.bfg.lru` implements an LRU cache class and a decorator for internal use.

0.6.3 (2009-01-19)

Bug Fixes

- Readd `root_policy` attribute on Router object (as a property which returns the `IRootFactory` utility). It was inadvertently removed in 0.6.2. Code in the wild depended upon its presence (esp. scripts and “debug” helpers).

Features

- URL-dispatch has been overhauled: it is no longer necessary to manually create a `RoutesMapper` in your application’s entry point callable in order to use URL-dispatch (aka `Routes`). A new `route` directive has been added to the available list of ZCML directives. Each `route` directive inserted into your application’s `configure.zcml` establishes a `Routes` mapper connection. If any `route` declarations are made via ZCML within a particular application, the `get_root` callable passed in to `repoze.bfg.router.make_app` will automatically be wrapped in the equivalent of a `RoutesMapper`. Additionally, the new `route` directive allows the specification of a `context_interfaces` attribute for a route, this will be used to tag the manufactured routes context with specific interfaces when a route specifying a `context_interfaces` attribute is matched.
- A new interface `repoze.bfg.interfaces.IContextNotFound` was added. This interface is attached to a “dummy” context generated when `Routes` cannot find a match and there is no “fallback” `get_root` callable that uses traversal.
- The `bfg_starter` and `bfg_zodb` “paster create” templates now contain images and CSS which are displayed when the default page is displayed after initial project generation.
- Allow the `repoze.bfg.view.static` helper to be passed a relative `root_path` name; it will be considered relative to the file in which it was called.
- The functionality of `repoze.bfg.convention` has been merged into the core. Applications which make use of `repoze.bfg.convention` will continue to work indefinitely, but it is recommended that apps stop depending upon it. To do so, substitute imports of `repoze.bfg.convention.bfg_view` with imports of `repoze.bfg.view.bfg_view`, and change the stanza in ZCML from `<convention package=". ">` to `<scan package=". ">`. As a result of the merge, `bfg` has grown a new dependency: `martian`.

- View functions which use the `pushpage` decorator are now pickleable (meaning their use won't prevent a `configure.zcml.cache` file from being written to disk).
- Instead of invariably using `webob.Request` as the “request factory” (e.g. in the `Router` class) and `webob.Response` and the “response factory” (e.g. in `render_template_to_response`), allow both to be overridden via a ZCML utility hook. See the “Using ZCML Hooks” chapter of the documentation for more information.

Deprecations

- The class `repoze.bfg.urldispatch.RoutesContext` has been renamed to `repoze.bfg.urldispatch.DefaultRoutesContext`. The class should be imported by the new name as necessary (although in reality it probably shouldn't be imported from anywhere except internally within BFG, as it's not part of the API).

Implementation Changes

- The `repoze.bfg.wsgi.wsgiapp` decorator now uses `webob.Request.get_response` to do its work rather than relying on homegrown WSGI code.
- The `repoze.bfg.view.static` helper now uses `webob.Request.get_response` to do its work rather than relying on homegrown WSGI code.
- The `repoze.bfg.urldispatch.RoutesModelTraverser` class has been moved to `repoze.bfg.traversal.RoutesModelTraverser`.
- The `repoze.bfg.registry.makeRegistry` function was renamed to `repoze.bfg.registry.populateRegistry` and now accepts a `registry` argument (which should be an instance of `zope.component.registry.Components`).

Documentation Additions

- Updated narrative `urldispatch` chapter with changes required by `<route..>` ZCML directive.
- Add a section on “Using BFG Security With URL Dispatch” into the `urldispatch` chapter of the documentation.
- Better documentation of security policy implementations that ship with `repoze.bfg`.
- Added a “Using ZPT Macros in `repoze.bfg`” section to the narrative templating chapter.

0.6.2 (2009-01-13)

Features

- Tests can be run with coverage output if you've got `nose` installed in the interpreter which you use to run tests. Using an interpreter with `nose` installed, do `python setup.py nosetests` within a checkout of the `repoze.bfg` package to see test coverage output.
- Added a `post` argument to the `repoze.bfg.testing:DummyRequest` constructor.
- Added `__len__` and `__nonzero__` to `repoze.bfg.testing:DummyModel`.
- The `repoze.bfg.registry.get_options` callable (now renamed to `repoze.bfg.settings.get_options`) used to return only framework-specific keys and values in the dictionary it returned. It now returns all the keys and values in the dictionary it is passed *plus* any framework-specific settings culled from the environment. As a side effect, all PasteDeploy application-specific config file settings are made available as attributes of the `ISettings` utility from within BFG.
- Renamed the existing BFG paster template to `bfg_starter`. Added another template (`bfg_zodb`) showing default ZODB setup using `repoze.zodbconn`.
- Add a method named `assert_` to the `DummyTemplateRenderer`. This method accepts keyword arguments. Each key/value pair in the keyword arguments causes an assertion to be made that the renderer received this key with a value equal to the asserted value.
- Projects generated by the paster templates now use the `DummyTemplateRenderer.assert_` method in their view tests.
- Make the (internal) thread local registry manager maintain a stack of registries in order to make it possible to call one BFG application from inside another.
- An interface specific to the HTTP verb (GET/PUT/POST/DELETE/HEAD) is attached to each request object on ingress. The HTTP-verb-related interfaces are defined in `repoze.bfg.interfaces` and are `IGETRequest`, `IPOSTRequest`, `IPUTRequest`, `IDELETERequest` and `IHEADRequest`. These interfaces can be specified as the `request_type` attribute of a bfg view declaration. A view naming a specific HTTP-verb-matching interface will be found only if the view is defined with a `request_type` that matches the HTTP verb in the incoming request. The more general `IRequest` interface can be used as the `request_type` to catch all requests (and this is indeed the default). All requests implement `IRequest`. The HTTP-verb-matching idea was pioneered by `repoze.bfg.restrequest`. That package is no longer required, but still functions fine.

Bug Fixes

- Fix a bug where the Paste configuration's `unicode_path_segments` (and `os.environ's` `BFG_UNICODE_PATH_SEGMENTS`) may have been defaulting to false in some circumstances. It now always defaults to true, matching the documentation and intent.
- The `repoze.bfg.traversal.find_model` API did not work properly when passed a path argument which was unicode and contained high-order bytes when the `unicode_path_segments` or `BFG_UNICODE_PATH_SEGMENTS` configuration variables were "true".
- A new module was added: `repoze.bfg.settings`. This contains deployment-settings-related code.

Implementation Changes

- The `make_app` callable within `repoze.bfg.router` now registers the `root_policy` argument as a utility (unnamed, using the new `repoze.bfg.interfaces.IRootFactory` as a provides interface) rather than passing it as the first argument to the `repoze.bfg.router.Router` class. As a result, the `repoze.bfg.router.Router` class only accepts a single argument: `registry`. The `repoze.bfg.router.Router` class retrieves the root policy via a utility lookup now. The `repoze.bfg.router.make_app` API also now performs some important application registrations that were previously handled inside `repoze.bfg.registry.makeRegistry`.

New Modules

- A `repoze.bfg.settings` module was added. It contains code related to deployment settings. Most of the code it contains was moved to it from the `repoze.bfg.registry` module.

Behavior Changes

- The `repoze.bfg.settings.Settings` class (an instance of which is registered as a utility providing `repoze.bfg.interfaces.ISettings` when any application is started) now automatically calls `repoze.bfg.settings.get_options` on the options passed to its constructor. This means that usage of `get_options` within an application's `make_app` function is no longer required (the "raw" options dict or None may be passed).
- Remove old code which attempts to recover from trying to unpickle a `z3c.pt` template; Chameleon has been the templating engine for a good long time now. Running `repoze.bfg` against a sandbox that has pickled `z3c.pt` templates it will now just fail with an unpickling error, but can be fixed by deleting the template cache files.

Deprecations

- Moved the `repoze.bfg.registry.Settings` class. This has been moved to `repoze.bfg.settings.Settings`. A deprecation warning is issued when it is imported from the older location.
- Moved the `repoze.bfg.registry.get_options` function. This has been moved to `repoze.bfg.settings.get_options`. A deprecation warning is issued when it is imported from the older location.
- The `repoze.bfg.interfaces.IRootPolicy` interface was renamed within the `interfaces` package. It has been renamed to `IRootFactory`. A deprecation warning is issued when it is imported from the older location.

0.6.1 (2009-01-06)

New Modules

- A new module `repoze.bfg.url` has been added. It contains the `model_url` API (moved from `repoze.bfg.traversal`) and an implementation of `urlencode` (like Python's `urllib.urlencode`) which can handle Unicode keys and values in parameters to the `query` argument.

Deprecations

- The `model_url` function has been moved from `repoze.bfg.traversal` into `repoze.bfg.url`. It can still be imported from `repoze.bfg.traversal` but an import from `repoze.bfg.traversal` will emit a `DeprecationWarning`.

Features

- A static helper class was added to the `repoze.bfg.views` module. Instances of this class are willing to act as BFG views which return static resources using files on disk. See the `repoze.bfg.view` docs for more info.
- The `repoze.bfg.url.model_url` API (nee' `repoze.bfg.traversal.model_url`) now accepts and honors a keyword argument named `query`. The value of this argument will be used to compose a query string, which will be attached to the generated URL before it is returned. See the API docs (in the docs directory or on the web) for more information.

0.6 (2008-12-26)

Backwards Incompatibilities

- Rather than prepare the “stock” implementations of the ZCML directives from the `zope.configuration` package for use under `repoze.bfg`, `repoze.bfg` now makes available the implementations of directives from the `repoze.zcml` package (see <http://static.repoze.org/zcmldocs>). As a result, the `repoze.bfg` package now depends on the `repoze.zcml` package, and no longer depends directly on the `zope.component`, `zope.configuration`, `zope.interface`, or `zope.proxy` packages.

The primary reason for this change is to enable us to eventually reduce the number of inappropriate `repoze.bfg` Zope package dependencies, as well as to shed features of dependent package directives that don’t make sense for `repoze.bfg`.

Note that currently the set of requirements necessary to use `bfg` has not changed. This is due to inappropriate Zope package requirements in `chameleon.zpt`, which will hopefully be remedied soon. NOTE: in lemonade index a `1.0b8-repozezcml0` package exists which does away with these requirements.

- BFG applications written prior to this release which expect the “stock” `zope.component` ZCML directive implementations (e.g. `adapter`, `subscriber`, or `utility`) to function now must either 1) include the `meta.zcml` file from `zope.component` manually (e.g. `<include package="zope.component" file="meta.zcml">`) and include the `zope.security` package as an `install_requires` dependency or 2) change the ZCML in their applications to use the declarations from `repoze.zcml` instead of the stock declarations. `repoze.zcml` only makes available the `adapter`, `subscriber` and `utility` directives.

In short, if you’ve got an existing BFG application, after this update, if your application won’t start due to an import error for “`zope.security`”, the fastest way to get it working again is to add `zope.security` to the “`install_requires`” of your BFG application’s `setup.py`, then add the following ZCML anywhere in your application’s `configure.zcml`:

```
<include package="zope.component" file="meta.zcml">
```

Then re-`setup.py develop` or reinstall your application.

- The <http://namespaces.repoze.org/bfg> XML namespace is now the default XML namespace in ZCML for pasteur-generated applications. The docs have been updated to reflect this.

- The copies of BFG's `meta.zcml` and `configure.zcml` were removed from the root of the `repoze.bfg` package. In 0.3.6, a new package named `repoze.bfg.includes` was added, which contains the “correct” copies of these ZCML files; the ones that were removed were for backwards compatibility purposes.
- The BFG `view` ZCML directive no longer calls `zope.component.interface.provideInterface` for the `for` interface. We don't support `provideInterface` in BFG because it mutates the global registry.

Other

- The minimum requirement for `chameleon.core` is now 1.0b13. The minimum requirement for `chameleon.zpt` is now 1.0b8. The minimum requirement for `chameleon.genshi` is now 1.0b2.
- Updated `pastertemplate` “`ez_setup.py`” to one that requires `setuptools` 0.6c9.
- Turn `view_execution_permitted` from the `repoze.bfg.view` module into a documented API.
- Doc cleanups.
- Documented how to create a view capable of serving static resources.

0.5.6 (2008-12-18)

- Speed up `traversal.model_url` execution by using a custom url quoting function instead of Python's `urllib.quote`, by caching URL path segment quoting and encoding results, by disusing Python's `urlparse.urljoin` in favor of a simple string concatenation, and by using `ob.__class__ is unicode` rather than `isinstance(ob, unicode)` in one strategic place.

0.5.5 (2008-12-17)

Backwards Incompatibilities

- In the past, during traversal, the `ModelGraphTraverser` (the default traverser) always passed each URL path segment to any `__getitem__` method of a model object as a byte string (a `str` object). Now, by default the `ModelGraphTraverser` attempts to decode the path segment to Unicode (a `unicode` object) using the UTF-8 encoding before passing it to the `__getitem__` method of a model object. This makes it possible for model objects to be dumber in `__getitem__` when trying to resolve a subobject, as model objects themselves no longer need to try to divine whether or not to try to decode the path segment passed by the traverser.

Note that since 0.5.4, URLs generated by `repoze.bfg`'s `model_url` API will contain UTF-8 encoded path segments as necessary, so any URL generated by BFG itself will be decodeable by the traverser. If another application generates URLs to a BFG application, to be resolved successfully, it should generate the URL with UTF-8 encoded path segments to be successfully resolved. The decoder is not at all magical: if a non-UTF-8-decodeable path segment (e.g. one encoded using UTF-16 or some other insanity) is passed in the URL, BFG will raise a `TypeError` with a message indicating it could not decode the path segment.

To turn on the older behavior, where path segments were not decoded to Unicode before being passed to model object `__getitem__` by the traverser, and were passed as a raw byte string, set the `unicode_path_segments` configuration setting to a false value in your BFG application's section of the paste .ini file, for example:

```
unicode_path_segments = False
```

Or start the application using the `BFG_UNICODE_PATH_SEGMENT` envvar set to a false value:

```
BFG_UNICODE_PATH_SEGMENTS=0
```

0.5.4 (2008-12-13)

Backwards Incompatibilities

- URL-quote “extra” element names passed in as `**elements` to the `traversal.model_url` API. If any of these names is a Unicode string, encode it to UTF-8 before URL-quoting. This is a slight backwards incompatibility that will impact you if you were already UTF-8 encoding or URL-quoting the values you passed in as `elements` to this API.

Bugfixes

- UTF-8 encode each segment in the model path used to generate a URL before url-quoting it within the `traversal.model_url` API. This is a bugfix, as Unicode cannot always be successfully URL-quoted.

Features

- Make it possible to run unit tests using a buildout-generated Python “interpreter”.
- Add `request.root` to `router.Router` in order to have easy access to the application root.

0.5.3 (2008-12-07)

- Remove the `ITestingTemplateRenderer` interface. When `testing.registerDummyRenderer` is used, it instead registers a dummy implementation using `ITemplateRenderer` interface, which is checked for when the built-in templating facilities do rendering. This change also allows developers to make explicit named utility registrations in the ZCML registry against `ITemplateRenderer`; these will be found before any on-disk template is looked up.

0.5.2 (2008-12-05)

- The component registration handler for views (functions or class instances) now observes component adaptation annotations (see `zope.component.adaptedBy`) and uses them before the fallback values for `for_` and `request_type`. This change does not affect existing code inasmuch as the code does not rely on these defaults when an annotation is set on the view (unlikely). This means that for a new-style class you can do `zope.component.adapts(ISomeContext, ISomeRequest)` at class scope or at module scope as a decorator to a bfg view function you can do `@zope.component.adapter(ISomeContext, ISomeRequest)`. This differs from `r.bfg.convention` inasmuch as you still need to put something in ZCML for the registrations to get done; it's only the defaults that will change if these declarations exist.
- Strip all slashes from end and beginning of path in `clean_path` within traversal machinery.

0.5.1 (2008-11-25)

- Add `keys`, `items`, and `values` methods to `testing.DummyModel`.
- Add `__delitem__` method to `testing.DummyModel`.

0.5.0 (2008-11-18)

- Fix `ModelGraphTraverser`; don't try to change the `__name__` or `__parent__` of an object that claims it implements `ILocation` during traversal even if the `__name__` or `__parent__` of the object traversed does not match the name used in the traversal step or the or the traversal parent. Rationale: it was insane to do so. This bug was only found due to a misconfiguration in an application that mistakenly had intermediate persistent non-`ILocation` objects; traversal was causing a persistent write on every request under this setup.
- `repoze.bfg.location.locate` now unconditionally sets `__name__` and `__parent__` on objects which provide `ILocation` (it previously only set them conditionally if they didn't match attributes already present on the object via equality).

0.4.9 (2008-11-17)

- Add chameleon text template API (chameleon `${name}` renderings where the template does not need to be wrapped in any containing XML).
- Change docs to explain install in terms of a `virtualenv` (unconditionally).
- Make `pushpage` decorator compatible with `repoze.bfg.convention`'s `bfg_view` decorator when they're stacked.
- Add `content_length` attribute to `testing.DummyRequest`.
- Change paster template `tests.py` to include a true unit test. Retain old test as an integration test. Update documentation.
- Document view registrations against classes and `repoze.bfg.convention` in context.
- Change the default paster template to register its single view against a class rather than an interface.
- Document adding a request type interface to the request via a subscriber function in the events narrative documentation.

0.4.8 (2008-11-12)

Backwards Incompatibilities

- `repoze.bfg.traversal.model_url` now always appends a slash to all generated URLs unless further elements are passed in as the third and following arguments. Rationale: views often use `model_url` without the third-and-following arguments in order to generate a URL for a model in order to point at the default view of a model. The URL that points to the default view of the *root* model is technically `http://mysite/` as opposed to `http://mysite` (browsers happen to ask for `'/'` implicitly in the GET request). Because URLs are never automatically generated for anything *except* models by `model_url`, and because the root model is not really special, we continue this pattern. The impact of this change is minimal (at most you will have too many slashes in your URL, which BFG deals with gracefully anyway).

0.4.7 (2008-11-11)

Features

- Allow `testing.registerEventListener` to be used with Zope 3 style “object events” (subscribers accept more than a single event argument). We extend the list with the arguments, rather than append.

0.4.6 (2008-11-10)

Bug Fixes

- The `model_path` and `model_url` traversal APIs returned the wrong value for the root object (e.g. `model_path` returned `' '` for the root object, while it should have been returning `'/'`).

0.4.5 (2008-11-09)

Features

- Added a `clone` method and a `__contains__` method to the `DummyModel` testing object.
- Allow `DummyModel` objects to receive extra keyword arguments, which will be attached as attributes.
- The `DummyTemplateRenderer` now returns `self` as its implementation.

0.4.4 (2008-11-08)

Features

- Added a `repoze.bfg.testing` module to attempt to make it slightly easier to write unittest-based automated tests of BFG applications. Information about this module is in the documentation.
- The default template renderer now supports testing better by looking for `ITestingTemplateRenderer` using a relative pathname. This is exposed indirectly through the API named `registerTemplateRenderer` in `repoze.bfg.testing`.

Deprecations

- The names `repoze.bfg.interfaces.ITemplate` , `repoze.bfg.interfaces.ITemplateFactory` and `repoze.bfg.interfaces.INodeTemplate` have been deprecated. These should now be imported as `repoze.bfg.interfaces.ITemplateRenderer` and `repoze.bfg.interfaces.ITemplateRendererFactory`, and `INodeTemplateRenderer` respectively.
- The name `repoze.bfg.chameleon_zpt.ZPTTemplateFactory` is deprecated. Use `repoze.bfg.chameleon_zpt.ZPTTemplateRenderer`.
- The name `repoze.bfg.chameleon_genshi.GenshiTemplateFactory` is deprecated. Use `repoze.bfg.chameleon_genshi.GenshiTemplateRenderer`.
- The name `repoze.bfg.xslt.XSLTemplateFactory` is deprecated. Use `repoze.bfg.xslt.XSLTemplateRenderer`.

0.4.3 (2008-11-02)

Bug Fixes

- Not passing the result of “`get_options`” as the second argument of `make_app` could cause attribute errors when attempting to look up settings against the `ISettings` object (internal). Fixed by giving the `Settings` objects defaults for `debug_authorization` and `debug_notfound`.
- Return an instance of `Allowed` (rather than `True`) from `has_permission` when no security policy is in use.
- Fix bug where default deny in authorization check would throw a `TypeError` (use `ACLDenied` instead of `Denied`).

0.4.2 (2008-11-02)

Features

- Expose a single `ILogger` named “`repoze.bfg.debug`” as a utility; this logger is registered unconditionally and is used by the authorization debug machinery. Applications may also make use of it as necessary rather than inventing their own logger, for convenience.
- The `BFG_DEBUG_AUTHORIZATION` envvar and the `debug_authorization` config file value now only imply debugging of view-invoked security checks. Previously, information was printed for every call to `has_permission` as well, which made output confusing. To debug `has_permission` checks and other manual permission checks, use the debugger and print statements in your own code.
- Authorization debugging info is now only present in the HTTP response body if `debug_authorization` is true.
- The format of authorization debug messages was improved.
- A new `BFG_DEBUG_NOTFOUND` envvar was added and a symmetric `debug_notfound` config file value was added. When either is true, and a `NotFound` response is returned by the BFG router (because a view could not be found), debugging information is printed to `stderr`. When this value is set true, the body of `HTTPNotFound` responses will also contain the same debugging information.
- `Allowed` and `Denied` responses from the security machinery are now specialized into two types: `ACL` types, and non-`ACL` types. The `ACL`-related responses are instances of `repoze.bfg.security.ACLAllowed` and `repoze.bfg.security.ACLDenied`. The non-`ACL`-related responses are `repoze.bfg.security.Allowed` and `repoze.bfg.security.Denied`. The `allowed`-type responses continue to evaluate equal to things that themselves evaluate equal to the `True` boolean, while the `denied`-type responses continue to evaluate equal to things that themselves evaluate equal to the `False` boolean. The only difference between the two types is the information attached to them for debugging purposes.
- Added a new `BFG_DEBUG_ALL` envvar and a symmetric `debug_all` config file value. When either is true, all other debug-related flags are set true unconditionally (e.g. `debug_notfound` and `debug_authorization`).

Documentation

- Added info about debug flag changes.
- Added a section to the security chapter named “Debugging Imperative Authorization Failures” (for e.g. `has_permission`).

Bug Fixes

- Change default paster template generator to use `Paste#http` server rather than `PasteScript#cherry` server. The cherry server has a security risk in it when `REMOTE_USER` is trusted by the downstream application.

0.4.1 (2008-10-28)

Bug Fixes

- If the `render_view_to_response` function was called, if the view was found and called, but it returned something that did not implement `IResponse`, the error would pass by unflagged. This was noticed when I created a view function that essentially returned `None`, but received a `NotFound` error rather than a `ValueError` when the view was rendered. This was fixed.

0.4.0 (2008-10-03)

Docs

- An “Environment and Configuration” chapter was added to the narrative portion of the documentation.

Features

- Ensure `bfg` doesn’t generate warnings when running under Python 2.6.
- The environment variable `BFG_RELOAD_TEMPLATES` is now available (serves the same purpose as `reload_templates` in the config file).
- A new configuration file option `debug_authorization` was added. This turns on printing of security authorization debug statements to `sys.stderr`. The `BFG_DEBUG_AUTHORIZATION` environment variable was also added; this performs the same duty.

Bug Fixes

- The environment variable `BFG_SECURITY_DEBUG` did not always work. It has been renamed to `BFG_DEBUG_AUTHORIZATION` and fixed.

Deprecations

- A deprecation warning is now issued when old API names from the `repoze.bfg.templates` module are imported.

Backwards incompatibilities

- The `BFG_SECURITY_DEBUG` environment variable was renamed to `BFG_DEBUG_AUTHORIZATION`.

0.3.9 (2008-08-27)

Features

- A `repoze.bfg.location` API module was added.

Backwards incompatibilities

- Applications must now use the `repoze.bfg.interfaces.ILocation` interface rather than `zope.location.interfaces.ILocation` to represent that a model object is “location-aware”. We’ve removed a dependency on `zope.location` for cleanliness purposes: as new versions of `zope` libraries are released which have improved dependency information, getting rid of our dependence on `zope.location` will prevent a newly installed `repoze.bfg` application from requiring the `zope.security`, `egg`, which not truly used at all in a “stock” `repoze.bfg` setup. These dependencies are still required by the stack at this time; this is purely a futureproofing move.

The security and model documentation for previous versions of `repoze.bfg` recommended using the `zope.location.interfaces.ILocation` interface to represent that a model object is “location-aware”. This documentation has been changed to reflect that this interface should now be imported from `repoze.bfg.interfaces.ILocation` instead.

0.3.8 (2008-08-26)

Docs

- Documented URL dispatch better in narrative form.

Bug fixes

- Routes URL dispatch did not have access to the WSGI environment, so conditions such as `method=GET` did not work.

Features

- Add `principals_allowed_by_permission` API to security module.
- Replace `z3c.pt` support with support for `chameleon.zpt`. Chameleon is the new name for the package that used to be named `z3c.pt`. NOTE: If you update a `repoze.bfg` SVN checkout that you're using for development, you will need to run “`setup.py install`” or “`setup.py develop`” again in order to obtain the proper Chameleon packages. `z3c.pt` is no longer supported by `repoze.bfg`. All API functions that used to render `z3c.pt` templates will work fine with the new packages, and your templates should render almost identically.
- Add a `repoze.bfg.chameleon_zpt` module. This module provides Chameleon ZPT support.
- Add a `repoze.bfg.xslt` module. This module provides XSLT support.
- Add a `repoze.bfg.chameleon_genshi` module. This provides direct Genshi support, which did not exist previously.

Deprecations

- Importing API functions directly from `repoze.bfg.template` is now deprecated. The `get_template`, `render_template`, `render_template_to_response` functions should now be imported from `repoze.chameleon_zpt`. The `render_transform`, and `render_transform_to_response` functions should now be imported from `repoze.bfg.xslt`. The `repoze.bfg.template` module will remain around “forever” to support backwards compatibility.

0.3.7 (2008-09-09)

Features

- Add compatibility with z3c.pt 1.0a7+ (z3c.pt became a namespace package).

Bug fixes

- `repoze.bfg.traversal.find_model` function did not function properly.

0.3.6 (2008-09-04)

Features

- Add startup process docs.
- Allow configuration cache to be bypassed by actions which include special “uncacheable” discriminators (for actions that have variable results).

Bug Fixes

- Move core `repoze.bfg` ZCML into a `repoze.bfg.includes` package so we can use `repoze.bfg` better as a namespace package. Adjust the code generator to use it. We’ve left around the `configure.zcml` in the `repoze.bfg` package directly so as not to break older apps.
- When a `zcml` application registry cache was unpickled, and it contained a reference to an object that no longer existed (such as a view), `bfg` would not start properly.

0.3.5 (2008-09-01)

Features

- Event notification is issued after application is created and configured (`IWSGIApplicationCreatedEvent`).
- New API module: `repoze.bfg.view`. This module contains the functions named `render_view_to_response`, `render_view_to_iterable`, `render_view` and `is_response`, which are documented in the API docs. These features aid programmatic (non-server-driven) view execution.

0.3.4 (2008-08-28)

Backwards incompatibilities

- Make `repoze.bfg` a namespace package so we can allow folks to create subpackages (e.g. `repoze.bfg.otherthing`) within separate eggs. This is a backwards incompatible change which makes it impossible to import “`make_app`” and “`get_options`” from the `repoze.bfg` module directly. This change will break all existing apps generated by the paster code generator. Instead, you need to import these functions as `repoze.bfg.router:make_app` and `repoze.bfg.registry:get_options`, respectively. Sorry folks, it has to be done now or never, and definitely better now.

Features

- Add `model_path` API function to traversal module.

Bugfixes

- Normalize path returned by `repoze.bfg.caller_path`.

0.3.3 (2008-08-23)

- Fix generated `test.py` module to use project name rather than package name.

0.3.2 (2008-08-23)

- Remove `sampleapp` sample application from `bfg` package itself.
- Remove dependency on `FormEncode` (only needed by `sampleapp`).
- Fix paster template generation so that case-sensitivity is preserved for project vs. package name.
- Depend on `z3c.pt` version 1.0a1 (which requires the `[lxml]` extra currently).
- Read and write a pickled ZCML actions list, stored as `configure.zcml.cache` next to the applications’s “normal” configuration file. A given `bfg` app will usually start faster if it’s able to read the pickle data. It fails gracefully to reading the real ZCML file if it cannot read the pickle.

0.3.1 (2008-08-20)

- Generated application differences: `make_app` entry point renamed to `app` in order to have a different name than the `bfg` function of the same name, to prevent confusion.
- Add “options” processing to `bfg`’s `make_app` to support runtime options. A new API function named `get_options` was added to the registry module. This function is typically used in an application’s `app` entry point. The Paste config file section for the app can now supply the `reload_templates` option, which, if true, will prevent the need to restart the appserver in order for `z3c.pt` or XSLT template changes to be detected.
- Use only the module name in generated project’s “test_suite” (run all tests found in the package).
- Default port for generated apps changed from 5432 to 6543 (Postgres default port is 6543).

0.3.0 (2008-08-16)

- Add `get_template` API to template module.

0.2.9 (2008-08-11)

- 0.2.8 was “brown bag” release. It didn’t work at all. Symptom: `ComponentLookupError` when trying to render a page.

0.2.8 (2008-08-11)

- Add `find_model` and `find_root` traversal APIs. In the process, make `ITraverser` a uni-adapter (on context) rather than a multiadapter (on context and request).

0.2.7 (2008-08-05)

- Add a `request_type` attribute to the available attributes of a `bfg:view` `configure.zcml` element. This attribute will have a value which is a dotted Python path, pointing at an interface. If the request object implements this interface when the view lookup is performed, the appropriate view will be called. This is meant to allow for simple “skinning” of sites based on request type. An event subscriber should attach the interface to the request on ingress to support skins.
- Remove “template only” views. These were just confusing and were never documented.
- Small url dispatch overhaul: the `connect` method of the `urldispatch.RoutesMapper` object now accepts a keyword parameter named `context_factory`. If this parameter is supplied, it must be a callable which returns an instance. This instance is used as the context for the request when a route is matched.
- The registration of a `RoutesModelTraverser` no longer needs to be performed by the application; it’s in the `bfg` ZCML now.

0.2.6 (2008-07-31)

- Add event sends for `INewRequest` and `INewResponse`. See the `events.rst` chapter in the documentation's `api` directory.

0.2.5 (2008-07-28)

- Add `model_url` API.

0.2.4 (2008-07-27)

- Added url-based dispatch.

0.2.3 (2008-07-20)

- Add API functions for `authenticated_userid` and `effective_principals`.

0.2.2 (2008-07-20)

- Add `authenticated_userid` and `effective_principals` API to security policy.

0.2.1 (2008-07-20)

- Add `find_interface` API.

0.2 (2008-07-19)

- Add `wsgiapp` decorator.
- The concept of “view factories” was removed in favor of always calling a view, which is a callable that returns a response directly (as opposed to returning a view). As a result, the `factory` attribute in the `bfg:view` ZCML statement has been renamed to `view`. Various interface names were changed also.
- `render_template` and `render_transform` no longer return a `Response` object. Instead, these return strings. The old behavior can be obtained by using `render_template_to_response` and `render_transform_to_response`.
- Added ‘`repoze.bfg.push:pushpage`’ decorator, which creates BFG views from callables which take (context, request) and return a mapping of top-level names.
- Added ACL-based security.
- Support for XSLT templates via a `render_transform` method

0.1 (2008-07-08)

- Initial release.

Glossary and Index

Glossary

ACE An *access control entry*. An access control entry is one element in an *ACL*. An access control entry is a three-tuple that describes three things: an *action* (one of either `Allow` or `Deny`), a *principal* (a string describing a user or group), and a *permission*. For example the ACE, (`Allow`, `'bob'`, `'read'`) is a member of an ACL that indicates that the principal bob is allowed the permission `read` against the resource the ACL is attached to.

ACL An *access control list*. An ACL is a sequence of *ACE* tuples. An ACL is attached to a resource instance. An example of an ACL is [(`Allow`, `'bob'`, `'read'`), (`Deny`, `'fred'`, `'write'`)]. If an ACL is attached to a resource instance, and that resource is findable via the context resource, it will be consulted any active security policy to determine whether a particular request can be fulfilled given the *authentication* information in the request.

action Represents a pending configuration statement generated by a call to a *configuration directive*. The set of pending configuration actions are processed when `pyramid.config.Configurator.commit()` is called.

add-on A Python *distribution* that uses Pyramid's extensibility to plug into a Pyramid application and provide extra, configurable services.

Agendaless Consulting A consulting organization formed by Paul Everitt, Tres Seaver, and Chris McDonough.

See also:

See also Agendaless Consulting.

Akhet Akhet is a Pyramid library and demo application with a Pylons-like feel. It's most known for its former application scaffold, which helped users transition from Pylons and those preferring a more Pylons-like API. The scaffold has been retired but the demo plays a similar role.

application registry A registry of configuration information consulted by Pyramid while servicing an application. An application registry maps resource types to views, as well as housing other application-specific component registrations. Every Pyramid application has one (and only one) application registry.

asset Any file contained within a Python *package* which is *not* a Python source code file.

asset descriptor An instance representing an *asset specification* provided by the `pyramid.path.AssetResolver.resolve()` method. It supports the methods and attributes documented in `pyramid.interfaces.IAssetDescriptor`.

asset specification A colon-delimited identifier for an *asset*. The colon separates a Python *package* name from a package subpath. For example, the asset specification `my.package:static/baz.css` identifies the file named `baz.css` in the `static` subdirectory of the `my.package` Python *package*. See *Understanding Asset Specifications* for more info.

authentication The act of determining that the credentials a user presents during a particular request are “good”. Authentication in Pyramid is performed via an *authentication policy*.

authentication policy An authentication policy in Pyramid terms is a bit of code which has an API which determines the current *principal* (or principals) associated with a request.

authorization The act of determining whether a user can perform a specific action. In pyramid terms, this means determining whether, for a given resource, any *principal* (or principals) associated with the request have the requisite *permission* to allow the request to continue. Authorization in Pyramid is performed via its *authorization policy*.

authorization policy An authorization policy in Pyramid terms is a bit of code which has an API which determines whether or not the principals associated with the request can perform an action associated with a permission, based on the information found on the *context* resource.

Babel A collection of tools for internationalizing Python applications. Pyramid does not depend on Babel to operate, but if Babel is installed, additional locale functionality becomes available to your application.

cache busting A technique used when serving a cacheable static asset in order to force a client to query the new version of the asset. See *Cache Busting* for more information.

Chameleon chameleon is an attribute language template compiler which supports the *ZPT* templating specification. It is written and maintained by Malthe Borch. It has several extensions, such as the ability to use bracketed (Mako-style) `#{name}` syntax. It is also much faster than the reference implementation of ZPT. Pyramid offers Chameleon templating out of the box in ZPT and text flavors.

configuration declaration An individual method call made to a *configuration directive*, such as registering a *view configuration* (via the `add_view()` method of the configurator) or *route configuration* (via the `add_route()` method of the configurator). A set of configuration declarations is also implied by the *configuration decoration* detected by a *scan* of code in a package.

configuration decoration Metadata implying one or more *configuration declaration* invocations. Often set by configuration Python *decorator* attributes, such as `pyramid.view.view_config`, aka `@view_config`.

configuration directive A method of the *Configurator* which causes a configuration action to occur. The method `pyramid.config.Configurator.add_view()` is a configuration directive, and application developers can add their own directives as necessary (see *Adding Methods to the Configurator via add_directive*).

configurator An object used to do *configuration declaration* within an application. The most common configurator is an instance of the `pyramid.config.Configurator` class.

conflict resolution Pyramid attempts to resolve ambiguous configuration statements made by application developers via automatic conflict resolution. Automatic conflict resolution is described in *Automatic Conflict Resolution*. If Pyramid cannot resolve ambiguous configuration statements, it is possible to manually resolve them as described in *Manually Resolving Conflicts*.

console script A script written to the `bin` (on UNIX, or `Scripts` on Windows) directory of a Python installation or *virtual environment* as the result of running `pip install` or `pip install -e ..`.

context A resource in the resource tree that is found during *traversal* or *URL dispatch* based on URL data; if it's found via traversal, it's usually a *resource* object that is part of a resource tree; if it's found via *URL dispatch*, it's an object manufactured on behalf of the route's "factory". A context resource becomes the subject of a *view*, and often has security information attached to it. See the *Traversal* chapter and the *URL Dispatch* chapter for more information about how a URL is resolved to a context resource.

CPython The C implementation of the Python language. This is the reference implementation that most people refer to as simply "Python"; *Jython*, Google's App Engine, and PyPy are examples of non-C based Python implementations.

declarative configuration The configuration mode in which you use the combination of *configuration decoration* and a *scan* to configure your Pyramid application.

decorator A wrapper around a Python function or class which accepts the function or class as its first argument and which returns an arbitrary object. Pyramid provides several decorators, used for configuration and return value modification purposes.

See also:

See also PEP 318.

Default Locale Name The *locale name* used by an application when no explicit locale name is set. See *Localization-Related Deployment Settings*.

default permission A *permission* which is registered as the default for an entire application. When a default permission is in effect, every *view configuration* registered with the system will be effectively amended with a `permission` argument that will require that the executing user possess the default permission in order to successfully execute the associated *view callable*.

See also:

See also *Setting a Default Permission*.

default root factory If an application does not register a *root factory* at Pyramid configuration time, a *default* root factory is used to create the default root object. Use of the default root object is useful in application which use *URL dispatch* for all URL-to-view code mappings, and does not (knowingly) use traversal otherwise.

Default view The default view of a *resource* is the view invoked when the *view name* is the empty string (`' '`). This is the case when *traversal* exhausts the path elements in the `PATH_INFO` of a request before it returns a *context* resource.

Deployment settings Deployment settings are settings passed to the *Configurator* as a `settings` argument. These are later accessible via a `request.registry.settings` dictionary in views or as `config.registry.settings` in configuration code. Deployment settings can be used as global application values.

discriminator The unique identifier of an *action*.

distribute Distribute is a fork of *setuptools* which runs on both Python 2 and Python 3.

distribution (Setuptools/distutils terminology). A file representing an installable library or application. Distributions are usually files that have the suffix of `.egg`, `.tar.gz`, or `.zip`. Distributions are the target of Setuptools-related commands such as `easy_install`.

distutils The standard system for packaging and distributing Python packages. See <https://docs.python.org/2/distutils/index.html> for more information. *setuptools* is actually an *extension* of the Distutils.

Django A full-featured Python web framework.

domain model Persistent data related to your application. For example, data stored in a relational database. In some applications, the *resource tree* acts as the domain model.

- dotted Python name** A reference to a Python object by name using a string, in the form `path.to.module:attribute`. Often used in Pyramid and setuptools configurations. A variant is used in dotted names within configurator method arguments that name objects (such as the “add_view” method’s “view” and “context” attributes): the colon (:) is not used; in its place is a dot.
- entry point** A *setuptools* indirection, defined within a setuptools *distribution* `setup.py`. It is usually a name which refers to a function somewhere in a package which is held by the distribution.
- event** An object broadcast to zero or more *subscriber* callables during normal Pyramid system operations during the lifetime of an application. Application code can subscribe to these events by using the subscriber functionality described in *Using Events*.
- exception response** A *response* that is generated as the result of a raised exception being caught by an *exception view*.
- Exception view** An exception view is a *view callable* which may be invoked by Pyramid when an exception is raised during request processing. See *Custom Exception Views* for more information.
- falsey string** A string representing a value of `False`. Acceptable values are `f`, `false`, `n`, `no`, `off` and `0`.
- finished callback** A user-defined callback executed by the *router* unconditionally at the very end of request processing. See *Using Finished Callbacks*.
- Forbidden view** An *exception view* invoked by Pyramid when the developer explicitly raises a *pyramid.httpexceptions.HTTPForbidden* exception from within *view* code or *root factory* code, or when the *view configuration* and *authorization policy* found for a request disallows a particular view invocation. Pyramid provides a default implementation of a forbidden view; it can be overridden. See *Changing the Forbidden View*.
- Genshi** An XML templating language by Christopher Lenz.
- Gettext** The GNU gettext library, used by the Pyramid translation machinery.
- Google App Engine** Google App Engine (aka “GAE”) is a Python application hosting service offered by Google. Pyramid runs on GAE.
- Green Unicorn** Aka *gunicorn*, a fast *WSGI* server that runs on UNIX under Python 2.6+ or Python 3.1+. See <http://gunicorn.org/> for detailed information.
- Grok** A web framework based on Zope 3.

HTTP Exception The set of exception classes defined in `pyramid.httpexceptions`. These can be used to generate responses with various status codes when raised or returned from a *view callable*.

See also:

See also *HTTP Exceptions*.

imperative configuration The configuration mode in which you use Python to call methods on a *Configurator* in order to add each *configuration declaration* required by your application.

interface A Zope interface object. In Pyramid, an interface may be attached to a *resource* object or a *request* object in order to identify that the object is “of a type”. Interfaces are used internally by Pyramid to perform view lookups and other policy lookups. The ability to make use of an interface is exposed to an application programmers during *view configuration* via the `context` argument, the `request_type` argument and the `containment` argument. Interfaces are also exposed to application developers when they make use of the *event* system. Fundamentally, Pyramid programmers can think of an interface as something that they can attach to an object that stamps it with a “type” unrelated to its underlying Python type. Interfaces can also be used to describe the behavior of an object (its methods and attributes), but unless they choose to, Pyramid programmers do not need to understand or use this feature of interfaces.

Internationalization The act of creating software with a user interface that can potentially be displayed in more than one language or cultural context. Often shortened to “i18n” (because the word “internationalization” is I, 18 letters, then N).

See also:

See also *Localization*.

introspectable An object which implements the attributes and methods described in `pyramid.interfaces.IIntrospectable`. Introspectables are used by the *introspector* to display configuration information about a running Pyramid application. An introspectable is associated with a *action* by virtue of the `pyramid.config.Configurator.action()` method.

introspector An object with the methods described by `pyramid.interfaces.IIntrospector` that is available in both configuration code (for registration) and at runtime (for querying) that allows a developer to introspect configuration statements and relationships between those statements.

Jinja2 A text templating language by Armin Ronacher.

jQuery A popular Javascript library.

JSON JavaScript Object Notation is a data serialization format.

Jython A Python implementation written for the Java Virtual Machine.

lineage An ordered sequence of objects based on a “*location* -aware” resource. The lineage of any given *resource* is composed of itself, its parent, its parent’s parent, and so on. The order of the sequence is resource-first, then the parent of the resource, then its parent’s parent, and so on. The parent of a resource in a lineage is available as its `__parent__` attribute.

Lingua A package by Wichert Akkerman which provides the `pot-create` command to extract translatable messages from Python sources and Chameleon ZPT template files.

Locale Name A string like `en`, `en_US`, `de`, or `de_AT` which uniquely identifies a particular locale.

Locale Negotiator An object supplying a policy determining which *locale name* best represents a given *request*. It is used by the `pyramid.i18n.get_locale_name()`, and `pyramid.i18n.negotiate_locale_name()` functions, and indirectly by `pyramid.i18n.get_localizer()`. The `pyramid.i18n.default_locale_negotiator()` function is an example of a locale negotiator.

Localization The process of displaying the user interface of an internationalized application in a particular language or cultural context. Often shortened to “l10” (because the word “localization” is L, 10 letters, then N).

See also:

See also *Internationalization*.

Localizer An instance of the class `pyramid.i18n.Localizer` which provides translation and pluralization services to an application. It is retrieved via the `pyramid.i18n.get_localizer()` function.

location The path to an object in a *resource tree*. See *Location-Aware Resources* for more information about how to make a resource object *location-aware*.

Mako Mako is a template language which refines the familiar ideas of componentized layout and inheritance using Python with Python scoping and calling semantics.

matchdict The dictionary attached to the *request* object as `request.matchdict` when a *URL dispatch* route has been matched. Its keys are names as identified within the route pattern; its values are the values matched by each pattern name.

Message Catalog A *gettext* `.mo` file containing translations.

Message Identifier A string used as a translation lookup key during localization. The `msgid` argument to a *translation string* is a message identifier. Message identifiers are also present in a *message catalog*.

METAL Macro Expansion for TAL, a part of *ZPT* which makes it possible to share common look and feel between templates.

middleware *Middleware* is a *WSGI* concept. It is a *WSGI* component that acts both as a server and an application. Interesting uses for middleware exist, such as caching, content-transport encoding, and other functions. See WSGI.org or PyPI to find middleware for your application.

mod_wsgi `mod_wsgi` is an Apache module developed by Graham Dumpleton. It allows *WSGI* applications (such as applications developed using *Pyramid*) to be served using the Apache web server.

module A Python source file; a file on the filesystem that typically ends with the extension `.py` or `.pyc`. Modules often live in a *package*.

multidict An ordered dictionary that can have multiple values for each key. Adds the methods `getall`, `getone`, `mixed`, `add` and `dict_of_lists` to the normal dictionary interface. See *Multidict* and *pyramid.interfaces.IMultiDict*.

Not Found View An *exception view* invoked by *Pyramid* when the developer explicitly raises a *pyramid.httpexceptions.HTTPNotFound* exception from within *view* code or *root factory* code, or when the current request doesn't match any *view configuration*. *Pyramid* provides a default implementation of a Not Found View; it can be overridden. See *Changing the Not Found View*.

package A directory on disk which contains an `__init__.py` file, making it recognizable to Python as a location which can be `import`-ed. A package exists to contain *module* files.

PasteDeploy *PasteDeploy* is a library used by *Pyramid* which makes it possible to configure *WSGI* components together declaratively within an `.ini` file. It was developed by Ian Bicking.

permission A string or Unicode object that represents an action being taken against a *context* resource. A permission is associated with a view name and a resource type by the developer. Resources are decorated with security declarations (e.g. an *ACL*), which reference these tokens also. Permissions are used by the active security policy to match the view permission against the resources's statements about which permissions are granted to which principal in a context in order to answer the question "is this user allowed to do this". Examples of permissions: `read`, or `view_blog_entries`.

physical path The path required by a traversal which resolve a *resource* starting from the *physical root*. For example, the physical path of the `abc` subobject of the physical root object is `/abc`. Physical paths can also be specified as tuples where the first element is the empty string (representing the root), and every other element is a Unicode object, e.g. `(' ', 'abc')`. Physical paths are also sometimes called "traversal paths".

physical root The object returned by the application *root factory*. Unlike the *virtual root* of a request, it is not impacted by *Virtual Hosting*: it will always be the actual object returned by the root factory, never a subobject.

pip The *Python Packaging Authority*’s recommended tool for installing Python packages.

pipeline The *PasteDeploy* term for a single configuration of a WSGI server, a WSGI application, with a set of *middleware* in-between.

pkg_resources A module which ships with *setuptools* and *distribute* that provides an API for addressing “asset files” within a Python *package*. Asset files are static files, template files, etc; basically anything non-Python-source that lives in a Python package can be considered a asset file.

See also:

See also `PkgResources`.

predicate A test which returns `True` or `False`. Two different types of predicates exist in Pyramid: a *view predicate* and a *route predicate*. View predicates are attached to *view configuration* and route predicates are attached to *route configuration*.

predicate factory A callable which is used by a third party during the registration of a route, view, or subscriber predicates to extend the configuration system. See *Adding a Third Party View, Route, or Subscriber Predicate* for more information.

pregenerator A pregenerator is a function associated by a developer with a *route*. It is called by `route_url()` in order to adjust the set of arguments passed to it by the user for special purposes. It will influence the URL returned by `route_url()`. See `pyramid.interfaces.IRoutePregenerator` for more information.

principal A *principal* is a string or Unicode object representing an entity, typically a user or group. Principals are provided by an *authentication policy*. For example, if a user has the *userid bob*, and is a member of two groups named *group foo* and *group bar*, then the request might have information attached to it indicating that Bob was represented by three principals: *bob*, *group foo* and *group bar*.

project (Setuptools/distutils terminology). A directory on disk which contains a `setup.py` file and one or more Python packages. The `setup.py` file contains code that allows the package(s) to be installed, distributed, and tested.

Pylons A lightweight Python web framework and a predecessor of Pyramid.

PyPI The Python Package Index, a collection of software available for Python.

PyPy PyPy is an “alternative implementation of the Python language”: <http://pypy.org/>

Pyramid Community Cookbook Additional, community-based documentation for Pyramid which presents topical, practical uses of Pyramid: [Pyramid Community Cookbook](#)

pyramid_debugtoolbar A Pyramid add-on which displays a helpful debug toolbar “on top of” HTML pages rendered by your application, displaying request, routing, and database information. `pyramid_debugtoolbar` is configured into the `development.ini` of all applications which use a Pyramid *scaffold*. For more information, see http://docs.pylonsproject.org/projects/pyramid_debugtoolbar/en/latest/.

pyramid_exclog A package which logs Pyramid application exception (error) information to a standard Python logger. This add-on is most useful when used in production applications, because the logger can be configured to log to a file, to UNIX syslog, to the Windows Event Log, or even to email. See its documentation.

pyramid_handlers An add-on package which allows Pyramid users to create classes that are analogues of Pylons 1 “controllers”. See http://docs.pylonsproject.org/projects/pyramid_handlers/en/latest/.

pyramid_jinja2 *Jinja2* templating system bindings for Pyramid, documented at http://docs.pylonsproject.org/projects/pyramid_jinja2/en/latest/. This package also includes a scaffold named `pyramid_jinja2_starter`, which creates an application package based on the Jinja2 templating system.

pyramid_redis_sessions A package by Eric Rasmussen which allows you to store Pyramid session data in a Redis database. See https://pypi.python.org/pypi/pyramid_redis_sessions for more information.

pyramid_zcml An add-on package to Pyramid which allows applications to be configured via *ZCML*. It is available on *PyPI*. If you use `pyramid_zcml`, you can use *ZCML* as an alternative to *imperative configuration* or *configuration decoration*.

Python The programming language in which Pyramid is written.

Python Packaging Authority The Python Packaging Authority (PyPA) is a working group that maintains many of the relevant projects in Python packaging.

pyvenv The *Python Packaging Authority* formerly recommended using the `pyvenv` command for creating virtual environments on Python 3.4 and 3.5, but it was deprecated in 3.6 in favor of `python3 -m venv` on UNIX or `python -m venv` on Windows, which is backward compatible on Python 3.3 and greater.

renderer A serializer which converts non-*Response* return values from a *view* into a string, and ultimately into a response, usually through *view configuration*. Using a renderer can make writing views that require templating or other serialization, like JSON, less tedious. See *Writing View Callables Which Use a Renderer* for more information.

renderer factory A factory which creates a *renderer*. See *Adding and Changing Renderers* for more information.

renderer globals Values injected as names into a renderer by a `pyramid.event.BeforeRender` event.

Repoze “Repoze” is essentially a “brand” of software developed by Agendaless Consulting and a set of contributors. The term has no special intrinsic meaning. The project’s website has more information. The software developed “under the brand” is available in a Subversion repository. Pyramid was originally known as `repoze.bfg`.

repoze.catalog An indexing and search facility (fielded and full-text) based on `zope.index`. See the documentation for more information.

repoze.lemonade Zope2 CMF-like data structures and helper facilities for CA-and-ZODB-based applications useful within Pyramid applications.

repoze.who Authentication middleware for *WSGI* applications. It can be used by Pyramid to provide authentication information.

repoze.workflow Barebones workflow for Python apps . It can be used by Pyramid to form a workflow system.

request An object that represents an HTTP request, usually an instance of the `pyramid.request.Request` class. See *Request and Response Objects* (narrative) and `pyramid.request` (API documentation) for information about request objects.

request factory An object which, provided a *WSGI* environment as a single positional argument, returns a Pyramid-compatible request.

request type An attribute of a *request* that allows for specialization of view invocation based on arbitrary categorization. The every *request* object that Pyramid generates and manipulates has one or more *interface* objects attached to it. The default interface attached to a request object is `pyramid.interfaces.IRequest`.

resource An object representing a node in the *resource tree* of an application. If *traversal* is used, a resource is an element in the resource tree traversed by the system. When traversal is used, a resource becomes the *context* of a *view*. If *url dispatch* is used, a single resource is generated for each request and is used as the context resource of a view.

Resource Location The act of locating a *context* resource given a *request*. *Traversal* and *URL dispatch* are the resource location subsystems used by Pyramid.

resource tree A nested set of dictionary-like objects, each of which is a *resource*. The act of *traversal* uses the resource tree to find a *context* resource.

response An object returned by a *view callable* that represents response data returned to the requesting user agent. It must implement the `pyramid.interfaces.IResponse` interface. A response object is typically an instance of the `pyramid.response.Response` class or a subclass such as `pyramid.httpexceptions.HTTPFound`. See *Request and Response Objects* for information about response objects.

response adapter A callable which accepts an arbitrary object and “converts” it to a `pyramid.response.Response` object. See *Changing How Pyramid Treats View Responses* for more information.

response callback A user-defined callback executed by the *router* at a point after a *response* object is successfully created.

See also:

See also *Using Response Callbacks*.

response factory An object which, provided a *request* as a single positional argument, returns a Pyramid-compatible response. See `pyramid.interfaces.IResponseFactory`.

reStructuredText A plain text markup format that is the defacto standard for documenting Python projects. The Pyramid documentation is written in reStructuredText.

root The object at which *traversal* begins when Pyramid searches for a *context* resource (for *URL Dispatch*, the root is *always* the context resource unless the `traverse=` argument is used in route configuration).

root factory The “root factory” of a Pyramid application is called on every request sent to the application. The root factory returns the traversal root of an application. It is conventionally named `get_root`. An application may supply a root factory to Pyramid during the construction of a *Configurator*. If a root factory is not supplied, the application creates a default root object using the *default root factory*.

route A single pattern matched by the *url dispatch* subsystem, which generally resolves to a *root factory* (and then ultimately a *view*).

See also:

See also *url dispatch*.

route configuration Route configuration is the act of associating request parameters with a particular *route* using pattern matching and *route predicate* statements. See *URL Dispatch* for more information about route configuration.

route predicate An argument to a *route configuration* which implies a value that evaluates to `True` or `False` for a given *request*. All predicates attached to a *route configuration* must evaluate to `True` for the associated route to “match” the current request. If a route does not match the current request, the next route (in definition order) is attempted.

router The *WSGI* application created when you start a Pyramid application. The router intercepts requests, invokes traversal and/or URL dispatch, calls view functions, and returns responses to the *WSGI* server on behalf of your Pyramid application.

Routes A system by Ben Bangert which parses URLs and compares them against a number of user defined mappings. The URL pattern matching syntax in Pyramid is inspired by the Routes syntax (which was inspired by Ruby On Rails pattern syntax).

routes mapper An object which compares path information from a request to an ordered set of route patterns. See *URL Dispatch*.

scaffold A project template that generates some of the major parts of a Pyramid application and helps users to quickly get started writing larger applications. Scaffolds are usually used via the `pcreate` command.

scan The term used by Pyramid to define the process of importing and examining all code in a Python package or module for *configuration decoration*.

session A namespace that is valid for some period of continual activity that can be used to represent a user’s interaction with a web application.

session factory A callable, which, when called with a single argument named `request` (a *request* object), returns a *session* object. See *Using the Default Session Factory*, *Using Alternate Session Factories* and `pyramid.config.Configurator.set_session_factory()` for more information.

setuptools Setuptools builds on Python’s `distutils` to provide easier building, distribution, and installation of libraries and applications. As of this writing, setuptools runs under Python 2, but not under Python 3. You can use *distribute* under Python 3 instead.

SQLAlchemy SQLAlchemy is an object relational mapper used in tutorials within this documentation.

subpath A list of element “left over” after the *router* has performed a successful traversal to a view. The subpath is a sequence of strings, e.g. `['left', 'over', 'names']`. Within Pyramid applications that use URL dispatch rather than traversal, you can use `*subpath` in the route pattern to influence the subpath. See *Using *subpath in a Route Pattern* for more information.

subscriber A callable which receives an *event*. A callable becomes a subscriber via *imperative configuration* or via *configuration decoration*. See *Using Events* for more information.

template A file with replaceable parts that is capable of representing some text, XML, or HTML when rendered.

thread local A thread-local variable is one which is essentially a global variable in terms of how it is accessed and treated, however, each thread used by the application may have a different value for this same “global” variable. Pyramid uses a small number of thread local variables, as described in *Thread Locals*.

See also:

See also the `stdlib` documentation for more information.

Translation Context A string representing the “context” in which a translation was made within a given *translation domain*. See the `gettext` documentation, 11.2.5 Using contexts for solving ambiguities for more information.

Translation Directory A translation directory is a *gettext* translation directory. It contains language folders, which themselves contain `LC_MESSAGES` folders, which contain `.mo` files. Each `.mo` file represents a set of translations for a language in a *translation domain*. The name of the `.mo` file (minus the `.mo` extension) is the translation domain name.

Translation Domain A string representing the “context” in which a translation was made. For example the word “java” might be translated differently if the translation domain is “programming-languages” than would be if the translation domain was “coffee”. A translation domain is represented by a collection of `.mo` files within one or more *translation directory* directories.

Translation String An instance of `pyramid.i18n.TranslationString`, which is a class that behaves like a Unicode string, but has several extra attributes such as `domain`, `msgid`, and `mapping` for use during translation. Translation strings are usually created by hand within software, but are sometimes created on the behalf of the system for automatic template translation. For more information, see *Internationalization and Localization*.

Translator A callable which receives a *translation string* and returns a translated Unicode object for the purposes of internationalization. A *localizer* supplies a translator to a Pyramid application accessible via its `translate` method.

traversal The act of descending “up” a tree of resource objects from a root resource in order to find a *context* resource. The Pyramid *router* performs traversal of resource objects when a *root factory* is specified. See the *Traversal* chapter for more information. Traversal can be performed *instead* of *URL dispatch* or can be combined *with* *URL dispatch*. See *Combining Traversal and URL Dispatch* for more information about combining traversal and *URL dispatch* (advanced).

truthy string A string representing a value of `True`. Acceptable values are `t`, `true`, `y`, `yes`, `on` and `1`.

tween A bit of code that sits between the Pyramid router’s main request handling function and the upstream WSGI component that uses Pyramid as its ‘app’. The word “tween” is a contraction of “between”. A tween may be used by Pyramid framework extensions, to provide, for example, Pyramid-specific view timing support, bookkeeping code that examines exceptions before they are returned to the upstream WSGI application, or a variety of other features. Tweens behave a bit like *WSGI middleware* but they have the benefit of running in a context in which they have access to the Pyramid *application registry* as well as the Pyramid rendering machinery. See *Registering Tweens*.

URL dispatch An alternative to *traversal* as a mechanism for locating a *context* resource for a *view*. When you use a *route* in your Pyramid application via a *route configuration*, you are using URL dispatch. See the *URL Dispatch* for more information.

userid A *userid* is a string or Unicode object used to identify and authenticate a real-world user or client. A *userid* is supplied to an *authentication policy* in order to discover the user’s *principals*. In the authentication policies which Pyramid provides, the default behavior returns the user’s *userid* as a principal, but this is not strictly necessary in custom policies that define their principals differently.

Venusian Venusian is a library which allows framework authors to defer decorator actions. Instead of taking actions when a function (or class) decorator is executed at import time, the action usually taken by the decorator is deferred until a separate “scan” phase. Pyramid relies on Venusian to provide a basis for its *scan* feature.

venv The *Python Packaging Authority*’s recommended tool for creating virtual environments on Python 3.3 and greater.

Note: whenever you encounter commands prefixed with `$VENV` (Unix) or `%VENV` (Windows), know that that is the environment variable whose value is the root of the virtual environment in question.

view Common vernacular for a *view callable*.

view callable A “view callable” is a callable Python object which is associated with a *view configuration*; it returns a *response* object. A view callable accepts a single argument: *request*, which will be an instance of a *request* object. An alternate calling convention allows a view to be defined as a callable which accepts a pair of arguments: *context* and *request*; this calling convention is useful for traversal-based applications in which a *context* is always very important. A view callable is the primary mechanism by which a developer writes user interface code within Pyramid. See *Views* for more information about Pyramid view callables.

view configuration View configuration is the act of associating a *view callable* with configuration information. This configuration information helps map a given *request* to a particular view callable and it can influence the response of a view callable. Pyramid views can be configured via *imperative configuration*, or by a special `@view_config` decorator coupled with a *scan*. See *View Configuration* for more information about view configuration.

view deriver A view deriver is a composable component of the view pipeline which is used to create a *view callable*. A view deriver is a callable implementing the `pyramid.interfaces.IViewDeriver` interface. Examples of built-in derivers including view mapper, the permission checker, and applying a renderer to a dictionary returned from the view.

View handler A view handler ties together `pyramid.config.Configurator.add_route()` and `pyramid.config.Configurator.add_view()` to make it more convenient to register a collection of views as a single class when using *url dispatch*. View handlers ship as part of the `pyramid_handlers` add-on package.

View Lookup The act of finding and invoking the “best” *view callable*, given a *request* and a *context* resource.

view mapper A view mapper is a class which implements the `pyramid.interfaces.IViewMapperFactory` interface, which performs view argument and return value mapping. This is a plug point for extension builders, not normally used by “civilians”.

view name The “URL name” of a view, e.g `index.html`. If a view is configured without a name, its name is considered to be the empty string (which implies the *default view*).

view predicate An argument to a *view configuration* which evaluates to `True` or `False` for a given *request*. All predicates attached to a view configuration must evaluate to `true` for the associated view to be considered as a possible callable for a given request.

virtual environment An isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.

virtual root A resource object representing the “virtual” root of a request; this is typically the *physical root* object unless *Virtual Hosting* is in use.

virtualenv The `virtualenv` tool that allows one to create virtual environments. In Python 3.3 and greater, `venv` is the preferred tool.

Note: whenever you encounter commands prefixed with `$VENV` (Unix) or `%VENV` (Windows), know that that is the environment variable whose value is the root of the virtual environment in question.

Waitress A *WSGI* server that runs on UNIX and Windows under Python 2.6+ and Python 3.2+. Projects generated via Pyramid scaffolding use Waitress as a *WSGI* server. See <http://docs.pylonsproject.org/projects/waitress/en/latest/> for detailed information.

WebOb WebOb is a *WSGI* request/response library created by Ian Bicking.

WebTest WebTest is a package which can help you write functional tests for your *WSGI* application.

WSGI Web Server Gateway Interface. This is a Python standard for connecting web applications to web servers, similar to the concept of Java Servlets. Pyramid requires that your application be served as a WSGI application.

ZCML Zope Configuration Markup Language, an XML dialect used by Zope and *pyramid_zcml* for configuration tasks.

ZODB Zope Object Database, a persistent Python object store.

Zope The Z Object Publishing Framework, a full-featured Python web framework.

Zope Component Architecture The Zope Component Architecture (aka ZCA) is a system which allows for application pluggability and complex dispatching based on objects which implement an *interface*. Pyramid uses the ZCA “under the hood” to perform view dispatching and other application configuration tasks.

ZPT The Zope Page Template templating language.

Symbols

- *subpath
 - hybrid applications, 584
- *traverse route pattern
 - hybrid applications, 580
- .ini
 - logging, 477
 - middleware, 482
 - settings, 350
- \$VENV/bin/pip vs. source bin/activate, 318
- __call__() (ICacheBuster method), 773
- __call__() (IRenderer method), 761
- __call__() (IRendererFactory method), 761
- __call__() (IRequestFactory method), 761
- __call__() (IResponse method), 766
- __call__() (IResponseFactory method), 761
- __call__() (IRoutePregenerator method), 758
- __call__() (ISessionFactory method), 760
- __call__() (IViewDeriver method), 773
- __call__() (IViewMapper method), 762
- __call__() (IViewMapperFactory method), 762
- __contains__() (IDict method), 762
- __delitem__() (IDict method), 762
- __getitem__() (IDict method), 763
- __hash__() (IIntrospectable method), 769
- __init__.py, 344
- __iter__() (IDict method), 762
- __setitem__() (IDict method), 762
- __str__() (IActionInfo method), 771

A

- absolute_asset_spec() (Configurator method), 726
- abspath() (IAssetDescriptor method), 772
- absspec() (IAssetDescriptor method), 772
- accept (Request attribute), 804
- accept_charset (Request attribute), 804
- accept_encoding (Request attribute), 805
- accept_language (Request attribute), 805
- accept_ranges (IResponse attribute), 765
- accept_ranges (Response attribute), 813
- access control entry, 568
- access control list, 567
- ACE, 568, **1157**
- ACE (special), 571
- ACL, 567, **1157**
 - resource, 567
- ACL inheritance, 571
- ACLAllowed (class in pyramid.security), 825
- ACLAuthorizationPolicy (class in pyramid.authorization), 684
- ACLDenied (class in pyramid.security), 825
- action, **1157**
- action() (Configurator method), 724
- action_info (IIntrospectable attribute), 768
- activating
 - translation, 521
- add() (IIntrospector method), 771
- add() (IMultiDict method), 764
- add-on, **1157**

- add_adapter() (JSON method), 784
- add_adapter() (JSONP method), 785
- add_directive, 650
- add_directive() (Configurator method), 724
- add_finished_callback() (Request method), 792
- add_forbidden_view() (Configurator method), 709
- add_notfound_view() (Configurator method), 708
- add_permission() (Configurator method), 713
- add_renderer() (Configurator method), 716
- add_request_method() (Configurator method), 713
- add_resource_url_adapter() (Configurator method), 716
- add_response_adapter() (Configurator method), 717
- add_response_callback() (Request method), 791
- add_route, 360
- add_route() (Configurator method), 694
- add_route_predicate() (Configurator method), 720
- add_settings() (Configurator method), 715
- add_static_view, 432
- add_static_view() (Configurator method), 698
- add_subscriber() (Configurator method), 710
- add_subscriber_predicate() (Configurator method), 720
- add_translation_dirs() (Configurator method), 714
- add_traverser() (Configurator method), 717
- add_tween() (Configurator method), 718
- add_view, 425
- add_view() (Configurator method), 699
- add_view_deriver() (Configurator method), 721
- add_view_predicate() (Configurator method), 721
- adding
 - renderer, 401
 - translation directory, 521
- adding directives
 - configurator, 650
- adding renderer globals, 599
- advanced
 - configuration, 642
- age (IResponse attribute), 767
- age (Response attribute), 813
- Agendaless Consulting, 295, **1157**
- Akhet, **1157**
- Akkerman, Wichert, 7
- alchemy scaffold, 327
- ALL_PERMISSIONS (in module pyramid.security), 825
- Allow (in module pyramid.security), 825
- allow (IResponse attribute), 767
- allow (Response attribute), 813
- Allowed (class in pyramid.security), 826
- app (IApplicationCreated attribute), 754
- app_iter (IResponse attribute), 766
- app_iter (Response attribute), 813
- app_iter_range() (IResponse method), 765
- app_iter_range() (Response method), 814
- application configuration, 324
- application registry, 668, **1158**
- application_url (Request attribute), 805
- ApplicationCreated (class in pyramid.events), 733
- apply_request_extensions() (in module pyramid.request), 813
- as_bytes() (Request method), 805
- asbool() (in module pyramid.settings), 831
- ascii_native_() (in module pyramid.compat), 684
- aslist() (in module pyramid.settings), 831
- assert_() (DummyTemplateRenderer method), 837
- asset, **1158**
- asset descriptor, **1158**
- asset specification, **1158**
- asset specifications, 432
- AssetResolver (class in pyramid.path), 778
- assets, 431
 - generating urls, 434
 - overriding, 443, 642
 - serving, 432
- Authenticated (in module pyramid.security), 824
- authenticated_userid (Request attribute), 789
- authenticated_userid() (AuthTktAuthenticationPolicy method), 675
- authenticated_userid() (BasicAuthAuthenticationPolicy method), 680
- authenticated_userid() (IAuthenticationPolicy method), 757
- authenticated_userid() (in module pyramid.security), 822

authenticated_userid() (RemoteUserAuthenticationPolicy method), 677
 authenticated_userid() (RepozeWho1AuthenticationPolicy method), 681
 authenticated_userid() (SessionAuthenticationPolicy method), 678
 authentication, **1158**
 authentication policy, **1158**
 authentication policy (creating), 574
 authentication policy (extending), 573
 authorization, **1158**
 authorization (Request attribute), 805
 authorization policy, 564, **1158**
 authorization policy (creating), 575
 AuthTktAuthenticationPolicy (class in pyramid.authentication), 673
 AuthTktCookieHelper (class in pyramid.authentication), 682
 AuthTktCookieHelper.AuthTicket (class in pyramid.authentication), 682
 AuthTktCookieHelper.BadTicket, 683
 automatic reloading of templates, 412

B

Babel, 518, **1158**
 BadCSRFOrgin, 737
 BadCSRFToken, 737
 Bangert, Ben, 7
 BaseCookieSessionFactory() (in module pyramid.session), 830
 BasicAuthAuthenticationPolicy (class in pyramid.authentication), 679
 Beelby, Chris, 7
 before render event, 599
 BeforeRender (class in pyramid.events), 735
 BeforeTraversal (class in pyramid.events), 734
 begin() (Configurator method), 690
 Bicking, Ian, 7, 446
 binary_type (in module pyramid.compat), 685
 blank() (IRequestFactory method), 761
 blank() (Request method), 805
 body (IResponse attribute), 767
 body (Request attribute), 805
 body (Response attribute), 814
 body_file (IResponse attribute), 764
 body_file (Request attribute), 805
 body_file (Response attribute), 814
 body_file_raw (Request attribute), 805
 body_file_seekable (Request attribute), 805
 book audience, 5
 book content overview, 5
 bootstrap() (in module pyramid.paster), 775
 Borch, Malthe, 7
 Brandl, Georg, 7
 built-in renderers, 395
 bytes_() (in module pyramid.compat), 685

C

Cache Busting, 436
 cache busting, **1158**
 cache_control (IResponse attribute), 767
 cache_control (Request attribute), 806
 cache_control (Response attribute), 814
 cache_expires (IResponse attribute), 765
 call_application() (Request method), 806
 CALLER_PACKAGE (in module pyramid.path), 777
 categories() (IIntrospector method), 770
 categorized() (IIntrospector method), 771
 category_name (IIntrospectable attribute), 769
 Chameleon, 412, **1158**
 translation strings, 518
 changed() (ISession method), 759
 changing
 renderer, 404
 charset (IResponse attribute), 765
 charset (Response attribute), 814
 check_csrf_origin() (in module pyramid.session), 826
 check_csrf_token() (in module pyramid.session), 827
 class_types (in module pyramid.compat), 685
 cleaning up after request, 451
 cleanUp() (in module pyramid.testing), 836
 clear() (BeforeRender method), 736

- clear() (IDict method), 762
- client_addr (Request attribute), 806
- clone() (DummyResource method), 836
- clone() (IRendererInfo method), 760
- code scanning, 325
- commit() (Configurator method), 690
- compiling
 - message catalog, 514
- conditional_response_app() (IResponse method), 765
- conditional_response_app() (Response method), 814
- configparser (in module pyramid.compat), 685
- configuration
 - advanced, 642
 - conflict detection, 642
 - including from external sources, 648
 - logging, 477
 - middleware, 482
- configuration declaration, **1159**
- configuration decoration, 325, **1159**
- configuration decorator, 608
- configuration directive, **1159**
- ConfigurationError, 737
- Configurator, 321
- configurator, **1159**
 - adding directives, 650
- Configurator (class in pyramid.config), 687
- Configurator testing API, 529
- conflict detection
 - configuration, 642
- conflict resolution, **1159**
- console script, 504, **1159**
- container resources, 535
- content_disposition (IResponse attribute), 766
- content_disposition (Response attribute), 814
- content_encoding (IResponse attribute), 766
- content_encoding (Response attribute), 814
- content_language (IResponse attribute), 767
- content_language (Response attribute), 814
- content_length (IResponse attribute), 764
- content_length (Request attribute), 806
- content_length (Response attribute), 814
- content_location (IResponse attribute), 765
- content_location (Response attribute), 814
- content_md5 (IResponse attribute), 766
- content_md5 (Response attribute), 815
- content_range (IResponse attribute), 765
- content_range (Response attribute), 815
- content_type (IResponse attribute), 767
- content_type (Request attribute), 806
- content_type (Response attribute), 815
- content_type_params (IResponse attribute), 768
- content_type_params (Response attribute), 815
- context, 556, **1159**
- context (Request attribute), 786
- ContextFound (class in pyramid.events), 734
- cookies (Request attribute), 807
- copy() (BeforeRender method), 736
- copy() (IResponse method), 764
- copy() (Request method), 807
- copy() (Response method), 815
- copy_body() (Request method), 807
- copy_get() (Request method), 807
- CPython, **1159**
- created (ISession attribute), 759
- creating a project, 327
- cross-site request forgery attacks, prevention, 459
- current_route_path() (in module pyramid.url), 844
- current_route_path() (Request method), 796
- current_route_url() (in module pyramid.url), 844
- current_route_url() (Request method), 796
- custom settings, 353

D

- date (IResponse attribute), 768
- date (Request attribute), 807
- date (Response attribute), 815
- date and currency formatting (i18n), 518
- de la Guardia, Carlos, 7
- debug settings, 467
- debug toolbar, 334
- debug_all, 467
- debug_authorization, 467
- debug_notfound, 467
- debug_routematch, 467

- debugging
 - route matching, 376
 - templates, 411
 - view configuration, 431
 - debugging authorization failures, 572
 - debugging not found errors, 430
 - declarative configuration, **1159**
 - decorator, **1159**
 - default
 - permission, 566
 - Default Locale Name, **1160**
 - default permission, **1160**
 - default root factory, **1160**
 - Default view, **1160**
 - default view, 556
 - default_locale_name, 467, 519
 - default_locale_negotiator() (in module `pyramid.i18n`), 753
 - Deferred (class in `pyramid.registry`), 781
 - delete_cookie() (IResponse method), 765
 - delete_cookie() (Response method), 815
 - Denied (class in `pyramid.security`), 825
 - Deny (in module `pyramid.security`), 825
 - DENY_ALL (in module `pyramid.security`), 825
 - deployment
 - settings, 353
 - Deployment settings, **1160**
 - derive_view() (Configurator method), 725
 - detecting languages, 520
 - development install, 328
 - dict_of_lists() (IMultiDict method), 763
 - discriminator, **1160**
 - discriminator (IIntrospectable attribute), 768
 - discriminator_hash (IIntrospectable attribute), 769
 - distribute, **1160**
 - distribution, **1160**
 - distributions, showing installed, 500
 - distutils, **1160**
 - Django, 295, 313, **1160**
 - domain
 - translation, 509
 - domain (Request attribute), 807
 - domain model, **1160**
 - dotted Python name, **1161**
 - DottedNameResolver (class in `pyramid.path`), 777
 - DummyRequest (class in `pyramid.testing`), 836
 - DummyResource (class in `pyramid.testing`), 836
 - DummyTemplateRenderer (class in `pyramid.testing`), 837
 - Duncan, Casey, 7
- ## E
- effective_principals (Request attribute), 789
 - effective_principals() (AuthTktAuthenticationPolicy method), 676
 - effective_principals() (BasicAuthAuthenticationPolicy method), 680
 - effective_principals() (IAuthenticationPolicy method), 756
 - effective_principals() (in module `pyramid.security`), 823
 - effective_principals() (RemoteUserAuthenticationPolicy method), 677
 - effective_principals() (Repoze-Who1AuthenticationPolicy method), 681
 - effective_principals() (SessionAuthenticationPolicy method), 678
 - encode_content() (IResponse method), 766
 - encode_content() (Response method), 815
 - end() (Configurator method), 690
 - entry point, **1161**
 - environ (IResponse attribute), 764
 - environment variables, 467
 - escape() (in module `pyramid.compat`), 685
 - etag (IResponse attribute), 767
 - etag (Response attribute), 815
 - event, 463, **1161**
 - Everitt, Paul, 7
 - Everyone (in module `pyramid.security`), 824
 - exc_info (Request attribute), 787
 - exception (Request attribute), 787
 - exception response, **1161**
 - exception responses, 454
 - Exception view, **1161**
 - exception view

- subrequest, 591
- exception views, 387
- exception_response() (in module pyramid.httpexceptions), 741
- EXCVIEW (in module pyramid tweens), 843
- excview_tween_factory() (in module pyramid.tweens), 843
- exec_() (in module pyramid.compat), 685
- exists() (IAAssetDescriptor method), 772
- exists() (ManifestCacheBuster static method), 833
- expires (IResponse attribute), 765
- expires (Response attribute), 816
- explicitly calling
 - renderer, 301
- explicitly calling
 - view renderer, 301
- extend() (IMultiDict method), 764
- extending
 - pshell, 492
- extending an existing application, 640
- extending configuration, 649
- extensible application, 638
- extracting
 - messages, 513

F

- factory (IRoute attribute), 758
- falsey string, **1161**
- file (IActionInfo attribute), 771
- FileIter (class in pyramid.response), 820
- FileResponse (class in pyramid.response), 819
- find_interface() (in module pyramid.traversal), 838
- find_resource() (in module pyramid.traversal), 838
- find_root() (in module pyramid.traversal), 839
- finding by interface or class
 - resource, 544
- finding by path
 - resource, 540
- finding root
 - resource, 542
- finished callback, 601, **1161**
- flash messages, 457
- flash(), 458

- flash() (ISession method), 759
- Forbidden (in module pyramid.exceptions), 737
- Forbidden view, **1161**
- forbidden view, 572, 595
- forbidden_view_config (class in pyramid.view), 849
- forget() (AuthTktAuthenticationPolicy method), 676
- forget() (AuthTktCookieHelper method), 683
- forget() (BasicAuthAuthenticationPolicy method), 680
- forget() (IAAuthenticationPolicy method), 756
- forget() (in module pyramid.security), 823
- forget() (RemoteUserAuthenticationPolicy method), 677
- forget() (RepozeWho1AuthenticationPolicy method), 681
- forget() (SessionAuthenticationPolicy method), 679
- forms, views, and unicode, 390
- framework, 295
- frameworks vs. libraries, 295
- from_bytes() (Request method), 807
- from_file() (Request method), 807
- from_file() (Response method), 816
- fromkeys() (BeforeRender method), 736
- Fulton, Jim, 7
- functional testing, 526
- functional tests, 532

G

- generate() (IRoute method), 758
- generating
 - hybrid URLs, 585
 - resource url, 537
- generating route URLs, 371
- generating static asset urls, 434
- generating urls
 - assets, 434
- Genshi, **1161**
- GET (Request attribute), 804
- get() (BeforeRender method), 736
- get() (IDict method), 762

- get() (IIntrospector method), 771
- get_app() (in module pyramid.paster), 776
- get_appsettings() (in module pyramid.paster), 776
- get_category() (IIntrospector method), 770
- get_csrf_token() (ISession method), 760
- get_current_registry, 666, 669, 671
- get_current_registry() (in module pyramid.threadlocal), 837
- get_current_request, 666
- get_current_request() (in module pyramid.threadlocal), 837
- get_locale_name() (in module pyramid.i18n), 753
- get_localizer() (in module pyramid.i18n), 753
- get_renderer() (in module pyramid.renderers), 781
- get_response() (Request method), 808
- get_root() (in module pyramid.scripting), 822
- get_settings() (Configurator method), 716
- getall() (IMultiDict method), 764
- getGlobalSiteManager, 671
- getmtime() (ManifestCacheBuster static method), 833
- getone() (IMultiDict method), 763
- getSiteManager, 668, 669
- Gettext, 512, **1161**
- gettext, 511
- getUtility, 668, 669
- global views
 - hybrid applications, 584
- global_registries (in module pyramid.config), 730
- Google App Engine, **1161**
- Green Unicorn, **1161**
- Grok, **1161**
- H**
- Hardwick, Nat, 7
- has_body (Response attribute), 816
- has_permission() (in module pyramid.security), 824
- has_permission() (Request method), 791
- Hathaway, Shane, 7
- headerlist (IResponse attribute), 768
- headerlist (Response attribute), 816
- headers (IResponse attribute), 766
- headers (Request attribute), 808
- headers (Response attribute), 816
- hello world program, 319
- helloworld (imperative), 321
- Holth, Daniel, 7
- hook_zca (configurator method), 670
- hook_zca() (Configurator method), 727
- host (Request attribute), 808
- host_port (Request attribute), 808
- host_url (Request attribute), 808
- hosting an app under a prefix, 524
- HTTP caching, 430
- HTTP Exception, **1162**
- HTTP exceptions, 386
- http redirect (from a view), 389
- http_version (Request attribute), 808
- HTTPAccepted, 742
- HTTPBadGateway, 750
- HTTPBadRequest, 745
- HTTPClientError, 742
- HTTPConflict, 747
- HTTPCreated, 742
- HTTPError, 742
- HTTPException, 741
- HTTPExpectationFailed, 749
- HTTPFailedDependency, 749
- HTTPForbidden, 745
- HTTPFound, 744
- HTTPGatewayTimeout, 750
- HTTPGone, 747
- HTTPInsufficientStorage, 750
- HTTPInternalServerError, 749
- HTTPLengthRequired, 747
- HTTPLocked, 749
- HTTPMethodNotAllowed, 746
- HTTPMovedPermanently, 743
- HTTPMultipleChoices, 743
- HTTPNoContent, 743
- HTTPNonAuthoritativeInformation, 742
- HTTPNotAcceptable, 746
- HTTPNotFound, 746
- HTTPNotImplemented, 749
- HTTPNotModified, 744

- HTTPOk, 741
- HTTPPartialContent, 743
- HTTPPaymentRequired, 745
- HTTPPreconditionFailed, 747
- HTTPProxyAuthenticationRequired, 746
- HTTPRedirection, 741
- HTTPRequestEntityTooLarge, 748
- HTTPRequestRangeNotSatisfiable, 748
- HTTPRequestTimeout, 747
- HTTPRequestURITooLong, 748
- HTTPResetContent, 743
- HTTPSeeOther, 744
- HTTPServerError, 742
- HTTPServiceUnavailable, 750
- HTTPTemporaryRedirect, 744
- HTTPUnauthorized, 745
- HTTPUnprocessableEntity, 749
- HTTPUnsupportedMediaType, 748
- HTTPUseProxy, 744
- HTTPVersionNotSupported, 750
- hybrid applications, 578
 - *subpath, 584
 - *traverse route pattern, 580
 - global views, 584
- hybrid URLs
 - generating, 585
- I
- i18n, 508
- IActionInfo (interface in pyramid.interfaces), 771
- IApplicationCreated (interface in pyramid.interfaces), 754
- IAssetDescriptor (interface in pyramid.interfaces), 772
- IAuthenticationPolicy (interface in pyramid.interfaces), 756
- IAuthorizationPolicy (interface in pyramid.interfaces), 757
- IBeforeRender (interface in pyramid.interfaces), 755
- IBeforeTraversal (interface in pyramid.interfaces), 755
- ICacheBuster (interface in pyramid.interfaces), 772
- IContextFound (interface in pyramid.interfaces), 754
- identify() (AuthTktCookieHelper method), 683
- IDict (interface in pyramid.interfaces), 762
- IExceptionResponse (interface in pyramid.interfaces), 757
- if_match (Request attribute), 808
- if_modified_since (Request attribute), 808
- if_none_match (Request attribute), 809
- if_range (Request attribute), 809
- if_unmodified_since (Request attribute), 809
- IIntrospectable (interface in pyramid.interfaces), 768
- IIntrospector (interface in pyramid.interfaces), 770
- im_func (in module pyramid.compat), 685
- imperative configuration, 321, 324, **1162**
- IMultiDict (interface in pyramid.interfaces), 763
- include() (Configurator method), 691
- including from external sources
 - configuration, 648
- INewRequest, 463
- INewRequest (interface in pyramid.interfaces), 754
- INewResponse, 463
- INewResponse (interface in pyramid.interfaces), 755
- INGRESS (in module pyramid.tweens), 843
- INGRESS (in module pyramid.viewderivers), 849
- ini file, 337
- ini file settings, 467
- initializing
 - message catalog, 513
- input_() (in module pyramid.compat), 685
- inside() (in module pyramid.location), 775
- install
 - Python (from package, Windows), 315
- install preparation, 314
- installing on Mac OS X, 317
- installing on UNIX, 317
- installing on Windows, 318
- integer_types (in module pyramid.compat), 685
- integration testing, 526
- integration tests, 532
- interactive shell, 491

- interface, **1162**
 - Internationalization, **1162**
 - internationalization, 508
 - introspectable, **1162**
 - Introspectable (class in pyramid.registry), 781
 - introspectable (Configurator attribute), 730
 - introspection, 623
 - introspector, 623, **1162**
 - introspector (Configurator attribute), 730
 - introspector (Registry attribute), 780
 - invalidate() (ISession method), 759
 - invoke_exception_view() (Request method), 790
 - invoke_subrequest() (Request method), 789
 - invoking a request, 498
 - IRenderer (interface in pyramid.interfaces), 761
 - IRendererFactory (interface in pyramid.interfaces), 761
 - IRendererInfo (interface in pyramid.interfaces), 760
 - IRequestFactory (interface in pyramid.interfaces), 761
 - IResourceURL (interface in pyramid.interfaces), 772
 - IResponse, 604
 - IResponse (interface in pyramid.interfaces), 764
 - IResponseFactory (interface in pyramid.interfaces), 761
 - IRoute (interface in pyramid.interfaces), 757
 - IRoutePregenerator (interface in pyramid.interfaces), 758
 - is_body_readable (Request attribute), 809
 - is_body_seekable (Request attribute), 809
 - is_nonstr_iter() (in module pyramid.compat), 685
 - is_response() (Request method), 809
 - is_xhr (Request attribute), 809
 - isdir() (IAssetDescriptor method), 772
 - ISession (interface in pyramid.interfaces), 759
 - ISessionFactory (interface in pyramid.interfaces), 760
 - items() (BeforeRender method), 736
 - items() (DummyResource method), 836
 - items() (IDict method), 763
 - iteritems_() (in module pyramid.compat), 685
 - iterkeys_() (in module pyramid.compat), 686
 - intervalues_() (in module pyramid.compat), 685
 - IViewDeriver (interface in pyramid.interfaces), 773
 - IViewDeriverInfo (interface in pyramid.interfaces), 774
 - IViewMapper (interface in pyramid.interfaces), 762
 - IViewMapperFactory (interface in pyramid.interfaces), 761
- ## J
- Jinja2, 412, **1162**
 - Jinja2 i18n, 519
 - jQuery, **1162**
 - JSON, **1162**
 - renderer, 396
 - JSON (class in pyramid.renderers), 783
 - json (Request attribute), 809
 - json (Response attribute), 816
 - json_body
 - request, 449
 - json_body (Request attribute), 803
 - json_body (Response attribute), 816
 - JSONP
 - renderer, 399
 - JSONP (class in pyramid.renderers), 784
 - Jython, **1163**
- ## K
- keys() (BeforeRender method), 736
 - keys() (DummyResource method), 836
 - keys() (IDict method), 762
 - Koym, Todd, 7
- ## L
- l10n, 508
 - Laflamme, Blaise, 7
 - Laflamme, Hugues, 7
 - last_modified (IResponse attribute), 766
 - last_modified (Response attribute), 816
 - leaf resources, 535
 - line (IActionInfo attribute), 771
 - lineage, **1163**
 - resource, 540

- lineage() (in module pyramid.location), 774
 - Lingua, 512, **1163**
 - listdir() (IAssetDescriptor method), 772
 - locale
 - negotiator, 521
 - setting, 522
 - Locale Name, **1163**
 - locale name, 517
 - Locale Negotiator, **1163**
 - locale negotiator, 522
 - locale_name (Localizer attribute), 752
 - locale_name (Request attribute), 804
 - Localization, **1163**
 - localization, 508
 - localization deployment settings, 519
 - Localizer, **1163**
 - localizer, 515
 - Localizer (class in pyramid.i18n), 752
 - localizer (Request attribute), 804, 809
 - location, **1163**
 - location (IResponse attribute), 764
 - location (Response attribute), 816
 - location-aware
 - resource, 536
 - security, 571
 - logging
 - .ini, 477
 - configuration, 477
 - settings, 477
 - long (in module pyramid.compat), 686
- M**
- MAIN (in module pyramid tweens), 843
 - make_body_seekable() (Request method), 809
 - make_localizer() (in module pyramid.i18n), 754
 - make_tempfile() (Request method), 809
 - make_wsgi_app, 322
 - make_wsgi_app() (Configurator method), 692
 - Mako, 412, **1163**
 - Mako i18n, 519
 - manifest (ManifestCacheBuster attribute), 833
 - MANIFEST.in, 340
 - ManifestCacheBuster (class in pyramid.static), 832
 - map_() (in module pyramid.compat), 686
 - mapping to view callable
 - resource, 413
 - URL pattern, 413
 - match() (IRoute method), 758
 - matchdict, 367, **1163**
 - matchdict (Request attribute), 788
 - matched_route, 367
 - matched_route (Request attribute), 788
 - matching
 - root URL, 371
 - matching the root URL, 371
 - matching views
 - printing, 489
 - max_forwards (Request attribute), 810
 - maybe_dotted() (Configurator method), 727
 - maybe_resolve() (DottedNameResolver method), 777
 - md5_etag() (IResponse method), 768
 - md5_etag() (Response method), 817
 - merge_cookies() (IResponse method), 764
 - merge_cookies() (Response method), 817
 - Merickel, Michael, 7
 - Message Catalog, **1163**
 - message catalog
 - compiling, 514
 - initializing, 513
 - updating, 514
 - Message Identifier, **1163**
 - message identifier, 509
 - messages
 - extracting, 513
 - METAL, **1164**
 - method (Request attribute), 810
 - middleware, **1164**
 - .ini, 482
 - configuration, 482
 - TransLogger, 482
 - mixed() (IMultiDict method), 763
 - mod_wsgi, **1164**
 - modifying
 - package structure, 347
 - module, **1164**

Moroz, Tom, 7

msgid

translation, 509

multidict, **1164**

multidict (WebOb), 449

MVC, 313

N

name (IRendererInfo attribute), 760

name (IRoute attribute), 758

native_() (in module pyramid.compat), 686

negotiate_locale_name, 517

negotiate_locale_name() (in module pyramid.i18n), 753

negotiator

locale, 521

new (ISession attribute), 759

new_csrf_token() (ISession method), 759

NewRequest, 463

NewRequest (class in pyramid.events), 733

NewResponse, 463

NewResponse (class in pyramid.events), 734

NO_PERMISSION_REQUIRED (in module pyramid.security), 825

not found error (debugging), 430

Not Found View, **1164**

not found view, 592

not_ (class in pyramid.config), 730

NotFound (in module pyramid.exceptions), 737

notfound_view_config (class in pyramid.view), 847

notify() (Registry method), 781

null_renderer (in module pyramid.renderers), 786

O

object tree, 535, 554

options (IViewDeriver attribute), 773

options (IViewDeriverInfo attribute), 774

Oram, Simon, 7

order (IIntrospectable attribute), 769

original_view (IViewDeriverInfo attribute), 774

Orr, Mike, 7

override_asset, 443

override_asset() (Configurator method), 715

overriding

assets, 443, 642

resource URL generation, 538

routes, 642

views, 641

overriding at runtime

renderer, 404

P

package, 343, **1164**

package (IRendererInfo attribute), 761

package (IViewDeriverInfo attribute), 774

package structure

modifying, 347

package_name (Registry attribute), 780

Paez, Patricio, 7

par: settings

adding custom, 475

params (Request attribute), 810

parse_manifest() (ManifestCacheBuster method), 833

parse_ticket() (AuthTktCookieHelper static method), 683

Passing in configuration variables, 393

PasteDeploy, 337, **1164**

PasteDeploy settings, 467

path (Request attribute), 810

path_info (Request attribute), 810

path_info_peek() (Request method), 810

path_info_pop() (Request method), 810

path_qs (Request attribute), 810

path_url (Request attribute), 810

pattern (IRoute attribute), 757

pcreate, 327

–help, 851

pdistreport, 500

–help, 852

peek_flash(), 459

peek_flash() (ISession method), 760

permission, **1164**

default, 566

permission names, 570

permissions, 565

- permits() (IAuthorizationPolicy method), 757
- Peters, Tim, 7
- PHASE0_CONFIG (in module pyramid.config), 731
- PHASE1_CONFIG (in module pyramid.config), 731
- PHASE2_CONFIG (in module pyramid.config), 731
- PHASE3_CONFIG (in module pyramid.config), 731
- physical path, **1164**
- physical root, **1164**
- physical_path (IResourceURL attribute), 772
- physical_path_tuple (IResourceURL attribute), 772
- pickle (in module pyramid.compat), 686
- PickleSerializer (class in pyramid.session), 831
- pip, **1165**
- pipeline, **1165**
- pkg_resources, **1165**
- pluralization, 515
- pluralize() (Localizer method), 752
- pluralizing (i18n), 516
- pop() (BeforeRender method), 736
- pop() (IDict method), 763
- pop_flash(), 459
- pop_flash() (ISession method), 760
- popitem() (BeforeRender method), 736
- popitem() (IDict method), 763
- POST (Request attribute), 804
- post() (PyramidTemplate method), 821
- post() (Template method), 821
- pragma (IResponse attribute), 767
- pragma (Request attribute), 810
- pragma (Response attribute), 817
- pre() (PyramidTemplate method), 821
- pre() (Template method), 821
- predicate, **1165**
- predicate factory, **1165**
- PredicateMismatch, 737
- predicates (IRoute attribute), 758
- predicates (IViewDeriverInfo attribute), 774
- predvalseq (class in pyramid.registry), 781
- pregenerator, **1165**
- pregenerator (IRoute attribute), 758
- prepare() (IExceptionResponse method), 757
- prepare() (in module pyramid.scripting), 822
- prequest, 498
 - help, 852
- prevent_http_cache, 467
- preventing cross-site request forgery attacks, 459
- principal, 570, **1165**
- principal names, 570
- principals_allowed_by_permission() (IAuthorizationPolicy method), 757
- principals_allowed_by_permission() (in module pyramid.security), 824
- printing
 - matching views, 489
 - routes, 495
 - tweens, 497
- production.ini, 340
- project, 327, **1165**
- project structure, 336
- protecting views, 565
- proutes, 495
 - help, 853
- pserve, 331
 - help, 854
- pshell, 491
 - help, 855
 - extending, 492
- ptweens, 497
 - help, 856
- pviews, 489
 - help, 856
- PY3 (in module pyramid.compat), 686
- Pylons, 295, 313, **1165**
- Pylons Project, 313
- Pylons-style controller dispatch, 393
- PyPI, **1165**
- PyPy, **1165**
- PYPY (in module pyramid.compat), 686
- pyramid and other frameworks, 313
- Pyramid Community Cookbook, **1165**
- pyramid genesis, 6
- pyramid.authentication (module), 673

pyramid.authorization (module), 684
 pyramid.compat (module), 684
 pyramid.config (module), 687
 pyramid.decorator (module), 731
 pyramid.events (module), 732
 pyramid.exceptions (module), 737
 pyramid.httpexceptions (module), 738
 pyramid.i18n (module), 751
 pyramid.interfaces (module), 754
 pyramid.location (module), 774
 pyramid.paster (module), 775
 pyramid.path (module), 777
 pyramid.registry (module), 780
 pyramid.renderers (module), 781
 pyramid.request (module), 786
 pyramid.response (module), 813
 pyramid.scaffolds (module), 821
 pyramid.scripting (module), 822
 pyramid.security (module), 822
 pyramid.session (module), 826
 pyramid.settings (module), 831
 pyramid.static (module), 832
 pyramid.testing, 529
 pyramid.testing (module), 834
 pyramid.threadlocal (module), 837
 pyramid.traversal (module), 838
 pyramid.tweens (module), 843
 pyramid.url (module), 844
 pyramid.view (module), 845
 pyramid.viewderivers (module), 849
 pyramid.wsgi (module), 850
 pyramid_debugtoolbar, **1166**
 pyramid_exclog, **1166**
 pyramid_handlers, **1166**
 pyramid_jinja2, **1166**
 pyramid_redis_sessions, 457, **1166**
 pyramid_zcml, **1166**
 PyramidTemplate (class in pyramid.scaffolds), 821
 Python, **1166**
 Python (from package, Windows)
 install, 315
 Python Packaging Authority, **1166**
 pyvenv, **1166**

Q

query_string (Request attribute), 810
 QueryStringCacheBuster (class in pyramid.static),
 833
 QueryStringConstantCacheBuster (class in pyra-
 mid.static), 834
 quote_path_segment() (in module pyra-
 mid.traversal), 840

R

range (Request attribute), 811
 redirecting to slash-appended routes, 374
 referer (Request attribute), 811
 referrer (Request attribute), 811
 register() (IIntrospectable method), 769
 Registry (class in pyramid.registry), 780
 registry (Configurator attribute), 730
 registry (IRendererInfo attribute), 761
 registry (IViewDeriverInfo attribute), 774
 registry (Request attribute), 786
 reify() (in module pyramid.decorator), 731
 relate() (IIntrospectable method), 769
 relate() (IIntrospector method), 770
 related() (IIntrospector method), 770
 relative_url() (Request method), 811
 reload, 331, 467
 reload settings, 467
 reload_all, 467
 reload_assets, 467, 475
 reload_templates, 475
 remember() (AuthTktAuthenticationPolicy
 method), 676
 remember() (AuthTktCookieHelper method), 683
 remember() (BasicAuthAuthenticationPolicy
 method), 680
 remember() (IAAuthenticationPolicy method), 756
 remember() (in module pyramid.security), 823
 remember() (RemoteUserAuthenticationPolicy
 method), 678
 remember() (RepozeWho1AuthenticationPolicy
 method), 682
 remember() (SessionAuthenticationPolicy
 method), 679

- remote_addr (Request attribute), 811
- remote_user (Request attribute), 811
- RemoteUserAuthenticationPolicy (class in pyramid.authentication), 676
- remove() (Introspector method), 770
- remove_conditional_headers() (Request method), 811
- render() (in module pyramid.renderers), 781
- render_template() (Template method), 821
- render_to_response() (in module pyramid.renderers), 782
- render_view() (in module pyramid.view), 846
- render_view_to_iterable() (in module pyramid.view), 846
- render_view_to_response() (in module pyramid.view), 845
- renderer, 394, **1166**
 - adding, 401
 - changing, 404
 - explicitly calling, 301
 - JSON, 396
 - JSONP, 399
 - overriding at runtime, 404
 - string, 396
 - system values, 408
 - templates, 409
- renderer (template), 408
- renderer factory, **1166**
- renderer globals, **1167**
- renderer response headers, 400
- renderers (built-in), 395
- rendering_val (IBeforeRender attribute), 756
- Repoze, **1167**
- repoze.bfg genesis, 6
- repoze.catalog, **1167**
- repoze.lemonade, **1167**
- repoze.who, **1167**
- repoze.workflow, **1167**
- repoze.zope2, 6
- RepozeWho1AuthenticationPolicy (class in pyramid.authentication), 681
- request, 354, **1167**
 - json_body, 449
 - request (and text/unicode), 449
- Request (class in pyramid.request), 786
- request (IBeforeTraversal attribute), 755
- request (IContextFound attribute), 755
- request (INewRequest attribute), 754
- request (INewResponse attribute), 755
- request (IResponse attribute), 767
- request attributes, 446
- request attributes (special), 447
- request factory, 596, **1167**
- request lifecycle, 354
- request method, 597
- request methods, 448
- request object, 446
- request processing, 354
- request type, **1167**
- request URLs, 448
- request.registry, 669
- request_iface (DummyRequest attribute), 837
- request_iface (Request attribute), 811
- RequestClass (IResponse attribute), 768
- requirements for installing packages, 317
- reraise() (in module pyramid.compat), 686
- resolve() (AssetResolver method), 779
- resolve() (DottedNameResolver method), 778
- resource, 550, **1167**
 - ACL, 567
 - finding by interface or class, 544
 - finding by path, 540
 - finding root, 542
 - lineage, 540
 - location-aware, 536
 - mapping to view callable, 413
- resource API functions, 545
- resource interfaces, 542, 562
- Resource Location, **1167**
- resource path generation, 539
- resource tree, 535, 554, **1167**
- resource url
 - generating, 537
- resource URL generation
 - overriding, 538
- resource_path() (in module pyramid.traversal), 839

- resource_path() (Request method), 802
 - resource_path_tuple() (in module `pyramid.traversal`), 840
 - resource_url, 537
 - resource_url() (in module `pyramid.url`), 844
 - resource_url() (Request method), 798
 - response, 385, **1168**
 - Response (class in `pyramid.response`), 813
 - response (creating), 453
 - response (INewResponse attribute), 755
 - response (Request attribute), 787, 811
 - response adapter, **1168**
 - response callback, 600, **1168**
 - response factory, 599, **1168**
 - response headers, 453
 - response headers (from a renderer), 400
 - response object, 452
 - response_adapter() (in module `pyramid.response`), 820
 - reStructuredText, **1168**
 - retry_after (IResponse attribute), 766
 - retry_after (Response attribute), 817
 - RFC
 - RFC 2068, 738
 - RFC 2616, 764
 - RFC 3986#section-3.5, 794
 - root, **1168**
 - root (Request attribute), 786
 - root factory, 556, **1168**
 - root URL
 - matching, 371
 - root url (matching), 371
 - Rossi, Chris, 7
 - route, **1168**
 - view callable lookup details, 382
 - route configuration, 360, **1168**
 - route configuration arguments, 366
 - route factory, 381
 - route matching, 366
 - debugging, 376
 - route ordering, 365
 - route path pattern syntax, 361
 - route predicate, **1169**
 - route predicates (custom), 378
 - route subpath, 584
 - route URLs, 371
 - route_path() (in module `pyramid.url`), 844
 - route_path() (Request method), 795
 - route_url() (in module `pyramid.url`), 844
 - route_url() (Request method), 793
 - router, 354, **1169**
 - Routes, **1169**
 - routes
 - overriding, 642
 - printing, 495
 - routes mapper, **1169**
 - running an application, 331
 - running tests, 329
- ## S
- Sawyers, Andrew, 7
 - scaffold, **1169**
 - scaffolds, 327
 - scan, **1169**
 - scan() (Configurator method), 693
 - scheme (Request attribute), 811
 - script_name (Request attribute), 812
 - Seaver, Tres, 7
 - security, 564
 - location-aware, 571
 - URL dispatch, 382
 - view, 429
 - send() (Request method), 812
 - server (IResponse attribute), 767
 - server (Response attribute), 817
 - server_name (Request attribute), 812
 - server_port (Request attribute), 812
 - serving
 - assets, 432
 - session, 454, **1169**
 - session (Request attribute), 788, 812
 - session factory, **1169**
 - session factory (alternates), 457
 - session factory (custom), 457
 - session factory (default), 455
 - session object, 456

- session.flash, 458
- session.get_csrf_token, 460
- session.new_csrf_token, 461
- session.peek_flash, 459
- session.pop_flash, 458
- SessionAuthenticationPolicy (class in pyramid.authentication), 678
- set_authentication_policy() (Configurator method), 711
- set_authorization_policy() (Configurator method), 711
- set_cookie() (IResponse method), 767
- set_cookie() (Response method), 817
- set_default_csrf_options() (Configurator method), 711
- set_default_permission() (Configurator method), 712
- set_locale_negotiator() (Configurator method), 715
- set_property() (Request method), 803
- set_request_factory() (Configurator method), 722
- set_request_property() (Configurator method), 714
- set_root_factory() (Configurator method), 722
- set_session_factory() (Configurator method), 723
- set_view_mapper() (Configurator method), 723
- setdefault() (BeforeRender method), 736
- setdefault() (IDict method), 763
- setting
 - locale, 522
- settings, 467
 - .ini, 350
 - deployment, 353
 - logging, 477
 - middleware, 482
- settings (IRendererInfo attribute), 760
- settings (IViewDeriverInfo attribute), 774
- settings (Registry attribute), 780
- setUp() (in module pyramid.testing), 834
- setup.py, 341
- setup.py develop, 328
- setup_logging() (in module pyramid.paster), 776
- setup_registry() (Configurator method), 727
- setuptools, **1169**
- Shipman, John, 7
- showing installed distributions, 500
- signed_deserialize() (in module pyramid.session), 826
- signed_serialize() (in module pyramid.session), 826
- SignedCookieSessionFactory() (in module pyramid.session), 827
- SimpleCookie (in module pyramid.compat), 686
- special ACE, 571
- special permission names, 570
- special view responses, 604
- SQLAlchemy, **1169**
- starter scaffold, 327
- startup, 331
- startup process, 350
- static asset urls, 434
- static assets view, 440
- static assets, 431
- static directory, 346
- static routes, 373
- static_path() (in module pyramid.url), 845
- static_path() (Request method), 798
- static_url() (in module pyramid.url), 844
- static_url() (Request method), 797
- static_view (class in pyramid.static), 832
- status (IResponse attribute), 765
- status (Response attribute), 819
- status_code (Response attribute), 819
- status_int (IResponse attribute), 768
- status_int (Response attribute), 819
- status_map (in module pyramid.httpexceptions), 741
- stream() (IAssetDescriptor method), 772
- string
 - renderer, 396
- string_types (in module pyramid.compat), 686
- subpath, 556, **1169**
- subpath (Request attribute), 786
- subpath (route), 584
- subrequest, 587
 - exception view, 591
 - use_tweens, 589
- subscriber, 463, **1169**

subscriber() (in module pyramid.events), 732
 system values
 renderer, 408

T

tearDown() (in module pyramid.testing), 835
 template, **1170**
 Template (class in pyramid.scaffolds), 821
 template automatic reload, 412
 template renderers, 408
 template system bindings, 412
 template_dir() (Template method), 821
 templates
 debugging, 411
 renderer, 409
 templates used as renderers, 408
 templates used directly, 405
 test setup, 527
 test tear down, 527
 testConfig() (in module pyramid.testing), 835
 testing_add_renderer() (Configurator method), 728
 testing_add_subscriber() (Configurator method), 728
 testing_resources() (Configurator method), 728
 testing_securitypolicy() (Configurator method), 729
 tests (running), 329
 tests.py, 346
 text (Request attribute), 812
 text (Response attribute), 819
 text_() (in module pyramid.compat), 686
 text_type (in module pyramid.compat), 686
 thread local, **1170**
 thread locals, 666
 title (IIntrospectable attribute), 768
 translate() (Localizer method), 752
 translating (i18n), 515
 translation, 515
 activating, 521
 domain, 509
 msgid, 509
 Translation Context, **1170**
 translation directories, 511

Translation Directory, **1170**
 translation directory, 521
 adding, 521
 Translation Domain, **1170**
 Translation String, **1170**
 translation string, 509
 translation string factory, 510
 translation strings
 Chameleon, 518
 TranslationString (class in pyramid.i18n), 751
 TranslationStringFactory() (in module pyramid.i18n), 751
 Translator, **1170**
 TransLogger, 482
 traversal, 549, **1170**
 traversal algorithm, 556
 traversal details, 553
 traversal examples, 559
 traversal quick example, 546
 traversal tree, 535, 554
 traversal_path() (in module pyramid.traversal), 843
 traverse() (in module pyramid.traversal), 841
 traversed (Request attribute), 787
 traverser, 602
 truthy string, **1170**
 tween, **1171**
 tweens
 printing, 497
 type (IRendererInfo attribute), 760
 type_name (IIntrospectable attribute), 769

U

ubody (Response attribute), 819
 unauthenticated_userid (Request attribute), 789
 unauthenticated_userid() (AuthTktAuthenticationPolicy method), 676
 unauthenticated_userid() (BasicAuthAuthenticationPolicy method), 680
 unauthenticated_userid() (IAuthenticationPolicy method), 756
 unauthenticated_userid() (in module pyramid.security), 823

- unauthenticated_userid() (RemoteUserAuthenticationPolicy method), 678
 - unauthenticated_userid() (Repoze-Who1AuthenticationPolicy method), 682
 - undefere() (in module pyramid.registry), 781
 - UnencryptedCookieSessionFactoryConfig() (in module pyramid.session), 829
 - unhook_zca() (Configurator method), 727
 - unicode and text (and the request), 449
 - unicode, views, and forms, 390
 - unicode_body (IResponse attribute), 764
 - unicode_body (Response attribute), 819
 - unit testing, 526
 - unittest, 527
 - unrelate() (IIntrospectable method), 768
 - unrelate() (IIntrospector method), 770
 - unset_cookie() (IResponse method), 766
 - unset_cookie() (Response method), 819
 - upath_info (Request attribute), 812
 - update() (BeforeRender method), 736
 - update() (IDict method), 762
 - updating
 - message catalog, 514
 - url (Request attribute), 812
 - URL dispatch, 360, 548, **1171**
 - security, 382
 - url generation (traversal), 545
 - URL generator, 603
 - URL pattern
 - mapping to view callable, 413
 - url_encode (in module pyramid.compat), 687
 - url_encoding (Request attribute), 812
 - url_open (in module pyramid.compat), 687
 - url_quote (in module pyramid.compat), 687
 - url_quote_plus (in module pyramid.compat), 687
 - url_unquote (in module pyramid.compat), 687
 - url_unquote_native() (in module pyramid.compat), 687
 - url_unquote_text() (in module pyramid.compat), 687
 - urlargs (Request attribute), 812
 - URLDecodeError, 737
 - urlencode() (in module pyramid.url), 845
 - urlparse (in module pyramid.compat), 686
 - urlvars (Request attribute), 812
 - uscript_name (Request attribute), 813
 - use_tweens
 - subrequest, 589
 - user_agent (Request attribute), 813
 - userid, **1171**
- ## V
- values() (BeforeRender method), 736
 - values() (DummyResource method), 836
 - values() (IDict method), 763
 - van Rossum, Guido, 7
 - vary (IResponse attribute), 765
 - vary (Response attribute), 819
 - Venusian, **1171**
 - venv, **1171**
 - view, **1171**
 - security, 429
 - VIEW (in module pyramid.viewderivers), 849
 - view callable, **1171**
 - view callable lookup details
 - route, 382
 - view callables, 384
 - view calling convention, 384, 392
 - view class, 384
 - view configuration, **1171**
 - debugging, 431
 - view configuration parameters, 413
 - view driver, **1172**
 - view drivers, 620
 - view exceptions, 386
 - view function, 384
 - View handler, **1172**
 - view http redirect, 389
 - View Lookup, **1172**
 - view lookup, 413, 551, 556
 - view mapper, 607, **1172**
 - view name, 556, **1172**
 - view predicate, **1172**
 - view renderer, 394
 - explicitly calling, 301

- view response, 385
- view security, 429
- view_config, 325
- view_config (class in pyramid.view), 846
- view_config decorator, 422
- view_config placement, 424
- view_defaults (class in pyramid.view), 847
- view_defaults class decorator, 426
- view_execution_permitted() (in module pyramid.security), 824
- view_name (Request attribute), 787
- views
 - overriding, 641
- views, forms, and unicode, 390
- views.py, 344
- virtual environment, **1172**
- virtual hosting, 524
- virtual root, 525, **1172**
- virtual_path (IResourceURL attribute), 772
- virtual_path_tuple (IResourceURL attribute), 772
- virtual_root (Request attribute), 787
- virtual_root() (in module pyramid.traversal), 840
- virtual_root_path (Request attribute), 787
- virtualenv, **1172**

W

- Waitress, **1172**
- WebOb, 446, **1172**
- WebTest, **1172**
- with_package() (Configurator method), 725
- WSGI, 333, **1173**
- WSGI application, 322
- wsgiapp() (in module pyramid.wsgi), 850
- wsgiapp2() (in module pyramid.wsgi), 850
- www_authenticate (IResponse attribute), 766
- www_authenticate (Response attribute), 819

Z

- ZCA, 668
- ZCA global API, 669
- ZCA global registry, 671
- ZCML, **1173**
- ZODB, **1173**

- zodb scaffold, 327
- Zope, 295, 313, **1173**
- Zope 2, 6
- Zope 3, 6
- Zope Component Architecture, 668, **1173**
- zope.component, 668
- ZPT, **1173**