Pylons Reference Documentation

Release 0.9.7

Ben Bangert, Graham Higgins, James Gardner

January 09, 2013

CONTENTS

CHAPTER

ONE

GETTING STARTED

This section is intended to get Pylons up and running as fast as possible and provide a quick overview of the project. Links are provided throughout to encourage exploration of the various aspects of Pylons.

1.1 Requirements

- Python 2.3+ (Python 2.4+ highly recommended)

1.2 Installing

Warning: These instructions require Python 2.4+. For installing with Python 2.3, see *Python 2.3 Installation Instructions*.

To avoid conflicts with system-installed Python libraries, Pylons comes with a boot-strap Python script that sets up a "virtual" Python environment. Pylons will then be installed under the virtual environment.

By the Way

virtualenv is a useful tool to create isolated Python environments. In addition to isolating packages from possible system conflicts, it makes it easy to install Python libraries using *easy_install* without dumping lots of packages into the system-wide Python.

The other great benefit is that no root access is required since all modules are kept under the desired directory. This makes it easy to setup a working Pylons install on shared hosting providers and other systems where system-wide access is unavailable.

---

1. Download the go-pylons.py script.
2. Run the script and specify a directory for the virtual environment to be created under:

```
$ python go-pylons.py mydevenv
```

Tip

The two steps can be combined on unix systems with curl using the following short-cut:

```
$ curl http://pylonshq.com/download/0.9.7/go-pylons.py | python - mydevenv
```

To isolate further from additional system-wide Python libraries, run with the –no-site-packages option:

```
$ python go-pylons.py --no-site-packages mydevenv
```

This will leave a functional virtualenv and Pylons installation. Activate the virtual environment (scripts may also be run by specifying the full path to the mydevenv/bin dir):

```
$ source mydevenv/bin/activate
```

Or on Window to activate:

```
> mydevenv\bin\activate.bat
```

1.2.1 Working Directly From the Source Code

Mercurial must be installed to retrieve the latest development source for Pylons. Mercurial packages are also available for Windows, MacOSX, and other OS's.

Check out the latest code:

```
$ hg clone https://www.knowledgetap.com/hg/pylons-dev Pylons
```

To tell setuptools to use the version in the `Pylons` directory:

```
$ cd Pylons
$ python setup.py develop
```

The active version of Pylons is now the copy in this directory, and changes made there will be reflected for Pylons apps running.

1.3 Creating a Pylons Project

Create a new project named `helloworld` with the following command:

```
$ paster create -t pylons helloworld
```

Note: Windows users must configure their `PATH` as described in *Windows Notes*, otherwise they must specify the full path to the `paster` command (including the virtual environment bin directory).

Running this will prompt for two choices:

1. which templating engine to use
2. whether to include *SQLAlchemy* support

Hit enter at each prompt to accept the defaults (Mako templating, no *SQLAlchemy*).

Here is the created directory structure with links to more information:

- **helloworld**
  - MANIFEST.in
  - README.txt
  - development.ini - *Runtime Configuration*
  - docs

  - ez_setup.py
  - helloworld (See the nested *helloworld directory*)
  - helloworld.egg-info
  - setup.cfg
  - setup.py - *Application Setup*
  - test.ini

The nested `helloworld directory` looks like this:

- **helloworld**
  - __init__.py
  - **config**
    - environment.py - *Environment*
    - middleware.py - *Middleware*
    - routing.py - *URL Configuration*
  - controllers - *Controllers*
  - **lib**
    - app_globals.py - *app_globals*
    - base.py
    - helpers.py - *Helpers*
  - model - *Models*
  - public
  - templates - *Templates*
  - tests - *Unit and functional testing*
  - websetup.py - *Runtime Configuration*

1.4 Running the application

Run the web application:

```
$ cd helloworld
$ paster serve --reload development.ini
```

The command loads the project's server configuration file in `development.ini` and serves the Pylons application.

Note: The `--reload` option ensures that the server is automatically reloaded if changes are made to Python files or the `development.ini` config file. This is very useful during development. To stop the server press **Ctrl+c** or the platform's equivalent.

Visiting http://127.0.0.1:5000/ when the server is running will show the welcome page.

1.5 Hello World

To create the basic hello world application, first create a *controller* in the project to handle requests:

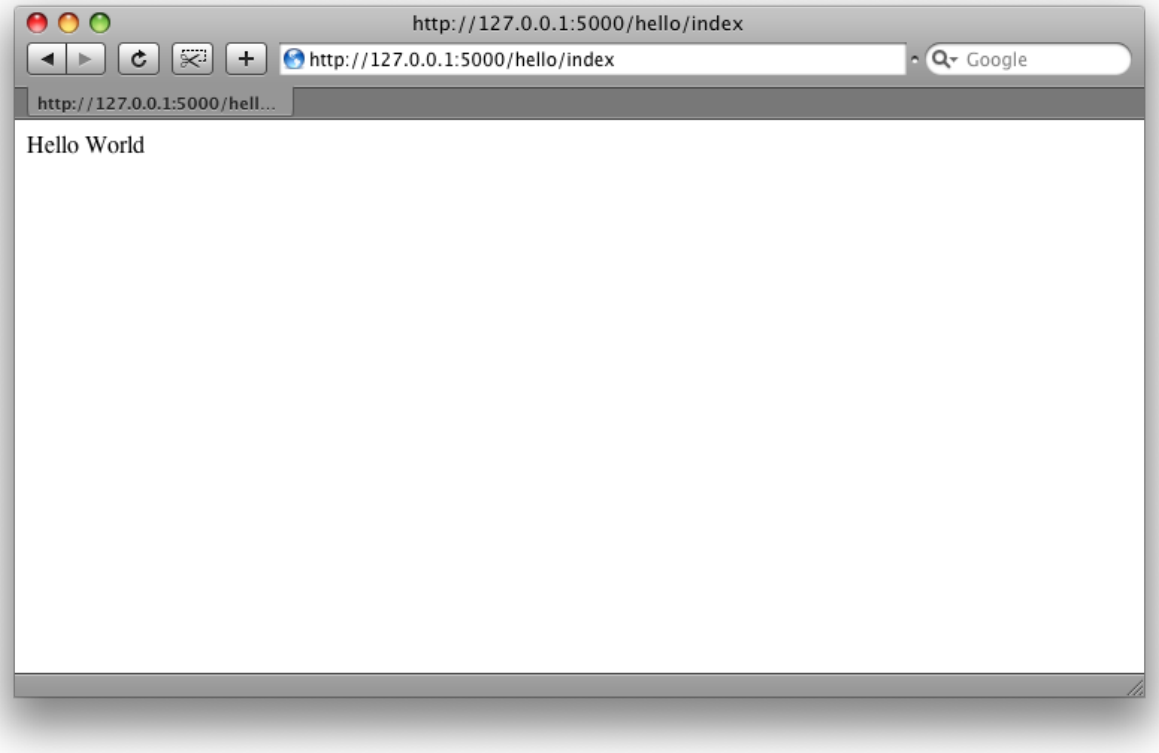
```
$ paster controller hello
```

Open the `helloworld/controllers/hello.py` module that was created. The default controller will return just the string 'Hello World':

```
import logging

from pylons import request, response, session, tmpl_context as c
from pylons.controllers.util import abort, redirect_to

from helloworld.lib.base import BaseController, render

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        # Return a rendered template
        #return render('/hello.mako')
        # or, Return a response
        return 'Hello World'
```

At the top of the module, some commonly used objects are imported automatically.

Navigate to http://127.0.0.1:5000/hello/index where there should be a short text string saying "Hello World" (start up the app if needed):



Tip

URL Configuration explains how URL's get mapped to controllers and their methods.

Add a template to render some of the information that's in the *environ*.

First, create a `hello.mako` file in the `templates` directory with the following contents:

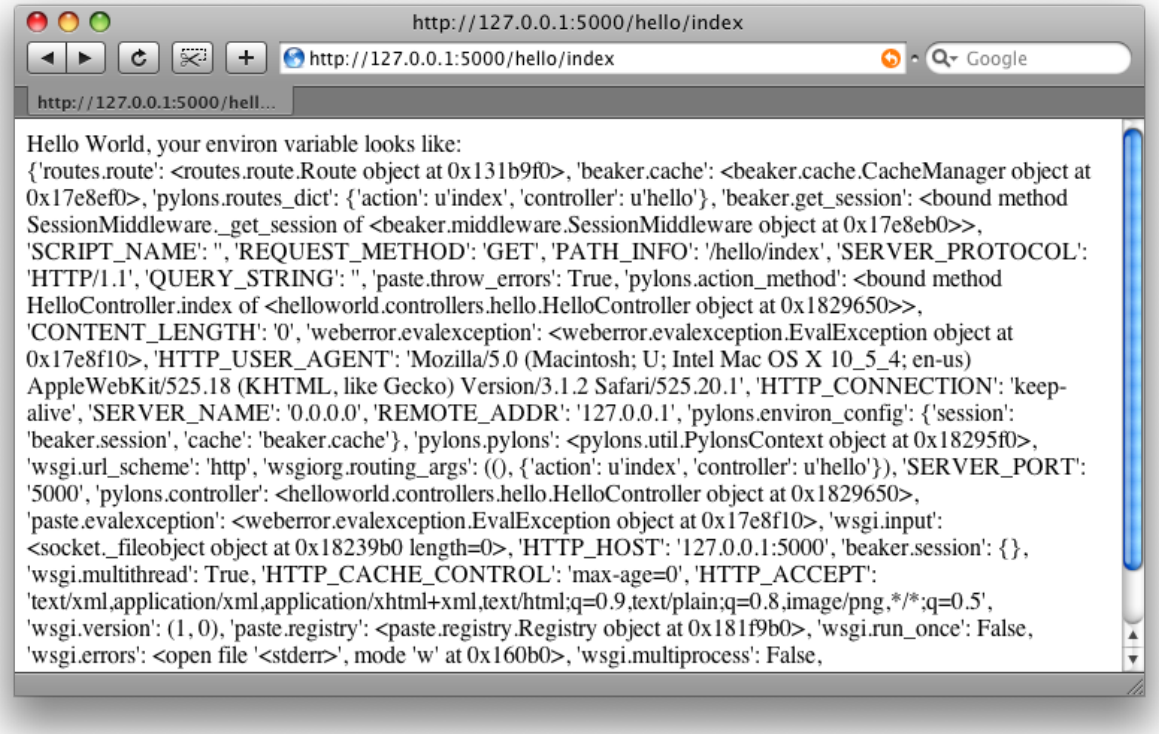
```
Hello World, the environ variable looks like: <br />

${request.environ}
```

The *request* variable in templates is used to get information about the current request. *Template globals* lists all the variables Pylons makes available for use in templates.

Next, update the `controllers/hello.py` module so that the index method is as follows:

```
class HelloController(BaseController):

    def index(self):
        return render('/hello.mako')
```

Refreshing the page in the browser will now look similar to this:



CHAPTER

TWO

CONCEPTS OF PYLONS

Understanding the basic concepts of Pylons, the flow of a request and response through the stack and how Pylons operates makes it easier to customize when needed, in addition to clearing up misunderstandings about why things behave the way they do.

This section acts as a basic introduction to the concept of a *WSGI* application, and *WSGI Middleware* in addition to showing how Pylons utilizes them to assemble a complete working web framework.

To follow along with the explanations below, create a project following the *Getting Started* Guide.

2.1 The 'Why' of a Pylons Project

A new Pylons project works a little differently than in many other web frameworks. Rather than loading the framework, which then finds a new projects code and runs it, Pylons creates a Python package that does the opposite. That is, when its run, it imports objects from Pylons, assembles the WSGI Application and stack, and returns it.

If desired, a new project could be completely cleared of the Pylons imports and run any arbitrary WSGI application instead. This is done for a greater degree of freedom and flexibility in building a web application that works the way the developer needs it to.

By default, the project is configured to use standard components that most developers will need, such as sessions, template engines, caching, high level request and response objects, and an *ORM*. By having it all setup in the project (rather than hidden away in 'framework' code), the developer is free to tweak and customize as needed.

In this manner, Pylons has setup a project with its *opinion* of what may be needed by the developer, but the developer is free to use the tools needed to accomplish the projects goals. Pylons offers an unprecedented level of customization by exposing its functionality through the project while still maintaining a remarkable amount of simplicity by retaining a single standard interface between core components (*WSGI*).

2.2 WSGI Applications

WSGI is a basic specification known as **PEP 333**, that describes a method for interacting with a HTTP server. This involves a way to get access to HTTP headers from the request, and how set HTTP headers and return content on the way back out.

A 'Hello World' WSGI Application:

```
def simple_app(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/html')])
    return ['<html><body>Hello World</body></html>']
```

This WSGI application does nothing but set a 200 status code for the response, set the HTTP 'Content-type' header, and return some HTML.

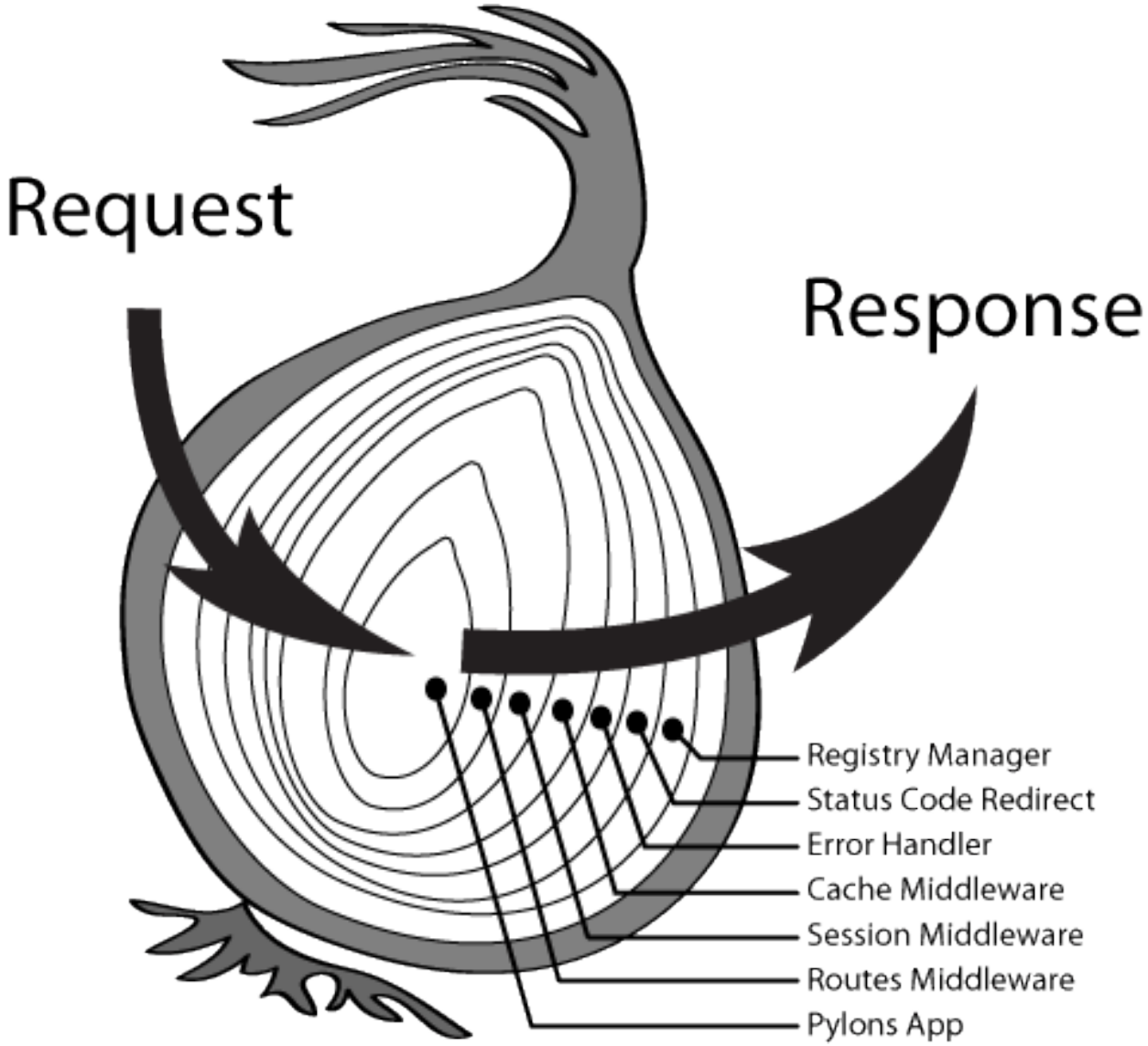
The WSGI specification lays out a set of keys that will be set in the environ dict.

The WSGI interface, that is, this method of calling a function (or method of a class) with two arguments, and handling a response as shown above, is used throughout Pylons as a standard interface for passing control to the next component.

Inside a new project's `config/middleware.py`, the *make_app* function is responsible for creating a WSGI application, wrapping it in WSGI middleware (explained below) and returning it so that it may handle requests from a HTTP server.

2.3 WSGI Middleware

Within `config/middleware.py` a Pylons application is wrapped in successive layers which add functionality. The process of wrapping the Pylons application in middleware results in a structure conceptually similar to the layers in an onion.



Once the middleware has been used to wrap the Pylons application, the make_app function returns the completed app with the following structure (outermost layer listed first):

```
Registry Manager
    Status Code Redirect
        Error Handler
            Cache Middleware
                Session Middleware
                    Routes Middleware
                        Pylons App (WSGI Application)
```

WSGI middleware is used extensively in Pylons to add functionality to the base WSGI application. In Pylons, the 'base' WSGI Application is the `PylonsApp`. It's responsible for looking in the *environ* dict that was passed in (from the Routes Middleware).

To see how this functionality is created, consider a small class that looks at the *HTTP_REFERER* header to see if it's Google:

```
class GoogleRefMiddleware(object):
    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        environ['google'] = False
        if 'HTTP_REFERER' in environ:
            if environ['HTTP_REFERER'].startswith('http://google.com'):
                environ['google'] = True
        return self.app(environ, start_response)
```

This is considered WSGI Middleware as it still can be called and returns like a WSGI Application, however, it's adding something to environ, and then calls a WSGI Application that it is initialized with. That's how the layers are built up in the *WSGI Stack* that is configured for a new Pylons project.

Some of the layers, like the Session, Routes, and Cache middleware, only add objects to the *environ* dict, or add HTTP headers to the response (the Session middleware for example adds the session cookie header). Others, such as the Status Code Redirect, and the Error Handler may fully intercept the request entirely, and change how it's responded to.

2.4 Controller Dispatch

When the request passes down the middleware, the incoming URL gets parsed in the RoutesMiddleware, and if it matches a URL (See *URL Configuration*), the information about the controller that should be called is put into the *environ* dict for use by `PylonsApp`.

The `PylonsApp` then attempts to find a controller in the `controllers` directory that matches the name of the controller, and searches for a class inside it by a similar scheme (controller name + 'Controller', ie, HelloController). Upon finding a controller, its then called like any other WSGI application using the same WSGI interface that `PylonsApp` was called with.

This is why the BaseController that resides in a project's `lib/base.py` module inherits from `WSGIController` and has a *__call__* method that takes the *environ* and *start_response*. The `WSGIController` locates a method in the class that corresponds to the *action* that Routes found, calls it, and returns the response completing the request.

2.5 Paster

Running the **paster** command all by itself will show the sets of commands it accepts:

```
$ paster
Usage: paster [paster_options] COMMAND [command_options]

Options:
  --version          show program's version number and exit
  --plugin=PLUGINS   Add a plugin to the list of commands (plugins are Egg
                     specs; will also require() the Egg)
  -h, --help         Show this help message

Commands:
  create          Create the file layout for a Python distribution
  grep            Search project for symbol
  help            Display help
  make-config     Install a package and create a fresh config file/directory
```

```
  points           Show information about entry points
  post             Run a request for the described application
  request          Run a request for the described application
  serve            Serve the described application
  setup-app        Setup an application, given a config file

pylons:
  controller       Create a Controller and accompanying functional test
  restcontroller   Create a REST Controller and accompanying functional test
  shell            Open an interactive shell with the Pylons app loaded
```

If **paster** is run inside of a Pylons project, this should be the output that will be printed. The last section, *pylons* will be absent if it is not run inside a Pylons project. This is due to a dynamic plugin system the **paster** script uses, to determine what sets of commands should be made available.

Inside a Pylons project, there is a directory ending in *.egg-info*, that has a `paster_plugins.txt` file in it. This file is looked for and read by the **paster** script, to determine what other packages should be searched dynamically for commands. Pylons makes several commands available for use in a Pylons project, as shown above.

2.6 Loading the Application

Running (and thus loading) an application is done using the **paster** command:

```
$ paster serve development.ini
```

This instructs the paster script to go into a 'serve' mode. It will attempt to load both a server and a WSGI application that should be served, by parsing the configuration file specified. It looks for a *[server]* block to determine what server to use, and an *[app]* block for what WSGI application should be used.

The basic egg block in the `development.ini` for a *helloworld* project:

```
[app:main]
use = egg:helloworld
```

That will tell paster that it should load the helloworld *egg* to locate a WSGI application. A new Pylons application includes a line in the `setup.py` that indicates what function should be called to make the WSGI application:

```
entry_points="""
[paste.app_factory]
main = helloworld.config.middleware:make_app

[paste.app_install]
main = pylons.util:PylonsInstaller
""",
```

Here, the *make_app* function is specified as the *main* WSGI application that Paste (the package that **paster** comes from) should use.

The *make_app* function from the project is then called, and the server (by default, a HTTP server) runs the WSGI application.

CHAPTER

THREE

PYLONS TUTORIALS

A small collection of relevant tutorials.

3.1 Quickwiki tutorial

3.1.1 Introduction

If you haven't done so already, please first read the *Getting Started* guide.

In this tutorial we are going to create a working wiki from scratch using Pylons 0.9.7 and SQLAlchemy. Our wiki will allow visitors to add, edit or delete formatted wiki pages.

3.1.2 Starting at the End

Pylons is designed to be easy for everyone, not just developers, so let's start by downloading and installing the finished QuickWiki in exactly the same way that end users of QuickWiki might do. Once we have explored its features we will set about writing it from scratch.

After you have installed Easy Install run these commands to install QuickWiki and create a config file:

```
$ easy_install QuickWiki==0.1.6
$ paster make-config QuickWiki test.ini
```

Next, ensure that the `sqlalchemy.url` variable in the `[app:main]` section of the configuration file (`development.ini`) specifies a value that is suitable for your setup. The data source name points to the database you wish to use.

Note: The default `sqlite:///%(here)s/quickwiki.db` uses a (file-based) SQLite database named `quickwiki.db` in the ini's top-level directory. This SQLite database will be created for you when running the **paster setup-app** command below, but you could also use MySQL, Oracle or PostgreSQL. Firebird and MS-SQL may also work. See the SQLAlchemy documentation for more information on how to connect to different databases. SQLite for example requires additional forward slashes in its URI, where the client/server databases should only use two. You will also need to make sure you have the appropriate Python driver for the database you wish to use. If you're using Python 2.5, a version of the pysqlite adapter is already included, so you can jump right in with the tutorial. You may need to get SQLite itself.

Finally create the database tables and serve the finished application:

```
$ paster setup-app test.ini
$ paster serve test.ini
```

That's it! Now you can visit http://127.0.0.1:5000 and experiment with the finished Wiki.

When you've finished, stop the server with `Control-C` so we can start developing our own version.

If you are interested in looking at the latest version of the QuickWiki source code it can be browsed online at http://www.knowledgetap.com/hg/QuickWiki or can be checked out using Mercurial:

```
$ hg clone http://www.knowledgetap.com/hg/QuickWiki
```

Note: To run the QuickWiki checked out from the repository, you'll need to first run **python setup.py develop** from the project's root directory. This will install its dependencies and generate Python Egg metadata in a `QuickWiki.egg-info` directory. The latter is required for the **paster** command (among other things) .

```
$ cd QuickWiki
$ python setup.py develop
```

3.1.3 Developing QuickWiki

If you skipped the "Starting at the End" section you will need to assure yourself that you have Pylons installed. See the *Getting Started*.

Then create your project:

```
$ paster create -t pylons QuickWiki
```

When prompted for which templating engine to use, simply hit enter for the default (Mako). When prompted for SQLAlchemy configuration, enter `True`.

Now let's start the server and see what we have:

```
$ cd QuickWiki
$ paster serve --reload development.ini
```

Note: We have started **paster serve** with the *`--reload`* option. This means any changes that we make to code will cause the server to restart (if necessary); your changes are immediately reflected on the live site.

Visit http://127.0.0.1:5000 where you will see the introduction page. Now delete the file `public/index.html` so we can see the front page of the wiki instead of this welcome page. If you now refresh the page, the Pylons built-in error document support will kick in and display an `Error 404` page, indicating the file could not be found. We'll setup a controller to handle this location later.

3.1.4 The Model

Pylons uses a Model-View-Controller architecture; we'll start by creating the model. We could use any system we like for the model, including SQLAlchemy or SQLObject. Optional SQLAlchemy integration is provided for new Pylons projects, which we enabled when creating the project, and thus we'll be using SQLAlchemy for the QuickWiki.

Note: SQLAlchemy is a powerful Python SQL toolkit and Object Relational Mapper (ORM) that is widely used by the Python community.

SQLAlchemy provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performance database access, adapted into a simple and Pythonic domain language. It has full and detailed documentation available on the SQLAlchemy website: http://sqlalchemy.org/docs/.

The most basic way of using SQLAlchemy is with explicit sessions where you create `Session` objects as needed.

Pylons applications typically employ a slightly more sophisticated setup, using SQLAlchemy's "contextual" thread-local sessions created via the `sqlalchemy.orm.scoped_session()` function. With this configuration, the application can use a single `Session` instance per web request, avoiding the need to pass it around explicitly. Instantiating a new scoped `Session` will actually find an existing one in the current thread if available. Pylons has setup a `Session` for us in the `model/meta.py` file. For further details, refer to the SQLAlchemy documentation on the Session.

Note: It is important to recognize the difference between SQLAlchemy's (or possibly another DB abstraction layer's) `Session` object and Pylons' standard *session* (with a lowercase 's') for web requests. See `beaker` for more on the latter. It is customary to reference the database session by `model.Session` or (more recently) `Session` outside of model classes.

The default imports already present in `model/__init__.py` provide SQLAlchemy objects such as the `sqlalchemy` module (aliased as `sa`) as well as the `metadata` object. `metadata` is used when defining and managing tables. Next we'll use these to build our wiki's model: we can remove the commented out Foo example and add the following to the end of the `model/__init__.py` file:

```
pages_table = sa.Table('pages', meta.metadata,
                sa.Column('title', sa.types.Unicode(40), primary_key=True),
                sa.Column('content', sa.types.Unicode(), default='')
                )
```

We've defined a table called `pages` which has two columns, `title` (the primary key) and `content`.

Note: SQLAlchemy also supports reflecting table information directly from a database. If we had already created the `pages` table in our database, SQLAlchemy could have constructed the `pages_table` object for us via the `autoload=True` parameter in place of the `Column` definitions, like this:

```
pages_table = sa.Table('pages', meta.metadata, autoload=True
                       autoload_with_engine)
```

The ideal way to create autoloaded tables is within the `init_model()` function (lazily), so the database isn't accessed when simply importing the `model` package. See SQLAlchemy table reflection documentation for more information.

Note: A primary key is a unique ID for each row in a database table. In the example above we are using the page title as a natural primary key. Some prefer to integer primary keys for all tables, so-called surrogate primary keys. The author of this tutorial uses both methods in his own code and is not advocating one method over the other, what's important is to choose the best database structure for your application. See the Pylons Cookbook for a quick general overview of relational databases if you're not familiar with these concepts.

A core philosophy of ORMs is that tables and domain classes are different beasts. So next we'll create the Python class that represents the pages of our wiki, and map these domain objects to rows in the `pages`

table via the sqlalchemy.orm.mapper() function. In a more complex application, you could break out model classes into separate .py files in your model directory, but for sake of simplicity in this case, we'll just stick to __init__.py.

Add this to the bottom of model/__init__.py:

```
class Page(object):

    def __init__(self, title, content=None):
        self.title = title
        self.content = content

    def __unicode__(self):
        return self.title

    __str__ = __unicode__

orm.mapper(Page, pages_table)
```

A Page object represents a row in the pages table, so self.title and self.content will be the values of the title and content columns.

Looking ahead, our wiki could use a way of marking up the content field into HTML. Also, any 'WikiWords' (words made by joining together two or more capitalized words) should be converted to hyperlinks to wiki pages.

We can use Python's docutils library to allow marking up content as reStructuredText. So next we'll add a method to our Page class that formats content as HTML and converts the WikiWords to hyperlinks. Add the following at the top of the model/__init__.py file:

```
import logging
import re
import sets
from docutils.core import publish_parts

from pylons import url
from quickwiki.lib.helpers import link_to
from quickwiki.model import meta

log = logging.getLogger(__name__)

# disable docutils security hazards:
# http://docutils.sourceforge.net/docs/howto/security.html
SAFE_DOCUTILS = dict(file_insertion_enabled=False, raw_enabled=False)
wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+)", re.UNICODE)
```

then add a get_wiki_content() method to the Page class:

```
class Page(object):

    def __init__(self, title, content=None):
        self.title = title
        self.content = content

    def get_wiki_content(self):
        """Convert reStructuredText content to HTML for display, and
        create links for WikiWords
        """
        content = publish_parts(self.content, writer_name='html',
                                settings_overrides=SAFE_DOCUTILS)['html_body']
```

```
        titles = sets.Set(wikiwords.findall(content))
        for title in titles:
            title_url = url(controller='pages', action='show', title=title)
            content = content.replace(title, link_to(title, title_url))
        return content

    def __unicode__(self):
        return self.title

    __str__ = __unicode__
```

The `Set` object provides us with only unique WikiWord names, so we don't try replacing them more than once (a "wikiword" is of course defined by the regular expression set globally).

Note: Pylons uses a **Model View Controller** architecture and so the formatting of objects into HTML should properly be handled in the View, i.e. in a template. However in this example, converting reStructuredText into HTML in a template is inappropriate so we are treating the HTML representation of the content as part of the model. It also gives us the chance to demonstrate that SQLAlchemy domain classes are real Python classes that can have their own methods.

The `link_to()` and `url()` functions referenced in the controller code are respectively: a helper imported from the `webhelpers.html` module indirectly via `lib/helpers.py`, and a utility function imported directly from the `pylons` module. They are utilities for creating links to specific controller actions. In this case we have decided that all WikiWords should link to the `show()` action of the `pages` controller which we'll create later. However, we need to ensure that the `link_to()` function is made available as a helper by adding an import statement to `lib/helpers.py`:

```
"""Helper functions

Consists of functions to typically be used within templates, but also
available to Controllers. This module is available to templates as 'h'.
"""
from webhelpers.html.tags import *
```

Since we have used docutils and SQLAlchemy, both third party packages, we need to edit our `setup.py` file so that anyone installing QuickWiki with Easy Install will automatically have these dependencies installed too. Edit your `setup.py` in your project root directory and add a docutils entry to the `install_requires` line (there will already be one for SQLAlchemy):

```
install_requires=[
    "Pylons>=0.9.7",
    "SQLAlchemy>=0.5",
    "docutils==0.4",
],
```

While we are we are making changes to `setup.py` we might want to complete some of the other sections too. Set the version number to 0.1.6 and add a description and URL which will be used on PyPi when we release it:

```
version='0.1.6',
description='QuickWiki - Pylons 0.9.7 Tutorial application',
url='http://docs.pylonshq.com/tutorials/quickwiki_tutorial.html',
```

We might also want to make a full release rather than a development release in which case we would remove the following lines from `setup.cfg`:

```
[egg_info]
tag_build = dev
tag_svn_revision = true
```

To test the automatic installation of the dependencies, run the following command which will also install docutils and SQLAlchemy if you don't already have them:

```
$ python setup.py develop
```

Note: The command **python setup.py develop** installs your application in a special mode so that it behaves exactly as if it had been installed as an egg file by an end user. This is really useful when you are developing an application because it saves you having to create an egg and install it every time you want to test a change.

3.1.5 Application Setup

Edit `websetup.py`, used by the **paster setup-app** command, to look like this:

```
"""Setup the QuickWiki application"""
import logging

from quickwiki import model
from quickwiki.config.environment import load_environment
from quickwiki.model import meta

log = logging.getLogger(__name__)

def setup_app(command, conf, vars):
    """Place any commands to setup quickwiki here"""
    load_environment(conf.global_conf, conf.local_conf)

    # Create the tables if they don't already exist
    log.info("Creating tables...")
    meta.metadata.create_all(bind=meta.engine)
    log.info("Successfully set up.")

    log.info("Adding front page data...")
    page = model.Page(title=u'FrontPage',
                      content=u'**Welcome** to the QuickWiki front page!')
    meta.Session.add(page)
    meta.Session.commit()
    log.info("Successfully set up.")
```

You can see that `config/environment.py`'s `load_environment()` function is called (which calls `model/__init__.py`'s `init_model()` function), so our engine is ready for binding and we can import the model. A SQLAlchemy `MetaData` object – which provides some utility methods for operating on database schema – usually needs to be connected to an engine, so the line

```
meta.metadata.bind = meta.engine
```

does exactly that and then

```
model.metadata.create_all(checkfirst=True)
```

uses the connection we've just set up and, creates the table(s) we've defined ... if they don't already exist. After the tables are created, the other lines add some data for the simple front page to our wiki.

By default, SQLAlchemy specifies `autocommit=False` when creating the `Session`, which means that operations will be wrapped in a transaction and `commit()`'ed atomically (unless your DB doesn't support transactions, like MySQL's default MyISAM tables – but that's beyond the scope of this tutorial).

The database SQLAlchemy will use is specified in the `ini` file, under the `[app:main]` section, as `sqlalchemy.url`. We'll customize the `sqlalchemy.url` value to point to a SQLite database named `quickwiki.db` that will reside in your project's root directory. Edit the `development.ini` file in the root directory of your project:

Note: If you've decided to use a different database other than SQLite, see the SQLAlchemy note in the Starting at the End section for information on supported database URIs.

```
[app:main]
use = egg:QuickWiki
#...
# Specify the database for SQLAlchemy to use.
# SQLAlchemy database URL
sqlalchemy.url = sqlite:///%(here)s/quickwiki.db
```

You can now run the **paster setup-app** command to setup your tables in the same way an end user would, remembering to drop and recreate the database if the version tested earlier has already created the tables:

```
$ paster setup-app development.ini
```

You should see the SQL sent to the database as the default `development.ini` is setup to log SQLAlchemy's SQL statements.

At this stage you will need to ensure you have the appropriate Python database drivers for the database you chose, otherwise you might find SQLAlchemy complains it can't get the DBAPI module for the dialect it needs.

You should also edit `quickwiki/config/deployment.ini_tmpl` so that when users run **paster make-config** the configuration file that is produced for them will also use `quickwiki.db`. In the `[app:main]` section:

```
# Specify the database for SQLAlchemy to use.
sqlalchemy.url = sqlite:///%(here)s/quickwiki.db
```

3.1.6 Templates

Note: Pylons uses the Mako templating engine by default, although as is the case with most aspects of Pylons, you are free to deviate from the default if you prefer.

In our project we will make use of the Mako inheritance feature. Add the main page template in `templates/base.mako`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>QuickWiki</title>
    ${h.stylesheet_link('/quick.css')}
  </head>

  <body>
```

```
    <div class="content">
      <h1 class="main">${self.header()}</h1>
      ${next.body()}\
      <p class="footer">
        Return to the ${h.link_to('FrontPage', url('FrontPage'))}
        | ${h.link_to('Edit ' + c.title, url('edit_page', title=c.title))}
      </p>
    </div>
  </body>
</html>
```

We'll setup all our other templates to inherit from this one: they will be automatically inserted into the `${next.body()}` line. Thus the whole page will be returned when we call the `render()` global from our controller. This lets us easily apply a consistent theme to all our templates.

If you are interested in learning some of the features of Mako templates have a look at the comprehensive Mako Documentation. For now we just need to understand that `next.body()` is replaced with the child template and that anything within `${...}` brackets is executed and replaced with the result. By default, the replacement content is HTML-escaped in order to meet modern standards of basic protection from accidentally making the app vulnerable to XSS exploit.

This `base.mako` also makes use of various helper functions attached to the `h` object. These are described in the WebHelpers documentation. We need to add some helpers to the `h` by importing them in the `lib/helpers.py` module (some are for later use):

```
"""Helper functions

Consists of functions to typically be used within templates, but also
available to Controllers. This module is available to templates as 'h'.
"""
from webhelpers.html import literal
from webhelpers.html.tags import *
from webhelpers.html.secure_form import secure_form
```

Note that the `helpers` module is available to templates as 'h', this is a good place to import or define directly any convenience functions that you want to make available to all templates.

3.1.7 Routing

Before we can add the actions we want to be able to route the requests to them correctly. Edit `config/routing.py` and adjust the 'Custom Routes' section to look like this:

```
# CUSTOM ROUTES HERE

map.connect('home', '/', controller='pages', action='show',
            title='FrontPage')
map.connect('pages', '/pages', controller='pages', action='index')
map.connect('show_page', '/pages/show/{title}', controller='pages',
            action='show')
map.connect('edit_page', '/pages/edit/{title}', controller='pages',
            action='edit')
map.connect('save_page', '/pages/save/{title}', controller='pages',
            action='save', conditions=dict(method='POST'))
map.connect('delete_page', '/pages/delete', controller='pages',
            action='delete')

# A bonus example - the specified defaults allow visiting
```

```
# example.com/FrontPage to view the page titled 'FrontPage':
map.connect('/{title}', controller='pages', action='show')

return map
```

Note that the default route has been replaced. This tells Pylons to route the root URL `/` to the `show()` method of the `PageController` class in `controllers/pages.py` and specify the `title` argument as `'FrontPage'`. It also says that any URL of the form `/SomePage` should be routed to the same method but the `title` argument will contain the value of the first part of the URL, in this case `SomePage`. Any other URLs that can't be matched by these maps are routed to the error controller as usual where they will result in a 404 error page being displayed.

One of the main benefits of using the Routes system is that you can also create URLs automatically, simply by specifying the routing arguments. For example if I want the URL for the page `FrontPage` I can create it with this code:

```
url(title='FrontPage')
```

Although the URL would be fairly simple to create manually, with complicated URLs this approach is much quicker. It also has the significant advantage that if you ever deploy your Pylons application at a URL other than `/`, all the URLs will be automatically adjusted for the new path without you needing to make any manual modifications. This flexibility is a real advantage.

Full information on the powerful things you can do to route requests to controllers and actions can be found in the Routes manual.

3.1.8 Controllers

Quick Recap: We've setup the model, configured the application, added the routes and setup the base template in `base.mako`, now we need to write the application logic and we do this with controllers. In your project's root directory, add a controller called `pages` to your project with this command:

```
$ paster controller pages
```

If you are using Subversion, this will automatically be detected and the new controller and tests will be automatically added to your subversion repository.

We are going to need the following actions:

`show(self, title)` displays a page based on the title

`edit(self, title)` displays a from for editing the page `title`

`save(self, title)` save the page `title` and show it with a saved message

`index(self)` lists all of the titles of the pages in the database

`delete(self, title)` deletes a page

show()

Let's get to work on the new controller in `controllers/pages.py`. First we'll import the `Page` class from our `model`, and the `Session` class from the `model.meta` module. We'll also import the `wikiwords` regular expression object, which we'll use in the `show()` method. Add this line with the imports at the top of the file:

```
from quickwiki.model import Page, wikiwords
from quickwiki.model.meta import Session
```

Next we'll add the convenience method `__before__()` to the `PagesController`, which is a special method Pylons always calls before calling the actual action method. We'll have `__before__()` obtain and make available the relevant query object from the database, ready to be queried. Our other action methods will need this query object, so we might as well create it one place.

```
class PagesController(BaseController):

    def __before__(self):
        self.page_q = Session.query(Page)
```

Now we can query the database using the query expression language provided by SQLAlchemy. Add the following `show()` method to `PagesController`:

```
def show(self, title):
    page = self.page_q.filter_by(title=title).first()
    if page:
        c.content = page.get_wiki_content()
        return render('/pages/show.mako')
    elif wikiwords.match(title):
        return render('/pages/new.mako')
    abort(404)
```

Add a template called `templates/pages/show.mako` that looks like this:

```
<%inherit file="/base.mako"/>\

<%def name="header()">${c.title}</%def>

${h.literal(c.content)}
```

This template simply displays the page title and content.

Note: Pylons automatically assigns all the action parameters to the Pylons context object `c` so that you don't have to assign them yourself. In this case, the value of `title` will be automatically assigned to `c.title` so that it can be used in the templates. We assign `c.content` manually in the controller.

We also need a template for pages that don't already exist. The template needs to display a message and link to the `edit()` action so that they can be created. Add a template called `templates/new.mako` that looks like this:

```
<%inherit file="/base.mako"/>\

<%def name="header()">${c.title}</%def>

<p>This page doesn't exist yet.
  <a href="${url('edit_page', title=c.title)}">Create the page</a>.
</p>
```

At this point we can test our QuickWiki to see how it looks. If you don't already have a server running, start it now with:

```
$ paster serve --reload development.ini
```

We can spruce up the appearance of page a little by adding the stylesheet we linked to in the `templates/base.mako` file earlier. Add the file `public/quick.css` with the following content and refresh the page to reveal a better looking wiki:

```
body {
  background-color: #888;
  margin: 25px;
}

div.content {
  margin: 0;
  margin-bottom: 10px;
  background-color: #d3e0ea;
  border: 5px solid #333;
  padding: 5px 25px 25px 25px;
}

h1.main {
  width: 100%;
}

p.footer{
  width: 100%;
  padding-top: 8px;
  border-top: 1px solid #000;
}

a {
  text-decoration: none;
}

a:hover {
  text-decoration: underline;
}
```

When you run the example you will notice that the word `QuickWiki` has been turned into a hyperlink by the `get_wiki_content()` method we added to our `Page` domain object earlier. You can click the link and will see an example of the new page screen from the `new.mako` template. If you follow the `Create the page` link you will see the Pylons automatic error handler kick in to tell you `Action edit is not implemented`. Well, we better write it next, but before we do, have a play with the *Interactive debugging*, try clicking on the + or >> arrows and you will be able to interactively debug your application. It is a tremendously useful tool.

edit()

To edit the wiki page we need to get the content from the database without changing it to HTML to display it in a simple form for editing. Add the `edit()` action:

```
def edit(self, title):
    page = self.page_q.filter_by(title=title).first()
    if page:
        c.content = page.content
    return render('/pages/edit.mako')
```

and then create the `templates/edit.mako` file:

```
<%inherit file="/base.mako"/>\

<%def name="header()">Editing ${c.title}</%def>

${h.secure_form(url('save_page', title=c.title))}
```

```
  ${h.textarea(name='content', rows=7, cols=40, content=c.content)} <br />
  ${h.submit(value='Save changes', name='commit')}
${h.end_form()}
```

Note: You may have noticed that we only set `c.content` if the page exists but that it is accessed in `h.text_area()` even for pages that don't exist and yet it doesn't raise an `AttributeError`.

We are making use of the fact that the `c` object returns an empty string `""` for any attribute that is accessed which doesn't exist. This can be a very useful feature of the `c` object, but can catch you on occasions where you don't expect this behavior. It can be disabled by setting `config['pylons.strict_c'] = True` in your project's `config/environment.py`.

We are making use of the `h` object to create our form and field objects. This saves a bit of manual HTML writing. The form submits to the `save()` action to save the new or updated content so let's write that next.

save()

The first thing the `save()` action has to do is to see if the page being saved already exists. If not it creates it with `page = model.Page(title)`. Next it needs the updated content. In Pylons you can get request parameters from form submissions via `GET` and `POST` requests from the appropriately named `request` object. For form submissions from *only* `GET` or `POST` requests, use `request.GET` or `request.POST`. Only `POST` requests should generate side effects (like changing data), so the save action will only reference `request.POST` for the parameters.

Then add the `save()` action:

```
@authenticate_form
def save(self, title):
    page = self.page_q.filter_by(title=title).first()
    if not page:
        page = Page(title)
    # In a real application, you should validate and sanitize
    # submitted data throughly! escape is a minimal example here.
    page.content = escape(request.POST.getone('content'))
    Session.add(page)
    Session.commit()
    flash('Successfully saved %s!' % title)
    redirect_to('show_page', title=title)
```

Note: `request.POST` is a MultiDict object: an ordered dictionary that may contain multiple values for each key. The MultiDict will always return one value for any existing key via the normal dict accessors `request.POST[key]` and `request.POST.get()`. When multiple values are expected, use the `request.POST.getall()` method to return all values in a list. `request.POST.getone()` ensures one value for key was sent, raising a `KeyError` when there are 0 or more than 1 values.

The `@authenticate_form()` decorator that appears immediately before the `save()` action checks the value of the hidden form field placed there by the `secure_form()` helper that we used in `templates/edit.mako` to create the form. The hidden form field carries an authorization token for prevention of certain Cross-site request forgery (CSRF) attacks.

Upon a successful save, we want to redirect back to the `show()` action and 'flash' a `Successfully saved` message at the top of the page. 'Flashing' a status message immediately after an action is a common requirement, and the *WebHelpers* package provides the `webhelpers.pylonslib.Flash` class that makes it easy. To utilize it, we'll create a flash object at the bottom of our `lib/helpers.py` module:

```
from webhelpers.pylonslib import Flash as _Flash

flash = _Flash()
```

And import it into our `controllers/pages.py`. Our new `show()` method is escaping the content via Python's `cgi.escape()` function, so we need to import that too, and also `@authenticate_form()`.

```
from cgi import escape

from pylons.decorators.secure import authenticate_form

from quickwiki.lib.helpers import flash
```

And finally utilize the `flash` object in our `templates/base.mako` template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>QuickWiki</title>
    ${h.stylesheet_link('/quick.css')}
  </head>

  <body>
    <div class="content">
      <h1 class="main">${self.header()}</h1>

      <% flashes = h.flash.pop_messages() %>
      % if flashes:
        % for flash in flashes:
        <div id="flash">
          <span class="message">${flash}</span>
        </div>
        % endfor
      % endif

      ${next.body()}\
      <p class="footer">
        Return to the ${h.link_to('FrontPage', url('FrontPage'))}
        | ${h.link_to('Edit ' + c.title, url('edit_page', title=c.title))}
      </p>
    </div>
  </body>
</html>
```

And add the following to the `public/quick.css` file:

```
div#flash .message {
  color: orangered;
}
```

The `%` syntax is used for control structures in mako – conditionals and loops. You must 'close' them with an 'end' tag as shown here. At this point we have a fully functioning wiki that lets you create and edit pages and can be installed and deployed by an end user with just a few simple commands.

Visit http://127.0.0.1:5000 and have a play.

It would be nice to get a title list and to be able to delete pages, so that's what we'll do next!

index()

Add the `index()` action:

```
def index(self):
    c.titles = [page.title for page in self.page_q.all()]
    return render('/pages/index.mako')
```

The `index()` action simply gets all the pages from the database. Create the `templates/index.mako` file to display the list:

```
<%inherit file="/base.mako"/>\

<%def name="header()">Title List</%def>

${h.secure_form(url('delete_page'))}

<ul id="titles">
  % for title in c.titles:
  <li>
    ${h.link_to(title, url('show_page', title=title))} -
    ${h.checkbox('title', title)}
  </li>
  % endfor
</ul>

${h.submit('delete', 'Delete')}

${h.end_form()}
```

This displays a form listing a link to all pages along with a checkbox. When submitted, the selected titles will be sent to a `delete()` action we'll create in the next step.

We need to edit `templates/base.mako` to add a link to the title list in the footer, but while we're at it, let's introduce a Mako function to make the footer a little smarter. Edit `base.mako` like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>QuickWiki</title>
    ${h.stylesheet_link('/quick.css')}
  </head>

  <body>
    <div class="content">
      <h1 class="main">${self.header()}</h1>

      <% flashes = h.flash.pop_messages() %>
      % if flashes:
        % for flash in flashes:
        <div id="flash">
          <span class="message">${flash}</span>
        </div>
        % endfor
      % endif

      ${next.body()}\
```

```
      <p class="footer">
        ${self.footer(request.environ['pylons.routes_dict']['action'])}\
      </p>
    </div>
  </body>
</html>

## Don't show links that are redundant for particular pages
<%def name="footer(action)">\
  Return to the ${h.link_to('FrontPage', url('home'))}
  % if action == "index":
    <% return %>
  % endif
  % if action != 'edit':
    | ${h.link_to('Edit ' + c.title, url('edit_page', title=c.title))}
  % endif
  | ${h.link_to('Title List', url('pages'))}
</%def>
```

The `<%def name="footer(action)">` creates a Mako function for display logic. As you can see, the function builds the HTML for the footer, but doesn't display the 'Edit' link when you're on the 'Title List' page or already on an edit page. It also won't show a 'Title List' link when you're already on that page. The `<% ... %>` tags shown on the `return` statement are the final new piece of Mako syntax: they're used much like the `${...}` tags, but for arbitrary Python code that does not directly render HTML. Also, the double hash (`##`) denotes a single-line comment in Mako.

So the `footer()` function is called in place of our old 'static' footer markup. We pass it a value from `pylons.routes_dict` which holds the name of the action for the current request. The trailing \ character just tells Mako not to render an extra newline.

If you visit http://127.0.0.1:5000/pages you should see the full titles list and you should be able to visit each page.

delete()

We need to add a `delete()` action that deletes pages submitted from `templates/index.mako`, then returns us back to the list of titles (excluding those that were deleted):

```
@authenticate_form
def delete(self):
    titles = request.POST.getall('title')
    pages = self.page_q.filter(Page.title.in_(titles))
    for page in pages:
        Session.delete(page)
    Session.commit()
    # flash only after a successful commit
    for title in titles:
        flash('Deleted %s.' % title)
    redirect_to('pages')
```

Again we use the `@authenticate_form()` decorator along with `secure_form()` used in `templates/index.mako`. We're expecting potentially multiple titles, so we use `request.POST.getall()` to return a list of titles. The titles are used to identify and load the `Page` objects, which are then deleted.

We use the SQL `IN` operator to match multiple titles in one query. We can do this via the more flexible `filter()` method which can accept an `in_()` clause created via the title column's attribute.

The `filter_by()` method we used in previous methods is a shortcut for the most typical filtering clauses. For example, the `show()` method's:

```
self.page_q.filter_by(title=title)
```

is equivalent to:

```
self.page_q.filter(Page.title == title)
```

After deleting the pages, the changes are committed, and only after successfully committing do we flash deletion messages. That way if there was a problem with the commit no flash messages are shown. Finally we redirect back to the index page, which re-renders the list of remaining titles.

Visit http://127.0.0.1:5000/index and have a go at deleting some pages. You may need to go back to the FrontPage and create some more if you get carried away!

That's it! A working, production-ready wiki in 20 mins. You can visit http://127.0.0.1:5000/ once more to admire your work.

3.1.9 Publishing the Finished Product

After all that hard work it would be good to distribute the finished package wouldn't it? Luckily this is really easy in Pylons too. In the project root directory run this command:

```
$ python setup.py bdist_egg
```

This will create an egg file in the `dist` directory which contains everything anyone needs to run your program. They can install it with:

```
$ easy_install QuickWiki-0.1.6-py2.5.egg
```

You should probably make eggs for each version of Python your users might require by running the above commands with both Python 2.4 and 2.5 to create both versions of the eggs.

If you want to register your project with PyPi at http://www.python.org/pypi you can run the command below. *Please only do this with your own projects though because QuickWiki has already been registered!*

```
$ python setup.py register
```

Warning: The PyPi authentication is very weak and passwords are transmitted in plain text. Don't use any sign in details that you use for important applications as they could be easily intercepted.

You will be asked a number of questions and then the information you entered in `setup.py` will be used as a basis for the page that is created.

Now visit http://www.python.org/pypi to see the new index with your new package listed.

Note: A CheeseShop Tutorial has been written and full documentation on setup.py is available from the Python website. You can even use reStructuredText in the `description` and `long_description` areas of `setup.py` to add formatting to the pages produced on PyPi (PyPi used to be called "the CheeseShop"). There is also another tutorial here.

Finally you can sign in to PyPi with the account details you used when you registered your application and upload the eggs you've created. If that seems too difficult you can even use this command which should be run for each version of Python supported to upload the eggs for you:

```
$ python setup.py bdist_egg upload
```

Before this will work you will need to create a `.pypirc` file in your home directory containing your username and password so that the **upload** command knows who to sign in as. It should look similar to this:

```
[server-login]
username: james
password: password
```

Note: This works on windows too but you will need to set your `HOME` environment variable first. If your home directory is `C:Documents and SettingsJames` you would put your `.pypirc` file in that directory and set your `HOME` environment variable with this command:

```
> SET HOME=C:\Documents and Settings\James
```

You can now use the **python setup.py bdist_egg upload** as normal.

Now that the application is on PyPi anyone can install it with the **easy_install** command exactly as we did right at the very start of this tutorial.

3.1.10 Security

A final word about security.

> **Warning:** Always set `debug = false` in configuration files for production sites and make sure your users do too.

You should NEVER run a production site accessible to the public with debug mode on. If there was a problem with your application and an interactive error page was shown, the visitor would be able to run any Python commands they liked in the same way you can when you are debugging. This would obviously allow them to do all sorts of malicious things so it is very important you turn off interactive debugging for production sites by setting `debug = false` in configuration files and also that you make users of your software do the same.

3.1.11 Summary

We've gone through the whole cycle of creating and distributing a Pylons application looking at setup and configuration, routing, models, controllers and templates. Hopefully you have an idea of how powerful Pylons is and, once you get used to the concepts introduced in this tutorial, how easy it is to create sophisticated, distributable applications with Pylons.

That's it, I hope you found the tutorial useful. You are encouraged to email any comments to the Pylons mailing list where they will be welcomed.

3.1.12 Thanks

A big thanks to Ches Martin for updating this document and the QuickWiki project for Pylons 0.9.6 / Pylons 0.9.7 / QuickWiki 0.1.5 / QuickWiki 0.1.6, Graham Higgins, and others in the Pylons community who contributed bug fixes and suggestions.

3.1.13 Todo

- Provide **paster shell** examples
- Incorporate testing into the tutorial
- Explain Ches's `validate_title()` method in the actual QuickWiki project
- Provide snapshots of every file modified at each step, to help resolve mistakes

3.2 Understanding Unicode

If you've ever come across text in a foreign language that contains lots of `????` characters or have written some Python code and received a message such as `UnicodeDecodeError: 'ascii' codec can't decode byte 0xff in position 6: ordinal not in range(128)` then you have run into a problem with character sets, encodings, Unicode and the like.

The truth is that many developers are put off by Unicode because most of the time it is possible to muddle through rather than take the time to learn the basics. To make the problem worse if you have a system that manages to fudge the issues and just about work and then start trying to do things properly with Unicode it often highlights problems in other parts of your code.

The good news is that Python has great Unicode support, so the rest of this article will show you how to correctly use Unicode in Pylons to avoid unwanted `?` characters and `UnicodeDecodeErrors`.

3.2.1 What is Unicode?

When computers were first being used the characters that were most important were unaccented English letters. Each of these letters could be represented by a number between 32 and 127 and thus was born ASCII, a character set where space was 32, the letter "A" was 65 and everything could be stored in 7 bits.

Most computers in those days were using 8-bit bytes so people quickly realized that they could use the codes 128-255 for their own purposes. Different people used the codes 128-255 to represent different characters and before long these different sets of characters were also standardized into *code pages*. This meant that if you needed some non-ASCII characters in a document you could also specify a codepage which would define which extra characters were available. For example Israel DOS used a code page called 862, while Greek users used 737. This just about worked for Western languages provided you didn't want to write an Israeli document with Greek characters but it didn't work at all for Asian languages where there are many more characters than can be represented in 8 bits.

Unicode is a character set that solves these problems by uniquely defining *every* character that is used anywhere in the world. Rather than defining a character as a particular combination of bits in the way ASCII does, each character is assigned a *code point*. For example the word `hello` is made from code points `U+0048 U+0065 U+006C U+006C U+006F`. The full list of code points can be found at http://www.unicode.org/charts/.

There are lots of different ways of encoding Unicode code points into bits but the most popular encoding is UTF-8. Using UTF-8, every code point from 0-127 is stored in a single byte. Only code points 128 and above are stored using 2, 3, in fact, up to 6 bytes. This has the useful side effect that English text looks exactly the same in UTF-8 as it did in ASCII, because for every ASCII character with hexadecimal value 0xXY, the corresponding Unicode code point is U+00XY. This backwards compatibility is why if you are developing an application that is only used by English speakers you can often get away without handling characters properly and still expect things to work most of the time. Of course, if you use a different encoding such as UTF-16 this doesn't apply since none of the code points are encoded to 8 bits.

The important things to note from the discussion so far are that:

- Unicode can represent pretty much any character in any writing system in widespread use today
- Unicode uses code points to represent characters and the way these map to bits in memory depends on the encoding
- **The most popular encoding is UTF-8 which has several convenient properties**
    1. It can handle any Unicode code point
    2. A Unicode string is turned into a string of bytes containing no embedded zero bytes. This avoids byte-ordering issues, and means UTF-8 strings can be processed by C functions such as strcpy() and sent through protocols that can't handle zero bytes
    3. A string of ASCII text is also valid UTF-8 text
    4. UTF-8 is fairly compact; the majority of code points are turned into two bytes, and values less than 128 occupy only a single byte.
    5. If bytes are corrupted or lost, it's possible to determine the start of the next UTF-8-encoded code point and resynchronize.

Note: Since Unicode 3.1, some extensions have even been defined so that the defined range is now U+000000 to U+10FFFF (21 bits), and formally, the character set is defined as 31-bits to allow for future expansion. It is a myth that there are 65,536 Unicode code points and that every Unicode letter can really be squeezed into two bytes. It is also incorrect to think that UTF-8 can represent less characters than UTF-16. UTF-8 simply uses a variable number of bytes for a character, sometimes just one byte (8 bits).

3.2.2 Unicode in Python

In Python Unicode strings are expressed as instances of the built-in `unicode` type. Under the hood, Python represents Unicode strings as either 16 or 32 bit integers, depending on how the Python interpreter was compiled.

The `unicode()` constructor has the signature `unicode(string[, encoding, errors])`. All of its arguments should be 8-bit strings. The first argument is converted to Unicode using the specified encoding; if you leave off the encoding argument, the ASCII encoding is used for the conversion, so characters greater than 127 will be treated as errors:

```
>>> unicode('hello')
u'hello'
>>> s = unicode('hello')
>>> type(s)
<type 'unicode'>
>>> unicode('hello' + chr(255))
Traceback (most recent call last):
File "<stdin>", line 1, in ?
UnicodeDecodeError: 'ascii' codec can't decode byte 0xff in position 6:
ordinal not in range(128)
```

The `errors` argument specifies what to do if the string can't be decoded to ascii. Legal values for this argument are `'strict'` (raise a `UnicodeDecodeError` exception), `'replace'` (replace the character that can't be decoded with another one), or `'ignore'` (just leave the character out of the Unicode result).

```
>>> unicode('\x80abc', errors='strict')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0:
ordinal not in range(128)
```

```
>>> unicode('\x80abc', errors='replace')
u'\ufffdabc'
>>> unicode('\x80abc', errors='ignore')
u'abc'
```

It is important to understand the difference between *encoding* and *decoding*. Unicode strings are considered to be the Unicode code points but any representation of the Unicode string has to be encoded to something else, for example UTF-8 or ASCII. So when you are converting an ASCII or UTF-8 string to Unicode you are *decoding* it and when you are converting from Unicode to UTF-8 or ASCII you are *encoding* it. This is why the error in the example above says that the ASCII codec cannot decode the byte `0x80` from ASCII to Unicode because it is not in the range(128) or 0-127. In fact `0x80` is hex for 128 which the first number outside the ASCII range. However if we tell Python that the character `0x80` is encoded with the `'latin-1'`, `'iso_8859_1'` or `'8859'` character sets (which incidentally are different names for the same thing) we get the result we expected:

```
>>> unicode('\x80', encoding='latin-1')
u'\x80'
```

Note: The character encodings Python supports are listed at http://docs.python.org/lib/standard-encodings.html

---

Unicode objects in Python have most of the same methods that normal Python strings provide. Python will try to use the `'ascii'` codec to convert strings to Unicode if you do an operation on both types:

```
>>> a = 'hello'
>>> b = unicode(' world!')
>>> print a + b
u'hello world!'
```

You can encode a Unicode string using a particular encoding like this:

```
>>> u'Hello World!'.encode('utf-8')
'Hello World!'
```

3.2.3 Unicode Literals in Python Source Code

In Python source code, Unicode literals are written as strings prefixed with the 'u' or 'U' character:

```
>>> u'abcdefghijk'
>>> U'lmnopqrstuv'
```

You can also use `"`, `"""` `` ` `` or `"'` versions too. For example:

```
>>> u"""This
... is a really long
... Unicode string"""
```

Specific code points can be written using the `\u` escape sequence, which is followed by four hex digits giving the code point. If you use `\U` instead you specify 8 hex digits instead of 4. Unicode literals can also use the same escape sequences as 8-bit strings, including `\x`, but `\x` only takes two hex digits so it can't express all the available code points. You can add characters to Unicode strings using the `unichr()` built-in function and find out what the ordinal is with `ord()`.

Here is an example demonstrating the different alternatives:

```
>>> s = u"\x66\u0072\u0061\U0000006e" + unichr(231) + u"ais"
>>> # ^^^^ two-digit hex escape
>>> # ^^^^^^ four-digit Unicode escape
>>> # ^^^^^^^^^^ eight-digit Unicode escape
>>> for c in s: print ord(c),
...
97 102 114 97 110 231 97 105 115
>>> print s
français
```

Using escape sequences for code points greater than 127 is fine in small doses but Python 2.4 and above support writing Unicode literals in any encoding as long as you declare the encoding being used by including a special comment as either the first or second line of the source file:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
u = u'abcdé'
print ord(u[-1])
```

If you don't include such a comment, the default encoding used will be ASCII. Versions of Python before 2.4 were Euro-centric and assumed Latin-1 as a default encoding for string literals; in Python 2.4, characters greater than 127 still work but result in a warning. For example, the following program has no encoding declaration:

```
#!/usr/bin/env python
u = u'abcdé'
print ord(u[-1])
```

When you run it with Python 2.4, it will output the following warning:

```
sys:1: DeprecationWarning: Non-ASCII character '\xe9' in file testas.py on line 2, but
 no encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

and then the following output:

```
233
```

For real world use it is recommended that you use the UTF-8 encoding for your file but you must be sure that your text editor actually saves the file as UTF-8 otherwise the Python interpreter will try to parse UTF-8 characters but they will actually be stored as something else.

Note: Windows users who use the SciTE editor can specify the encoding of their file from the menu using the `File->Encoding`.

Note: If you are working with Unicode in detail you might also be interested in the `unicodedata` module which can be used to find out Unicode properties such as a character's name, category, numeric value and the like.

3.2.4 Input and Output

We now know how to use Unicode in Python source code but input and output can also be different using Unicode. Of course, some libraries natively support Unicode and if these libraries return Unicode objects you will not have to do anything special to support them. XML parsers and SQL databases frequently support Unicode for example.

If you remember from the discussion earlier, Unicode data consists of code points. In order to send Unicode data via a socket or write it to a file you usually need to encode it to a series of bytes and then decode the data back to Unicode when reading it. You can of course perform the encoding manually reading a byte at the time but since encodings such as UTF-8 can have variable numbers of bytes per character it is usually much easier to use Python's built-in support in the form of the `codecs` module.

The codecs module includes a version of the `open()` function that returns a file-like object that assumes the file's contents are in a specified encoding and accepts Unicode parameters for methods such as `.read()` and `.write()`.

The function's parameters are open(filename, mode='rb', encoding=None, errors='strict', buffering=1). `mode` can be 'r', 'w', or 'a', just like the corresponding parameter to the regular built-in `open()` function. You can add a `+` character to update the file. `buffering` is similar to the standard function's parameter. `encoding` is a string giving the encoding to use, if not specified or specified as `None`, a regular Python file object that accepts 8-bit strings is returned. Otherwise, a wrapper object is returned, and data written to or read from the wrapper object will be converted as needed. `errors` specifies the action for encoding errors and can be one of the usual values of `'strict'`, `'ignore'`, or `'replace'` which we saw right at the begining of this document when we were encoding strings in Python source files.

Here is an example of how to read Unicode from a UTF-8 encoded file:

```
import codecs
f = codecs.open('unicode.txt', encoding='utf-8')
for line in f:
    print repr(line)
```

It's also possible to open files in update mode, allowing both reading and writing:

```
f = codecs.open('unicode.txt', encoding='utf-8', mode='w+')
f.write(u"\x66\u0072\u0061\U0000006e" + unichr(231) + u"ais")
f.seek(0)
print repr(f.readline()[:1])
f.close()
```

Notice that we used the `repr()` function to display the Unicode data. This is very useful because if you tried to print the Unicode data directly, Python would need to encode it before it could be sent the console and depending on which characters were present and the character set used by the console, an error might be raised. This is avoided if you use `repr()`.

The Unicode character `U+FEFF` is used as a byte-order mark or BOM, and is often written as the first character of a file in order to assist with auto-detection of the file's byte ordering. Some encodings, such as UTF-16, expect a BOM to be present at the start of a file, but with others such as UTF-8 it isn't necessary.

When such an encoding is used, the BOM will be automatically written as the first character and will be silently dropped when the file is read. There are variants of these encodings, such as 'utf-16-le' and 'utf-16-be' for little-endian and big-endian encodings, that specify one particular byte ordering and don't skip the BOM.

Note: Some editors including SciTE will put a byte order mark (BOM) in the text file when saved as UTF-8, which is strange because UTF-8 doesn't need BOMs.

3.2.5 Unicode Filenames

Most modern operating systems support the use of Unicode filenames. The filenames are transparently converted to the underlying filesystem encoding. The type of encoding depends on the operating system.

On Windows 9x, the encoding is `mbcs`.

On Mac OS X, the encoding is `utf-8`.

On Unix, the encoding is the user's preference according to the result of nl_langinfo(CODESET), or None if the nl_langinfo(CODESET) failed.

On Windows NT+, file names are Unicode natively, so no conversion is performed. getfilesystemencoding still returns `mbcs`, as this is the encoding that applications should use when they explicitly want to convert Unicode strings to byte strings that are equivalent when used as file names.

`mbcs` is a special encoding for Windows that effectively means "use whichever encoding is appropriate". In Python 2.3 and above you can find out the system encoding with `sys.getfilesystemencoding()`.

Most file and directory functions and methods support Unicode. For example:

```
filename = u"\x66\u0072\u0061\U0000006e" + unichr(231) + u"ais"
f = open(filename, 'w')
f.write('Some data\n')
f.close()
```

Other functions such as `os.listdir()` will return Unicode if you pass a Unicode argument and will try to return strings if you pass an ordinary 8 bit string. For example running this example as `test.py`:

```
filename = u"Sample " + unichar(5000)
f = open(filename, 'w')
f.close()

import os
print os.listdir('.')
print os.listdir(u'.')
```

will produce the following output:

```
['Sample?', 'test.py']
[u'Sample\u1388', u'test.py']
```

Applying this to Web Programming

So far we've seen how to use encoding in source files and seen how to decode text to Unicode and encode it back to text. We've also seen that Unicode objects can be manipulated in similar ways to strings and we've seen how to perform input and output operations on files. Next we are going to look at how best to use Unicode in a web app.

The main rule is this:

Your application should use Unicode for all strings internally, decoding any input to Unicode as soon as it enters the application and encoding the Unicode to UTF-8 or another encoding only on output.

If you fail to do this you will find that `UnicodeDecodeError` s will start popping up in unexpected places when Unicode strings are used with normal 8-bit strings because Python's default encoding is ASCII and it will try to decode the text to ASCII and fail. It is always better to do any encoding or decoding at the edges of your application otherwise you will end up patching lots of different parts of your application unnecessarily as and when errors pop up.

Unless you have a very good reason not to it is wise to use UTF-8 as the default encoding since it is so widely supported.

The second rule is:

Always test your application with characters above 127 and above 255 wherever possible.

If you fail to do this you might think your application is working fine, but as soon as your users do put in non-ASCII characters you will have problems. Using arabic is always a good test and www.google.ae is a good source of sample text.

The third rule is:

Always do any checking of a string for illegal characters once it's in the form that will be used or stored, otherwise the illegal characters might be disguised.

For example, let's say you have a content management system that takes a Unicode filename, and you want to disallow paths with a '/' character. You might write this code:

```
def read_file(filename, encoding):
    if '/' in filename:
        raise ValueError("'/' not allowed in filenames")
    unicode_name = filename.decode(encoding)
    f = open(unicode_name, 'r')
    # ... return contents of file ...
```

This is INCORRECT. If an attacker could specify the 'base64' encoding, they could pass `L2V0Yy9wYXNzd2Q=` which is the base-64 encoded form of the string `'/etc/passwd'` which is a file you clearly don't want an attacker to get hold of. The above code looks for `/` characters in the encoded form and misses the dangerous character in the resulting decoded form.

Those are the three basic rules so now we will look at some of the places you might want to perform Unicode decoding in a Pylons application.

3.2.6 Request Parameters

Pylons automatically coerces incoming form parameters (`request.POST`, `GET` (quote GET) and `params`) into unicode objects (as of Pylons 0.9.6).

The request object contains a `charset` (encoding) attribute defining what the parameters should be decoded to (via value.decode(charset, errors)), and the decoding `errors` handler.

The unicode conversion of parameters can be disabled when `charset` is set to None.

```
def index(self):
    #request.charset = 'utf-8' # utf-8 is the default charset
    #request.errors = 'replace' # replace is the default error handler
    # a MultiDict-like object of string names and unicode values
    decoded_get = request.GET

    # The raw data is always still available when charset is None
    request.charset = None
    raw_get = request.GET
    raw_params = request.params
```

Pylons can also be configured to not coerece parameters to unicode objects by default. This is done by setting the following in the Pylons config object (at the bottom of your project's `config/environment.py`):

```
# Don't coerce parameters to unicode
config['pylons.request_options']['charset'] = None
# You can also change the default error handler
#config['pylons.request_options']['errors'] = 'strict'
```

When the `request` object is instructed to always automatically decode to unicode via the `request_settings` dictionary, the dictionary's `charset` value acts as a fallback charset. If a `charset` was sent by the browser (via the `Content-Type` header), the browser's value will take precedent: this takes place when the `request` object is constructed.

`FieldStorage` (file upload) objects will be handled specially for unicode parameters: what's provided is a copy of the original `FieldStorage` object with a unicode version of its `filename` attribute.

See *File Uploads* for more information on working with file uploads/`FieldStorage` objects.

Note: Only parameter values (not their associated names) are decoded to unicode by default. Since parameter names commonly map directly to Python variable names (which are restricted to the ASCII character set), it's usually preferable to handle them as strings. For example, passing form parameters to a function as keyword arguments (e.g. **request.params.mixed()) doesn't work with unicode keys.

To make `WSGIRequest` decode parameter names anyway, enable the `decode_param_names` option on either the WSGIRequest object or the `request_settings` dictionary. `FieldStorage's name` attributes are also decoded to unicode when this option is enabled.

3.2.7 Templating

Pylons uses Mako as its default templating language. Mako handles all content as unicode internally. It only deals in raw strings upon the final rendering of the template (the Mako `render()` function, used by the Pylons `render()` function/Buffet plugin). The encoding of the rendered string can be configured; Pylons sets the default value to UTF-8. To change this value, edit your project's `config/environment.py` file and add the following option:

```
# Customize templating options via this variable
tmpl_options = config['buffet.template_options']

tmpl_options['mako.output_encoding'] = 'utf-8'
```

replacing `utf-8` with the encoding you wish to use.

More information can be found at Mako's Unicode Chapter.

3.2.8 Output Encoding

Web pages should be generated with a specific encoding, most likely UTF-8. At the very least, that means you should specify the following in the `<head>` section:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

The charset should also be specified in the `Content-Type` header (which Pylons automatically does for you):

```
response.headers['Content-type'] = 'text/html; charset=utf-8'
```

Pylons has a notion of `response_options`, complimenting the `request_options` mentioned in the Request Parameters section above. The default request charset can be changed by setting the following in the Pylons config object (at the bottom of your project's `config/environment.py`):

```
config['pylons.response_options']['charset'] = 'utf-8'
```

replacing `utf-8` with the charset you wish to use.

If you specify that your output is UTF-8, generally the web browser will give you UTF-8. If you want the browser to submit data using a different character set, you can set the encoding by adding the `accept-encoding` tag to your form. Here is an example:

```
<form accept-encoding="US-ASCII" ...>
```

However, be forewarned that if the user tries to give you non-ASCII text, then:

- Firefox will translate the non-ASCII text into HTML entities.
- IE will ignore your suggested encoding and give you UTF-8 anyway.

The lesson to be learned is that if you output UTF-8, you had better be prepared to accept UTF-8 by decoding the data in `request.params` as described in the section above entitled Request Parameters'.

Another technique which is sometimes used to determine the character set is to use an algorithm to analyse the input and guess the encoding based on probabilities.

For instance, if you get a file, and you don't know what encoding it is encoded in, you can often rename the file with a .txt extension and then try to open it in Firefox. Then you can use the "View->Character Encoding" menu to try to auto-detect the encoding.

3.2.9 Databases

Your database driver should automatically convert from Unicode objects to a particular charset when writing and back again when reading. Again it is normal to use UTF-8 which is well supported.

You should check your database's documentation for information on how it handles Unicode.

For example MySQL's Unicode documentation is here http://dev.mysql.com/doc/refman/5.0/en/charset-unicode.html

Also note that you need to consider both the encoding of the database and the encoding used by the database driver.

If you're using MySQL together with SQLAlchemy, see the following, as there are some bugs in MySQLdb that you'll need to work around:

http://www.mail-archive.com/sqlalchemy@googlegroups.com/msg00366.html

Summary

Hopefully you now understand the history of Unicode, how to use it in Python and where to apply Unicode encoding and decoding in a Pylons application. You should also be able to use Unicode in your web app remembering the basic rule to use UTF-8 to talk to the world, do the encode and decode at the edge of your application.

Further Reading

This information is based partly on the following articles which can be consulted for further information.:

http://www.joelonsoftware.com/articles/Unicode.html

http://www.amk.ca/python/howto/unicode

Please feel free to report any mistakes to the Pylons mailing list or to the author. Any corrections or clarifications would be gratefully received.

CHAPTER

FOUR

CONTROLLERS

THE FIRST PYLON OF THE TEMPLE OF MEDINET HABU

In the *MVC* paradigm the *controller* interprets the inputs, commanding the model and/or the view to change as appropriate. Under Pylons, this concept is extended slightly in that a Pylons controller is not directly interpreting the client's request, but is acting to determine the appropriate way to assemble data from the model, and render it with the correct template.

The controller interprets requests from the user and calls portions of the model and view as necessary to fulfill the request. So when the user clicks a Web link or submits an HTML form, the controller itself doesn't output anything or perform any real processing. It takes the request and determines which model components to invoke and which formatting to apply to the resulting data.

Pylons uses a class, where the superclass provides the *WSGI* interface and the subclass implements the application-specific controller logic.

The Pylons WSGI Controller handles incoming web requests that are dispatched from the Pylons WSGI application `PylonsApp`.

These requests result in a new instance of the `WSGIController` being created, which is then called with the dict options from the Routes match. The standard WSGI response is then returned with start_response called as per the WSGI spec.

Since Pylons controllers are actually called with the WSGI interface, normal WSGI applications can also be Pylons 'controllers'.

4.1 Standard Controllers

Standard Controllers intended for subclassing by web developers

4.1.1 Keeping methods private

The default route maps any controller and action, so you will likely want to prevent some controller methods from being callable from a URL.

Pylons uses the default Python convention of private methods beginning with _. To hide a method `edit_generic` in this class, just changing its name to begin with _ will be sufficient:

```
class UserController(BaseController):
    def index(self):
        return "This is the index."

    def _edit_generic(self):
        """I can't be called from the web!"""
        return True
```

4.1.2 Special methods

Special controller methods you may define:

__before__ This method is called before your action is, and should be used for setting up variables/objects, restricting access to other actions, or other tasks which should be executed before the action is called.

__after__ This method is called after the action is, unless an unexpected exception was raised. Subclasses of `HTTPException` (such as those raised by `redirect_to` and `abort`) are expected; e.g. `__after__` will be called on redirects.

4.1.3 Adding Controllers dynamically

It is possible for an application to add controllers without restarting the application. This requires telling Routes to re-scan the controllers directory.

New controllers may be added from the command line with the paster command (recommended as that also creates the test harness file), or any other means of creating the controller file.

For Routes to become aware of new controllers present in the controller directory, an internal flag is toggled to indicate that Routes should rescan the directory:

```
from routes import request_config

mapper = request_config().mapper
mapper._created_regs = False
```

On the next request, Routes will rescan the controllers directory and those routes that use the `:controller` dynamic part of the path will be able to match the new controller.

4.1.4 Attaching WSGI apps

Note: This recipe assumes a basic level of familiarity with the WSGI Specification (PEP 333)

WSGI runs deep through Pylons, and is present in many parts of the architecture. Since Pylons controllers are actually called with the WSGI interface, normal WSGI applications can also be Pylons 'controllers'.

Optionally, if a full WSGI app should be mounted and handle the remainder of the URL, Routes can automatically move the right part of the URL into the `SCRIPT_NAME`, so that the WSGI application can properly handle its `PATH_INFO` part.

This recipe will demonstrate adding a basic WSGI app as a Pylons controller.

Create a new controller file in your Pylons project directory:

```
$ paster controller wsgiapp
```

This sets up the basic imports that you may want available when using other WSGI applications.

Edit your controller so it looks like this:

```
import logging

from YOURPROJ.lib.base import *

log = logging.getLogger(__name__)

def WsgiappController(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/plain')])
    return ["Hello World"]
```

When hooking up other WSGI applications, they will expect the part of the URL that was used to get to this controller to have been moved into `SCRIPT_NAME`. `Routes` can properly adjust the environ if a map route for this controller is added to the `config/routing.py` file:

```
# CUSTOM ROUTES HERE

# Map the WSGI application
map.connect('wsgiapp/{path_info:.*}', controller='wsgiapp')
```

By specifying the `path_info` dynamic path, Routes will put everything leading up to the `path_info` in the `SCRIPT_NAME` and the rest will go in the `PATH_INFO`.

4.2 Using the WSGI Controller to provide a WSGI service

4.2.1 The Pylons WSGI Controller

Pylons' own WSGI Controller follows the WSGI spec for calling and return values

The Pylons WSGI Controller handles incoming web requests that are dispatched from `PylonsApp`. These requests result in a new instance of the `WSGIController` being created, which is then called with the dict options from the Routes match. The standard WSGI response is then returned with `start_response()` called as per the WSGI spec.

4.2.2 WSGIController methods

Special `WSGIController` methods you may define:

`__before__` This method will be run before your action is, and should be used for setting up variables/objects, restricting access to other actions, or other tasks which should be executed before the action is called.

`__after__` Method to run after the action is run. This method will *always* be run after your method, even if it raises an Exception or redirects.

Each action to be called is inspected with `_inspect_call()` so that it is only passed the arguments in the Routes match dict that it asks for. The arguments passed into the action can be customized by overriding the `_get_method_args()` function which is expected to return a dict.

In the event that an action is not found to handle the request, the Controller will raise an "Action Not Found" error if in debug mode, otherwise a `404 Not Found` error will be returned.

4.3 Using the REST Controller with a RESTful API

4.3.1 Using the paster restcontroller template

```
$ paster restcontroller --help
```

Create a REST Controller and accompanying functional test

The RestController command will create a REST-based Controller file for use with the `resource()` REST-based dispatching. This template includes the methods that `resource()` dispatches to in addition to doc strings for clarification on when the methods will be called.

The first argument should be the singular form of the REST resource. The second argument is the plural form of the word. If its a nested controller, put the directory information in front as shown in the second example below.

Example usage:

```
$ paster restcontroller comment comments
Creating yourproj/yourproj/controllers/comments.py
Creating yourproj/yourproj/tests/functional/test_comments.py
```

If you'd like to have controllers underneath a directory, just include the path as the controller name and the necessary directories will be created for you:

```
$ paster restcontroller admin/trackback admin/trackbacks
Creating yourproj/controllers/admin
Creating yourproj/yourproj/controllers/admin/trackbacks.py
Creating yourproj/yourproj/tests/functional/test_admin_trackbacks.py
```

4.3.2 An Atom-Style REST Controller for Users

```
# From http://pylonshq.com/pasties/503
import logging

from formencode.api import Invalid
from pylons import url
from simplejson import dumps

from restmarks.lib.base import *

log = logging.getLogger(__name__)

class UsersController(BaseController):
    """REST Controller styled on the Atom Publishing Protocol"""
    # To properly map this controller, ensure your
    # config/routing.py file has a resource setup:
    #     map.resource('user', 'users')

    def index(self, format='html'):
        """GET /users: All items in the collection.<br>
            @param format the format passed from the URI.
        """
        #url('users')
        users = model.User.select()
        if format == 'json':
            data = []
            for user in users:
                d = user._state['original'].data
                del d['password']
                d['link'] = url('user', id=user.name)
                data.append(d)
            response.headers['content-type'] = 'text/javascript'
            return dumps(data)
        else:
            c.users = users
            return render('/users/index_user.mako')

    def create(self):
        """POST /users: Create a new item."""
        # url('users')
        user = model.User.get_by(name=request.params['name'])
        if user:
            # The client tried to create a user that already exists
            abort(409, '409 Conflict',
                  headers=[('location', url('user', id=user.name))])
        else:
            try:
                # Validate the data that was sent to us
```

```
                params = model.forms.UserForm.to_python(request.params)
            except Invalid, e:
                # Something didn't validate correctly
                abort(400, '400 Bad Request -- %s' % e)
            user = model.User(**params)
            model.objectstore.flush()
            response.headers['location'] = url('user', id=user.name)
            response.status_code = 201
            c.user_name = user.name
            return render('/users/created_user.mako')

    def new(self, format='html'):
        """GET /users/new: Form to create a new item.
            @param format the format passed from the URI.
        """
        # url('new_user')
        return render('/users/new_user.mako')

    def update(self, id):
        """PUT /users/id: Update an existing item.
            @param id the id (name) of the user to be updated
        """
        # Forms posted to this method should contain a hidden field:
        #    <input type="hidden" name="_method" value="PUT" />
        # Or using helpers:
        #    h.form(url('user', id=ID),
        #           method='put')
        # url('user', id=ID)
        old_name = id
        new_name = request.params['name']
        user = model.User.get_by(name=id)

        if user:
            if (old_name != new_name) and model.User.get_by(name=new_name):
                abort(409, '409 Conflict')
            else:
                params = model.forms.UserForm.to_python(request.params)
                user.name = params['name']
                user.full_name = params['full_name']
                user.email = params['email']
                user.password = params['password']
                model.objectstore.flush()
                if user.name != old_name:
                    abort(301, '301 Moved Permanently',
                          [('Location', url('users', id=user.name))])
                else:
                    return

    def delete(self, id):
        """DELETE /users/id: Delete an existing item.
            @param id the id (name) of the user to be updated
        """
        # Forms posted to this method should contain a hidden field:
        #    <input type="hidden" name="_method" value="DELETE" />
        # Or using helpers:
        #    h.form(url('user', id=ID),
        #           method='delete')
        # url('user', id=ID)
```

```
        user = model.User.get_by(name=id)
        user.delete()
        model.objectstore.flush()
        return

    def show(self, id, format='html'):
        """GET /users/id: Show a specific item.
            @param id the id (name) of the user to be updated.
            @param format the format of the URI requested.
        """
        # url('user', id=ID)
        user = model.User.get_by(name=id)
        if user:
            if format=='json':
                data = user._state['original'].data
                del data['password']
                data['link'] = url('user', id=user.name)
                response.headers['content-type'] = 'text/javascript'
                return dumps(data)
            else:
                c.data = user
                return render('/users/show_user.mako')
        else:
            abort(404, '404 Not Found')

    def edit(self, id, format='html'):
        """GET /users/id;edit: Form to edit an existing item.
            @param id the id (name) of the user to be updated.
            @param format the format of the URI requested.
        """
        # url('edit_user', id=ID)
        user = model.User.get_by(name=id)
        if not user:
            abort(404, '404 Not Found')
        # Get the form values from the table
        c.values = model.forms.UserForm.from_python(user.__dict__)
        return render('/users/edit_user.mako')
```

4.4 Using the XML-RPC Controller for XML-RPC requests

In order to deploy this controller you will need at least a passing familiarity with XML-RPC itself. We will first review the basics of XML-RPC and then describe the workings of the `Pylons XMLRPCController`. Finally, we will show an example of how to use the controller to implement a simple web service.

After you've read this document, you may be interested in reading the companion document: "A blog publishing web service in XML-RPC" which takes the subject further, covering details of the MetaWeblog API (a popular XML-RPC service) and demonstrating how to construct some basic service methods to act as the core of a MetaWeblog blog publishing service.

4.4.1 A brief introduction to XML-RPC

XML-RPC is a specification that describes a Remote Procedure Call (RPC) interface by which an application can use the Internet to execute a specified procedure call on a remote XML-RPC server. The name of the procedure to be called and any required parameter values are "marshalled" into XML. The XML forms the

body of a POST request which is despatched via HTTP to the XML-RPC server. At the server, the procedure is executed, the returned value(s) is/are marshalled into XML and despatched back to the application. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

4.4.2 XML-RPC Controller that speaks WSGI

Pylons uses Python's xmlrpclib library to provide a specialised `XMLRPCController` class that gives you the full range of these XML-RPC Introspection facilities for use in your service methods and provides the foundation for constructing a set of specialised service methods that provide a useful web service — such as a blog publishing interface.

This controller handles XML-RPC responses and complies with the XML-RPC Specification as well as the XML-RPC Introspection specification.

As part of its basic functionality an XML-RPC server provides three standard introspection procedures or "service methods" as they are called. The Pylons `XMLRPCController` class provides these standard service methods ready-made for you:

- `system.listMethods()` Returns a list of XML-RPC methods for this XML-RPC resource
- `system.methodSignature()` Returns an array of arrays for the valid signatures for a method. The first value of each array is the return value of the method. The result is an array to indicate multiple signatures a method may be capable of.
- `system.methodHelp()` Returns the documentation for a method

By default, methods with names containing a dot are translated to use an underscore. For example, the `system.methodHelp` is handled by the method `system_methodHelp()`.

Methods in the XML-RPC controller will be called with the method given in the XML-RPC body. Methods may be annotated with a signature attribute to declare the valid arguments and return types.

For example:

```
class MyXML(XMLRPCController):
    def userstatus(self):
        return 'basic string'
    userstatus.signature = [['string']]

    def userinfo(self, username, age=None):
        user = LookUpUser(username)
        result = {'username': user.name}
        if age and age > 10:
            result['age'] = age
        return result
    userinfo.signature = [['struct', 'string'],
                          ['struct', 'string', 'int']]
```

Since XML-RPC methods can take different sets of data, each set of valid arguments is its own list. The first value in the list is the type of the return argument. The rest of the arguments are the types of the data that must be passed in.

In the last method in the example above, since the method can optionally take an integer value, both sets of valid parameter lists should be provided.

Valid types that can be checked in the signature and their corresponding Python types:

XMLRPC	Python
string	str
array	list
boolean	bool
int	int
double	float
struct	dict
dateTime.iso8601	xmlrpclib.DateTime
base64	xmlrpclib.Binary

Note, requiring a signature is optional.

Also note that a convenient fault handler function is provided.

```
def xmlrpc_fault(code, message):
    """Convenience method to return a Pylons response XMLRPC Fault"""
```

(The XML-RPC Home page and the XML-RPC HOW-TO both provide further detail on the XML-RPC specification.)

4.4.3 A simple XML-RPC service

This simple service `test.battingOrder` accepts a positive integer < 51 as the parameter `posn` and returns a string containing the name of the US state occupying that ranking in the order of ratifying the constitution / joining the union.

```
import xmlrpclib

from pylons import request
from pylons.controllers import XMLRPCController

states = ['Delaware', 'Pennsylvania', 'New Jersey', 'Georgia',
          'Connecticut', 'Massachusetts', 'Maryland', 'South Carolina',
          'New Hampshire', 'Virginia', 'New York', 'North Carolina',
          'Rhode Island', 'Vermont', 'Kentucky', 'Tennessee', 'Ohio',
          'Louisiana', 'Indiana', 'Mississippi', 'Illinois', 'Alabama',
          'Maine', 'Missouri', 'Arkansas', 'Michigan', 'Florida', 'Texas',
          'Iowa', 'Wisconsin', 'California', 'Minnesota', 'Oregon',
          'Kansas', 'West Virginia', 'Nevada', 'Nebraska', 'Colorado',
          'North Dakota', 'South Dakota', 'Montana', 'Washington', 'Idaho',
          'Wyoming', 'Utah', 'Oklahoma', 'New Mexico', 'Arizona', 'Alaska',
          'Hawaii']

class RpctestController(XMLRPCController):

    def test_battingOrder(self, posn):
        """This docstring becomes the content of the
        returned value for system.methodHelp called with
        the parameter "test.battingOrder"). The method
        signature will be appended below ...
        """
        # XML-RPC checks agreement for arity and parameter datatype, so
        # by the time we get called, we know we have an int.
        if posn > 0 and posn < 51:
            return states[posn-1]
        else:
            # Technically, the param value is correct: it is an int.
```

```
            # Raising an error is inappropriate, so instead we
            # return a facetious message as a string.
            return 'Out of cheese error.'
    test_battingOrder.signature = [['string', 'int']]
```

4.4.4 Testing the service

For developers using OS X, there's an XML/RPC client that is an extremely useful diagnostic tool when developing XML-RPC (it's free ... but not entirely bug-free). Or, you can just use the Python interpreter:

```
>>> from pprint import pprint
>>> import xmlrpclib
>>> srvr = xmlrpclib.Server("http://example.com/rpctest/")
>>> pprint(srvr.system.listMethods())
['system.listMethods',
 'system.methodHelp',
 'system.methodSignature',
 'test.battingOrder']
>>> print srvr.system.methodHelp('test.battingOrder')
This docstring becomes the content of the
returned value for system.methodHelp called with
the parameter "test.battingOrder"). The method
signature will be appended below ...

Method signature: [['string', 'int']]
>>> pprint(srvr.system.methodSignature('test.battingOrder'))
[['string', 'int']]
>>> pprint(srvr.test.battingOrder(12))
'North Carolina'
```

To debug XML-RPC servers from Python, create the client object using the optional verbose=1 parameter. You can then use the client as normal and watch as the XML-RPC request and response is displayed in the console.

CHAPTER

FIVE

VIEWS

THE FIRST PYLON OF HARMHABÎ AT KARNAK.[1]

In the MVC paradigm the *view* manages the presentation of the model.

The view is the interface the user sees and interacts with. For Web applications, this has historically been an HTML interface. HTML remains the dominant interface for Web apps but new view options are rapidly appearing.

These include Macromedia Flash, JSON and views expressed in alternate markup languages like XHTML, XML/XSL, WML, and Web services. It is becoming increasingly common for web apps to provide specialised views in the form of a REST API that allows programmatic read/write access to the data model.

More complex APIs are quite readily implemented via SOAP services, yet another type of view on to the data model.

The growing adoption of RDF, the graph-based representation scheme that underpins the Semantic Web, brings a perspective that is strongly weighted towards machine-readability.

Handling all of these interfaces in an application is becoming increasingly challenging. One big advantage of MVC is that it makes it easier to create these interfaces and develop a web app that supports many different views and thereby provides a broad range of services.

Typically, no significant processing occurs in the view; it serves only as a means of outputting data and allowing the user (or the application) to act on that data, irrespective of whether it is an online store or an employee list.

5.1 Templates

Template rendering engines are a popular choice for handling the task of view presentation.

To return a processed template, it must be rendered and returned by the controller:

```
from helloworld.lib.base import BaseController, render

class HelloController(BaseController):
    def sample(self):
        return render('/sample.mako')
```

Using the default Mako template engine, this will cause Mako to look in the `helloworld/templates` directory (assuming the project is called 'helloworld') for a template filed called `sample.mako`.

The `render()` function used here is actually an alias defined in your projects' `base.py` for Pylons' `render_mako()` function.

5.1.1 Directly-supported template engines

Pylons provides pre-configured options for using the Mako, Genshi and Jinja2 template rendering engines. They are setup automatically during the creation of a new Pylons project, or can be added later manually.

5.2 Passing Variables to Templates

To pass objects to templates, the standard Pylons method is to attach them to the *tmpl_context* (aliased as *c* in controllers and templates, by default) object in the *Controllers*:

```
import logging

from pylons import request, response, session, tmpl_context as c
```

```
from pylons.controllers.util import abort, redirect_to

from helloworld.lib.base import BaseController, render

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        c.name = "Fred Smith"
        return render('/sample.mako')
```

Using the variable in the template:

```
Hi there ${c.name}!
```

5.2.1 Strict vs Attribute-Safe tmpl_context objects

The *tmpl_context* object is created at the beginning of every request, and by default is an instance of the `AttribSafeContextObj` class, which is an Attribute-Safe object. This means that accessing attributes on it that do **not** exist will return an empty string **instead** of raising an `AttributeError` error.

This can be convenient for use in templates since it can act as a default:

```
Hi there ${c.name}
```

That will work when *c.name* has not been set, and is a bit shorter than what would be needed with the strict `ContextObj` context object.

Switching to the strict version of the *tmpl_context* object can be done in the `config/environment.py` by adding (after the config.init_app):

```
config['pylons.strict_c'] = True
```

5.3 Default Template Variables

By default, all templates have a set of variables present in them to make it easier to get to common objects. The full list of available names present in the templates global scope:

- *c* – Template context object (Alias for *tmpl_context*)
- *tmpl_context* – Template context object
- `config` – Pylons `PylonsConfig` object (acts as a dict)
- *g* – Project application globals object (Alias for *app_globals*)
- *app_globals* – Project application globals object
- *h* – Project helpers module reference
- `request` – Pylons `Request` object for this request
- `response` – Pylons `Response` object for this request
- `session` – Pylons session object (unless Sessions are removed)
- `translator` – Gettext translator object configured for current locale

- `ungettext()` – Unicode capable version of gettext's ngettext function (handles plural translations)
- `_()` – Unicode capable gettext translate function
- `N_()` – gettext no-op function to mark a string for translation, but doesn't actually translate

5.4 Configuring Template Engines

A new Pylons project comes with the template engine setup inside the projects' `config/environment.py` file. This section creates the Mako template lookup object and attaches it to the *app_globals* object, for use by the template rendering function.

```
# these imports are at the top
from mako.lookup import TemplateLookup
from pylons.error import handle_mako_error

# this section is inside the load_environment function
# Create the Mako TemplateLookup, with the default auto-escaping
config['pylons.app_globals'].mako_lookup = TemplateLookup(
    directories=paths['templates'],
    error_handler=handle_mako_error,
    module_directory=os.path.join(app_conf['cache_dir'], 'templates'),
    input_encoding='utf-8', default_filters=['escape'],
    imports=['from webhelpers.html import escape'])
```

5.4.1 Using Multiple Template Engines

Since template engines are configured in the `config/environment.py` section, then used by render functions, it's trivial to setup additional template engines, or even differently configured versions of a single template engine. However, custom render functions will frequently be needed to utilize the additional template engine objects.

Example of additional Mako template loader for a different templates directory for admins, which falls back to the normal templates directory:

```
# Add the additional path for the admin template
paths = dict(root=root,
             controllers=os.path.join(root, 'controllers'),
             static_files=os.path.join(root, 'public'),
             templates=[os.path.join(root, 'templates')],
             admintemplates=[os.path.join(root, 'admintemplates'),
                             os.path.join(root, 'templates')])

config['pylons.app_globals'].mako_admin_lookup = TemplateLookup(
    directories=paths['admin_templates'],
    error_handler=handle_mako_error,
    module_directory=os.path.join(app_conf['cache_dir'], 'admintemplates'),
    input_encoding='utf-8', default_filters=['escape'],
    imports=['from webhelpers.html import escape'])
```

That adds the additional template lookup instance, next a *custom render function* is needed that utilizes it:

```
from pylons.templating import cached_template, pylons_globals

def render_mako_admin(template_name, extra_vars=None, cache_key=None,
                      cache_type=None, cache_expire=None):
```

```
    # Create a render callable for the cache function
    def render_template():
        # Pull in extra vars if needed
        globs = extra_vars or {}

        # Second, get the globals
        globs.update(pylons_globals())

        # Grab a template reference
        template = globs['app_globals'].mako_admin_lookup.get_template(template_name)

        return template.render(**globs)

    return cached_template(template_name, render_template, cache_key=cache_key,
                           cache_type=cache_type, cache_expire=cache_expire)
```

The only change from the `render_mako()` function that comes with Pylons is to use the *mako_admin_lookup* rather than the *mako_lookup* that is used by default.

5.5 Custom `render()` functions

Writing custom render functions can be used to access specific features in a template engine, such as Genshi, that go beyond the default `render_genshi()` functionality or to add support for additional template engines.

Two helper functions for use with the render function are provided to make it easier to include the common Pylons globals that are useful in a template in addition to enabling easy use of cache capabilities. The `pylons_globals()` and `cached_template()` functions can be used if desired.

Generally, the custom render function should reside in the project's `lib/` directory, probably in `base.py`.

Here's a sample Genshi render function as it would look in a project's `lib/base.py` that doesn't fully render the result to a string, and rather than use `c` assumes that a dict is passed in to be used in the templates global namespace. It also returns a Genshi stream instead the rendered string.

```
from pylons.templating import pylons_globals

def render(template_name, tmpl_vars):
    # First, get the globals
    globs = pylons_globals()

    # Update the passed in vars with the globals
    tmpl_vars.update(globs)

    # Grab a template reference
    template = globs['app_globals'].genshi_loader.load(template_name)

    # Render the template
    return template.generate(**tmpl_vars)
```

Using the `pylons_globals()` function also makes it easy to get to the *app_globals* object which is where the template engine was attached in `config/environment.py`. Changed in version 0.9.7: Prior to 0.9.7, all templating was handled through a layer called 'Buffet'. This layer frequently made customization of the template engine difficult as any customization required additional plugin modules being installed. Pylons 0.9.7 now deprecates use of the Buffet plug-in layer.

See Also:

`pylons.templating` - Pylons templating API

5.6 Templating with Mako

5.6.1 Introduction

The template library deals with the *view*, presenting the model. It generates (X)HTML code, CSS and Javascript that is sent to the browser. *(In the examples for this section, the project root is ''myapp''.)*

Static vs. dynamic

Templates to generate dynamic web content are stored in *myapp/templates*, static files are stored in *myapp/public*.

Both are served from the server root, **if there is a name conflict the static files will be served in preference**

5.6.2 Making a template hierarchy

Create a base template

In *myapp/templates* create a file named *base.mako* and edit it to appear as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    ${self.head_tags()}
  </head>
  <body>
    ${self.body()}
  </body>
</html>
```

A base template such as the very basic one above can be used for all pages rendered by Mako. This is useful for giving a consistent look to the application.

- Expressions wrapped in *${...}* are evaluated by Mako and returned as text
- *${* and *}* may span several lines but the closing brace should not be on a line by itself (or Mako throws an error)
- Functions that are part of the *self* namespace are defined in the Mako templates

Create child templates

Create another file in *myapp/templates* called *my_action.mako* and edit it to appear as follows:

```
<%inherit file="/base.mako" />

<%def name="head_tags()">
  <!-- add some head tags here -->
</%def>

<h1>My Controller</h1>
```

```
<p>Lorem ipsum dolor ...</p>
```

This file define the functions called by *base.mako*.

- The *inherit* tag specifies a parent file to pass program flow to
- Mako defines functions with *<%def name="function_name()">...</%def>*, the contents of the tag are returned
- Anything left after the Mako tags are parsed out is automatically put into the *body()* function

A consistent feel to an application can be more readily achieved if all application pages refer back to single file (in this case *base.mako*)..

Check that it works

In the controller action, use the following as a *return()* value,

```
return render('/my_action.mako')
```

Now run the action, usually by visiting something like `http://localhost:5000/my_controller/my_action` in a browser. Selecting 'View Source' in the browser should reveal the following output:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
  <!-- add some head tags here -->
  </head>
  <body>

<h1>My Controller</h1>

<p>Lorem ipsum dolor ...</p>

  </body>
</html>
```

See Also:

The Mako documentation Reasonably straightforward to follow

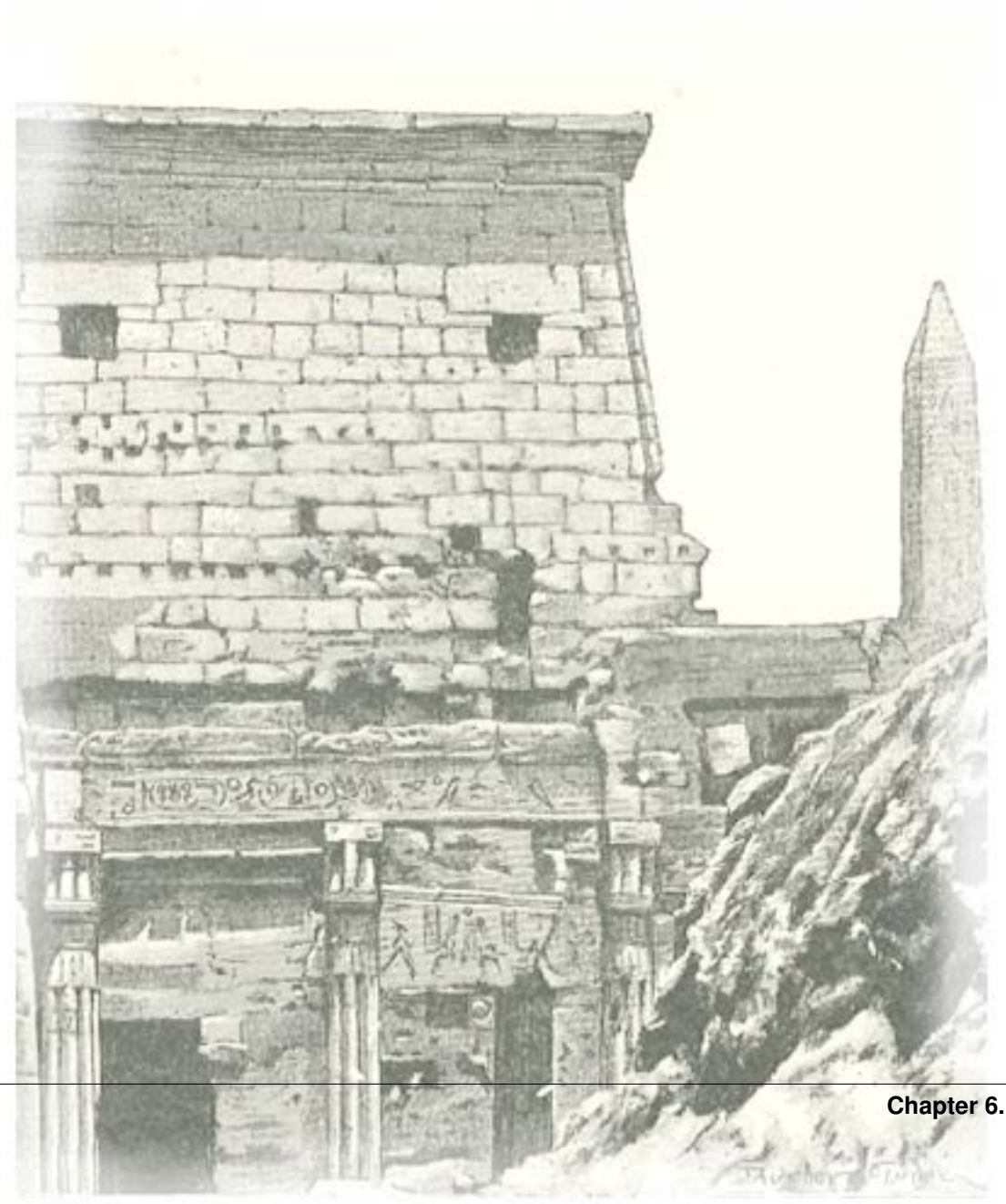
See the *Internationalization and Localization* Provides more help on making your application more worldly.

CHAPTER

SIX

MODELS

6.1 About the model



In the MVC paradigm the *model* manages the behavior and data of the application domain, responds to requests for information about its state and responds to instructions to change state.

The model represents enterprise data and business rules. It is where most of the processing takes place when using the MVC design pattern. Databases are in the remit of the model, as are component objects such as *EJBs* and *ColdFusion Components*.

The data returned by the model is display-neutral, i.e. the model applies no formatting. A single model can provide data for any number of display interfaces. This reduces code duplication as model code is written only once and is then reused by all of the views.

Because the model returns data without applying any formatting, the same components can be used with any interface. For example, most data is typically formatted with HTML but it could also be formatted with Macromedia Flash or WAP.

The model also isolates and handles state management and data persistence. For example, a Flash site or a wireless application can both rely on the same session-based shopping cart and e-commerce processes.

Because the model is self-contained and separate from the controller and the view, changing the data layer or business rules is less painful. If it proves necessary to switch databases, e.g. from MySQL to Oracle, or change a data source from an RDBMS to LDAP, the only required task is that of altering the model. If the view is written correctly, it won't care at all whether a list of users came from a database or an LDAP server.

This freedom arises from the way that the three parts of an MVC-based application act as *black boxes*, the inner workings of each one are hidden from, and are independent of, the other two. The approach promotes well-defined interfaces and self-contained components.

Note: *adapted from an Oct 2002 TechRepublic article by by Brian Kotek: "MVC design pattern brings about better organization and code reuse"* - http://articles.techrepublic.com.com/5100-10878_11-1049862.html

---

6.2 Model basics

Pylons provides a `model` package to put your database code in but does not offer a database engine or API. Instead there are several third-party APIs to choose from.

6.2.1 SQL databases

SQLAlchemy

SQLAlchemy is by far the most common approach for Pylons databases. It provides a connection pool, a SQL statement builder, an object-relational mapper (ORM), and transaction support. SQLAlchemy works with several database engines (MySQL, PostgreSQL, SQLite, Oracle, Firebird, MS-SQL, Access via ODBC, etc) and understands the peculiar SQL dialect of each, making it possible to port a program from one engine to another by simply changing the connection string. Although its API is still changing gradually, SQLAlchemy is well tested, widely deployed, has excellent documentation, and its mailing list is quick with answers. *Using SQLAlchemy* describes the recommended way to configure a Pylons application for SQLAlchemy.

SQLAlchemy lets you work at three different levels, and you can even use multiple levels in the same program:

- The object-relational mapper (ORM) lets you interact with the database using your own object classes rather than writing SQL code.

- The SQL expression language has many methods to create customized SQL statements, and the result cursor is more friendly than DBAPI's.
- The low-level execute methods accept literal SQL strings if you find something the SQL builder can't do, such as adding a column to an existing table or modifying the column's type. If they return results, you still get the benefit of SQLAlchemy's result cursor.

The first two levels are *database neutral*, meaning they hide the differences between the databases' SQL dialects. Changing to a different database is merely a matter of supplying a new connection URL. Of course there are limits to this, but SQLAlchemy is 90% easier than rewriting all your SQL queries.

The SQLAlchemy manual should be your next stop for questions not covered here. It's very well written and thorough.

SQLAlchemy add-ons

Most of these provide a higher-level ORM, either by combining the table definition and ORM class definition into one step, or supporting an "active record" style of access. *Please take the time to learn how to do things "the regular way" before using these shortcuts in a production application*. Understanding what these add-ons do behind the scenes will help if you have to troubleshoot a database error or work around a limitation in the add-on later.

SQLSoup, an extension to SQLAlchemy, provides a quick way to generate ORM classes based on existing database tables.

If you're familiar with ActiveRecord, used in Ruby on Rails, then you may want to use the Elixir layer on top of SQLAlchemy.

Tesla is a framework built on top of Pylons and Elixir/SQLAlchemy.

Non-SQLAlchemy libraries

Most of these expose only the object-relational mapper; their SQL builder and connection pool are not meant to be used directly.

Storm

Geniusql

DB-API

All the SQL libraries above are built on top of Python's DB-API, which provides a common low-level interface for interacting with several database engines: MySQL, PostgreSQL, SQLite, Oracle, Firebird, MS-SQL, Access via ODBC, etc. Most programmers do not use DB-API directly because its API is low-level and repetitive and does not provide a connection pool. There's no "DB-API package" to install because it's an abstract interface rather than software. Instead, install the Python package for the particular engine you're interested in. Python's Database Topic Guide describes the DB-API and lists the package required for each engine. The sqlite3 package for SQLite is included in Python 2.5.

6.2.2 Object databases

Object databases store Python dicts, lists, and classes in pickles, allowing you to access hierarchical data using normal Python statements rather than having to map them to tables, relations, and a foreign language (SQL).

ZODB

Durus [1]

6.2.3 Other databases

Pylons can also work with other database systems, such as the following:

Schevo uses Durus to combine some features of relational and object databases. It is written in Python.

CouchDb is a document-based database. It features a Python API.

The Datastore database in Google App Engine.

6.3 Working with SQLAlchemy

6.3.1 Install SQLAlchemy

We'll assume you've already installed Pylons and have the *easy_install* command. At the command line, run:

```
easy_install SQLAlchemy
```

Next you'll have to install a database engine and its Python bindings. If you don't know which one to choose, SQLite is a good one to start with. It's small and easy to install, and Python 2.5 includes bindings for it. Installing the database engine is beyond the scope of this article, but here are the Python bindings you'll need for the most popular engines:

```
easy_install pysqlite # If you use SQLite and Python 2.4 (not needed for Python 2.5)
easy_install MySQL-python # If you use MySQL
easy_install psycopg2 # If you use PostgreSQL
```

See the Python Package Index (formerly the Cheeseshop) for other database drivers.

Check Your Version

To see which version of SQLAlchemy you have, go to a Python shell and look at sqlalchemy.__version__ :

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
0.5.0
```

6.3.2 Defining tables and ORM classes

When you answer "yes" to the SQLAlchemy question when creating a Pylons project, it configures a simple default model. The model consists of two files: *__init__.py* and *meta.py*. *__init__.py* contains your table definitions and ORM classes, and an `init_model()` function which must be called at application startup. *meta.py* is merely a container for SQLAlchemy's housekeeping objects (`Session`, `metadata`, and `engine`), which not all applications will use. If your application is small, you can put your table definitions in *__init__.py* for simplicity. If your application has many tables or multiple databases, you may prefer to split them up into multiple modules within the model.

[1] Durus is not thread safe, so you should use its server mode if your application writes to the database. Do not share connections between threads. ZODB is thread safe, so it may be a more convenient alternative.

Here's a sample *model/__init__.py* with a "persons" table, which is based on the default model with the comments removed:

```
"""The application's model objects"""
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta

def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    ## Reflected tables must be defined and mapped here
    #global reflected_table
    #reflected_table = sa.Table("Reflected", meta.metadata, autoload=True,
    #                           autoload_with=engine)
    #orm.mapper(Reflected, reflected_table)
    #
    meta.Session.configure(bind=engine)
    meta.engine = engine


t_persons = sa.Table("persons", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True),
    sa.Column("name", sa.types.String(100), primary_key=True),
    sa.Column("email", sa.types.String(100)),
    )

class Person(object):
    pass

orm.mapper(Person, t_persons)
```

This model has one table, "persons", assigned to the variable `t_persons`. `Person` is an ORM class which is tied to the table via the mapper.

If the table already exists, you can read its column definitions from the database rather than specifying them manually; this is called *reflecting* the table. The advantage is you don't have to specify the column types in Python code. Reflecting must be done inside `init_model()` because it depends on a live database engine, which is not available when the module is imported. (An *engine* is a SQLAlchemy object that knows how to connect to a particular database.) Here's the second example with reflection:

```
"""The application's model objects"""
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta

def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    # Reflected tables must be defined and mapped here
    global t_persons
    t_persons = sa.Table("persons", meta.metadata, autoload=True,
                         autoload_with=engine)
    orm.mapper(Person, t_persons)

    meta.Session.configure(bind=engine)
    meta.engine = engine
```

```
t_persons = None

class Person(object):
    pass
```

Note how `t_persons` and the `orm.mapper()` call moved into `init_model`, while the `Person` class didn't have to. Also note the `global t_persons` statement. This tells Python that `t_persons` is a global variable outside the function. `global` is required when assigning to a global variable inside a function. It's not required if you're merely modifying a mutable object in place, which is why `meta` doesn't have to be declared global.

SQLAlchemy 0.5 has an optional Declarative syntax which defines the table and the ORM class in one step:

```
"""The application's model objects"""
import sqlalchemy as sa
from sqlalchemy import orm
from sqlalchemy.ext.declarative import declarative_base

from myapp.model import meta

_Base = declarative_base()

def init_model(engine):
    """Call me before using any of the tables or classes in the model"""
    meta.Session.configure(bind=engine)
    meta.engine = engine


class Person(_Base):
    __tablename__ = "persons"

    id = sa.Column(sa.types.Integer, primary_key=True)
    name = sa.Column(sa.types.String(100))
    email = sa.Column(sa.types.String(100))
```

Relation example

Here's an example of a *Person* and an *Address* class with a many:many relationship on *people.my_addresses*. See Relational Databases for People in a Hurry and the SQLAlchemy manual for details.

```
t_people = sa.Table('people', meta.metadata,
    sa.Column('id', sa.types.Integer, primary_key=True),
    sa.Column('name', sa.types.String(100)),
    sa.Column('email', sa.types.String(100)),
    )

t_addresses_people = sa.Table('addresses_people', meta.metadata,
    sa.Column('id', sa.types.Integer, primary_key=True),
    sa.Column('person_id', sa.types.Integer, sa.ForeignKey('people.id')),
    sa.Column('address_id', sa.types.Integer, sa.ForeignKey('addresses.id')),
    )

t_addresses = sa.Table('addresses', meta.metadata,
    sa.Column('id', sa.types.Integer, primary_key=True),
    sa.Column('address', sa.types.String(100)),
    )
```

```
class Person(object):
    pass

class Address(object):
    pass

orm.mapper(Address, t_addresses)
orm.mapper(Person, t_people, properties = {
    'my_addresses' : orm.relation(Address, secondary = t_addresses_people),
    })
```

Using SQLAlchemy's SQL Layer

SQLAlchemy's lower level SQL expressions can be used along with your ORM models, and organizing them as class methods can be an effective way to keep the domain logic separate, and write efficient queries that return subsets of data that don't map cleanly to the ORM.

Consider the case that you want to get all the unique addresses from the relation example above. The following method in the Address class can make it easy:

```
# Additional imports
from sqlalchemy import select, func

from myapp.model.meta import Session


class Address(object):
    @classmethod
    def unique_addresses(cls):
        """Query the db for distinct addresses, return them as a list"""
        query = select([func.distinct(t_addresses.c.address).label('address')],
                       from_obj=[t_addresses])
        return [row['address'] for row in Session.execute(query).fetchall()]
```

See Also:

SQLAlchemy's SQL Expression Language Tutorial

Using the model standalone

You now have everything necessary to use the model in a standalone script such as a cron job, or to test it interactively. You just need to create a SQLAlchemy engine and connect it to the model. This example uses a database "test.sqlite" in the current directory:

```
% python
Python 2.5.1 (r251:54863, Oct 5 2007, 13:36:32)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlalchemy as sa
>>> engine = sa.create_engine("sqlite:///test.sqlite")
>>> from myapp import model
>>> model.init_model(engine)
```

Now you can use the tables, classes, and Session as described in the SQLAlchemy manual. For example:

```
#!/usr/bin/env python
import sqlalchemy as sa
import tmpapp.model as model
import tmpapp.model.meta as meta

DB_URL = "sqlite:///test.sqlite"

engine = sa.create_engine(DB_URL)
model.init_model(engine)

# Create all tables, overwriting them if they exist.
if hasattr(model, "_Base"):
    # SQLAlchemy 0.5 Declarative syntax
    model._Base.metadata.drop_all(bind=engine, checkfirst=True)
    model._Base.metadata.create_all(bind=engine)
else:
    # SQLAlchemy 0.4 and 0.5 syntax without Declarative
    meta.metadata.drop_all(bind=engine, checkfirst=True)
    meta.metadataa.create_all(bind=engine)

# Create two records and insert them into the database using the ORM.
a = model.Person()
a.name = "Aaa"
a.email = "aaa@example.com"
meta.Session.add(a)

b = model.Person()
b.name = "Bbb"
b.email = "bbb@example.com"
meta.Session.add(b)

meta.Session.commit()

# Display all records in the persons table.
print "Database data:"
for p in meta.Session.query(model.Person):
    print "id:", p.id
    print "name:", p.name
    print "email:", p.email
    print
```

6.3.3 The config file

When your Pylons application runs, it needs to know which database to connect to. Normally you put this information in *development.ini* and activate the model in *environment.py*. Put the following in *development.ini* in the *[app:main]* section, depending on your database,

For SQLite

```
sqlalchemy.url = sqlite:///%(here)s/mydatabasefilename.sqlite
```

Where *mydatabasefilename.db* is the path to your SQLite database file. "%(here)s" represents the directory containing the development.ini file. If you're using an absolute path, use four slashes after the colon: "sqlite:////var/lib/myapp/database.sqlite". Don't use a relative path (three slashes) because the current

directory could be anything. The example has three slashes because the value of "%(here)s" always starts with a slash (or the platform equivalent; e.g., "C:\foo" on Windows).

For MySQL

```
sqlalchemy.url = mysql://username:password@host:port/database
sqlalchemy.pool_recycle = 3600
```

Enter your username, password, host (localhost if it is on your machine), port number (usually 3306) and the name of your database. The second line is an example of setting engine options.

It's important to set "pool_recycle" for MySQL to prevent "MySQL server has gone away" errors. This is because MySQL automatically closes idle database connections without informing the application. Setting the connection lifetime to 3600 seconds (1 hour) ensures that the connections will be expired and recreated before MySQL notices they're idle.

Don't be tempted to use the ".echo" option to enable SQL logging because it may cause duplicate log output. Instead see the Logging section below to integrate MySQL logging into Paste's logging system.

For PostgreSQL

```
sqlalchemy.url = postgres://username:password@host:port/database
```

Enter your username, password, host (localhost if it is on your machine), port number (usually 5432) and the name of your database.

6.3.4 The engine

Put this at the top of *myapp/config/environment.py*:

```
from sqlalchemy import engine_from_config
from myapp.model import init_model
```

And this in the *load_environment* function:

```
engine = engine_from_config(config, 'sqlalchemy.')
init_model(engine)
```

The second argument is the prefix to look for. If you named your keys "sqlalchemy.default.url", you would put "sqlalchemy.default." here. The prefix may be anything, as long as it's consistent between the config file and this function call.

6.3.5 Controller

The paster create SQLAlchemy option adds the following to the top of *myapp/lib/base.py* (the base controller):

```
from myapp.model import meta
```

and also changes the .*__call__* method to:

```
def __call__(self, environ, start_response):
    try:
        return WSGIController.__call__(self, environ, start_response)
    finally:
        meta.Session.remove()
```

The .remove() method is so that any leftover ORM data in the current web request is discarded. This usually happens automatically as a product of garbage collection but calling .remove() ensures this is the case.

6.3.6 Building the database

To actually create the tables in the database, you call the metadata's *.create_all()* method. You can do this interactively or use *paster*'s application initialization feature. To do this, put the code in *myapp/websetup.py*. After the *load_environment()* call, put:

```
from myapp.model import meta
log.info("Creating tables")
meta.metadata.drop_all(bind=meta.engine, checkfirst=True)
meta.metadata.create_all(bind=meta.engine)
log.info("Successfully setup")
```

Or for the new SQLAlchemy 0.5 Declarative syntax:

```
from myapp import model
log.info("Creating tables")
model._Base.metadata.drop_all(bind=meta.engine, checkfirst=True)
model._Base.metadata.create_all(bind=meta.engine)
log.info("Successfully setup")
```

Then run the following on the command line:

```
$ paster setup-app development.ini
```

6.3.7 Data queries and modifications

Important: this section assumes you're putting the code in a high-level model function. If you're putting it directly into a controller method, you'll have to put a *model.* prefix in front of every object defined in the model, or import the objects individually. Also note that the *Session* object here (capital s) is not the same as the Beaker *session* object (lowercase s) in controllers.

Here's how to enter new data into the database:

```
mr_jones = Person()
mr_jones.name = 'Mr Jones'
meta.Session.add(mr_jones)
meta.Session.commit()
```

mr_jones here is an instance of *Person*. Its properties correspond to the column titles of *t_people* and contain the data from the selected row. A more sophisticated application would have a *Person.__init__* method that automatically sets attributes based on its arguments.

An example of loading a database entry in a controller method, performing a sex change, and saving it:

```
person_q = meta.Session.query(Person) # An ORM Query object for accessing the Person table
mr_jones = person_q.filter(Person.name=='Mr Jones').one()
print mr_jones.name # prints 'Mr Jones'
mr_jones.name = 'Mrs Jones' # only the object instance is changed here ...
meta.Session.commit() # ... only now is the database updated
```

To return a list of entries use:

```
all_mr_joneses = person_q.filter(Person.name=='Mr Jones').all()
```

To get all list of all the people in the table use:

```
everyone = person_q.all()
```

To retrieve by id:

```
someuser = person_q.get(5)
```

You can iterate over every person even more simply:

```
print "All people"
for p in person_q:
    print p.name
print
print "All Mr Joneses:"
for p in person_q.filter(Person.name=='Mr Jones'):
    print p.name
```

To delete an entry use the following:

```
mr_jones = person_q.filter(Person.name=='Mr Jones').one()
meta.Session.delete(mr_jones)
meta.Session.commit()
```

Working with joined objects

Recall that the *my_addresses* property is a list of *Address* objects

```
print mr_jones.my_addresses[0].address # prints first address
```

To add an existing address to 'Mr Jones' we do the following:

```
address_q = meta.Session.query(Address)

# Retrieve an existing address
address = address_q.filter(Address.address=='33 Pine Marten Lane, Pleasantville').one()

# Add to the list
mr_jones.my_addresses.append(new_address)

# issue updates to the join table
meta.Session.commit()
```

To add an entirely new address to 'Mr Jones' we do the following:

```
new_address = Address() # Construct an empty address object
new_address.address = '33 Pine Marten Lane, Pleasantville'
mr_jones.my_addresses.append(new_address) # Add to the list
meta.Session.commit() # Commit changes to the database
```

After making changes you must call *meta.Session.commit()* to store them permanently in the database; otherwise they'll be discarded at the end of the web request. You can also call *meta.Session.rollback()* at any time to undo any changes that haven't been committed.

To search on a joined object we can pass an entire object as a query:

```
search_address = Address()
search_address.address = '33 Pine Marten Lane, Pleasantville'
residents_at_33_pine_marten_lane = \
    person_q.filter(Person.my_addresses.contains(search_address)).all()
```

All attributes must match in the query object.

Or we can search on a joined objects' property,

```
residents_at_33_pine_marten_lane = \
 person_q.join('my_addresses').filter(
    Address.address=='33 Pine Marten Lane, Pleasantville').all()
```

A shortcut for the above is to use *any()*:

```
residents_at_33_pine_marten_lane = \
 person_q.filter(Person.my_addresses.any(
    Address.address=='33 Pine Marten Lane, Pleasantville')).all()
```

To disassociate an address from Mr Jones we do the following:

```
del mr_jones.my_addresses[0] # Delete the reference to the address
meta.Session.commit()
```

To delete the address itself in the address table, normally we'd have to issue a separate *delete()* for the *Address* object itself:

```
meta.Session.delete(mr_jones.my_addresses[0]) # Delete the Address object
del mr_jones.my_addresses[0]
meta.Session.commit() # Commit both operations to the database
```

However, SQLAlchemy supports a shortcut for the above operation. Configure the mapper relation using *cascade = "all, delete-orphan"* instead:

```
orm.mapper(Address, t_addresses)
orm.mapper(Person, t_people, properties = {
'my_addresses': orm.relation(
        Address, secondary=t_addresses_people, cascade="all,delete-orphan"),
})
```

Then, any items removed from *mr_jones.my_addresses* is automatically deleted from the database:

```
del mr_jones.my_addresses[0] # Delete the reference to the address,
                             # also deletes the Address
meta.Session.commit()
```

For any relationship, you can add *cascade = "all, delete-orphan"* as an extra argument to *relation()* in your mappers to ensure that when a join is deleted the joined object is deleted as well, so that the above delete() operation is not needed - only the removal from the *my_addresses* list. Beware though that despite its name, *delete-orphan* removes joined objects even if another object is joined to it.

Non-ORM SQL queries

Use *meta.Session.execute()* to execute a non-ORM SQL query within the session's transaction. Bulk updates and deletes can modify records significantly faster than looping through a query and modifying the ORM instances.

```
q = sa.select([table1.c.id, table1.c.name], order_by=[table1.c.name])
records = meta.Session.execute(q).fetchall()

# Example of a bulk SQL UPDATE.
update = table1.update(table1.c.name=="Jack")
meta.Session.execute(update, name="Ed")
meta.Session.commit()

# Example of updating all matching records using an expression.
update = table1.update(values={table1.c.entry_id: table1.c.entry_id + 1000})
meta.Session.exececute(update)
meta.Session.commit()

# Example of a bulk SQL DELETE.
delete = table1.delete(table1.c.name.like("M%"))
meta.Session.execute(delete)
meta.Session.commit()

# Database specific, use only if SQLAlchemy doesn't have methods to construct the desired query.
meta.Session.execute("ALTER TABLE Foo ADD new_column (VARCHAR(255)) NOT NULL")
```

Warning: The last example changes the database structure and may adversely interact with ORM operations.

Further reading

The Query object has many other features, including filtering on conditions, ordering the results, grouping, etc. These are excellently described in the SQLAlchemy manual. See especially the Data Mapping and Session / Unit of Work chapters.

6.3.8 Testing Your Models

Normal model usage works fine in model tests, however to use the metadata you must specify an engine connection for it. To have your tables created for every unit test in your project, use a test_models.py such as:

```
from myapp.tests import *
from myapp import model
from myapp.model import meta

class TestModels(TestController):

    def setUp(self):
        meta.Session.remove()
        meta.metadata.create_all(meta.engine)

    def test_index(self):
        # test your models
        pass
```

Note: Notice that the tests inherit from TestController. This is to ensure that the application is setup so that the models will work.

"nosetests –with-pylons=/path/to/test.ini ..." is another way to ensure that your model is properly initialized before the tests are run. This can be used when running non-controller tests.

6.3.9 Multiple engines

Some applications need to connect to multiple databases (engines). Some always bind certain tables to the same engines (e.g., a general database and a logging database); this is called "horizontal partitioning". Other applications have several databases with the same structure, and choose one or another depending on the current request. A blogging app with a separate database for each blog, for instance. A few large applications store different records from the same logical table in different databases to prevent the database size from getting too large; this is called "vertical partitioning" or "sharding". The pattern above can accommodate any of these schemes with a few minor changes.

First, you can define multiple engines in your config file like this:

```
sqlalchemy.default.url = "mysql://..."
sqlalchemy.default.pool_recycle = 3600
sqlalchemy.log.url = "sqlite://..."
```

This defines two engines, "default" and "log", each with its own set of options. Now you have to instantiate every engine you want to use.

```
default_engine = engine_from_config(config, 'sqlalchemy.default.')
log_engine = engine_from_config(config, 'sqlalchemy.log.')
init_model(default_engine, log_engine)
```

Of course you'll have to modify *init_model()* to accept both arguments and create two engines.

To bind different tables to different databases, but always with a particular table going to the same engine, use the *binds* argument to *sessionmaker* rather than *bind*:

```
binds={"table1": engine1, "table2": engine2}
Session = scoped_session(sessionmaker(binds=binds))
```

To choose the bindings on a per-request basis, skip the sessionmaker bind(s) argument, and instead put this in your base controller's *__call__* method before the superclass call, or directly in a specific action method:

```
meta.Session.configure(bind=meta.engine)
```

binds= works the same way here too.

6.3.10 Discussion on coding style, the Session object, and bound metadata

All ORM operations require a *Session* and an engine. All non-ORM SQL operations require an engine. (Strictly speaking, they can use a connection instead, but that's beyond the scope of this tutorial.) You can either pass the engine as the *bind*= argument to every SQLAlchemy method that does an actual database query, or bind the engine to a session or metadata. This tutorial recommends binding the session because that is the most flexible, as shown in the "Multiple Engines" section above.

It's also possible to bind a metadata to an engine using the *MetaData(engine)* syntax, or to change its binding with *metadata.bind = engine*. This would allow you to do autoloading without the *autoload_with* argument, and certain SQL operations without specifying an engine or session. Bound metadata was common in earlier versions of SQLAlchemy but is no longer recommended for beginners because it can cause unexpected behavior when ORM and non-ORM operations are mixed.

Don't confuse SQLAlchemy sessions and Pylons sessions; they're two different things! The *session* object used in controllers (*pylons.session*) is an industry standard used in web applications to maintain state between web requests by the same user. SQLAlchemy's session is an object that synchronizes ORM objects in memory with their corresponding records in the database.

The *Session* variable in this chapter is _not_ a SQLAlchemy session object; it's a "contextual session" class. Calling it returns the (new or existing) session object appropriate for this web request, taking into account threading and middleware issues. Calling its class methods (*Session.commit()*, *Session.query(...)*, etc) implicitly calls the corresponding method on the appropriate session. You can normally just call the *Session* class methods and ignore the internal session objects entirely. See "Contextual/Thread-local Sessions" in the SQLAlchemy manual for more information. This is equivalent to SQLAlchemy 0.3's *SessionContext* but with a different API.

"Transactional" sessions are a new feature in SQLAlchemy 0.4; this is why we're using *Session.commit()* instead of *Session.flush()*. The *autocommit=False* (*transactional=True* in SQLALchemy 0.4) and *autoflush=True* args (which are the defaults) to *sessionmaker* enable this, and should normally be used together.

6.3.11 Fancy classes

Here's an ORM class with some extra features:

```
class Person(object):

    def __init__(self, firstname, lastname, sex):
        if not firstname:
            raise ValueError("arg 'firstname' cannot be blank")
        if not lastname:
            raise ValueError("arg 'lastname' cannot be blank")
        if sex not in ["M", "F"]:
            raise ValueError("sex must be 'M' or 'F'")
        self.firstname = firstname
        self.lastname = lastname
        self.sex = sex

    def __repr__(self):
        myclass = self.__class__.__name__
        return "<%s %s %s>" % (myclass, self.firstname, self.lastname)
        #return "%s(%r, %r)" % (myclass, self.firstname, self.lastname, self.sex)
        #return "<%s %s>" % (self.firstname, self.lastname)

    @property
    def name(self):
        return "%s %s" % (self.firstname, self.lastname)

    @classmethod
    def all(cls, order=None, sex=None):
        """Return a Query of all Persons. The caller can iterate this,
        do q.count(), add additional conditions, etc.
        """
        q = meta.Session.query(Person)
        if order and order.lower().startswith("d"):
            q = q.order_by([Person.birthdate.desc()])
        else:
            q = q.order_by([Person.lastname, Person.firstname])
        return q

    @classmethod
```

```
    def recent(self, cutoff_days=30):
        cutoff = datetime.date.today() - datetime.timedelta(days=cutoff_days)
        q = meta.Session.query(Person).order_by(
                [Person.last_transaction_date.desc()])
        q = q.filter(Person.last_transaction_date >= cutoff)
        return q
```

With this class you can create new records with constructor args. This is not only convenient but ensures the record starts off with valid data (no required field empty). *.__init__* is not called when loading an existing record from the database, so it doesn't interfere with that. Instances can print themselves in a friendly way, and a read-only property is calculated from multiple fields.

Class methods return high-level queries for the controllers. If you don't like the class methods you can have a separate *PersonSearch* class for them. The methods get the session from the *myapp.model.meta* module where we've stored it. Note that this module imported the *meta* module, not the *Session* object directly. That's because *init_model()* replaces the *Session* object, so if we'd imported the *Session* object directly we'd get its original value rather than its current value.

You can do many more things in SQLAlchemy, such as a read-write property on a hidden column, or specify relations or default ordering in the *orm.mapper* call. You can make a composite property like *person.location.latitude* and *person.location.longitude* where *latitude* and *longitude* come from different table columns. You can have a class that mimics a list or dict but is associated with a certain table. Some of these properties you'll make with Pylons normal property mechanism; others you'll do with the *property* argument to *orm.mapper*. And you can have relations up the gazoo, which can be lazily loaded if you don't use one side of the relation much of the time, or eagerly loaded to minimize the number of queries. (Only the latter use SQL joins.) You can have certain columns in your class lazily loaded too, although SQLAlchemy calls this "deferred" rather than "lazy". SQLAlchemy will automatically load the columns or related table when they're accessed.

If you have any more clever ideas for fancy classes, please add a comment to this article.

6.3.12 Logging

SQLAlchemy has several loggers that chat about the various aspects of its operation. To log all SQL statements executed along with their parameter values, put the following in `development.ini`:

```
[logger_sqlalchemy]
level = INFO
handlers =
qualname = sqlalchemy.engine
```

Then modify the "[loggers]" section to enable your new logger:

```
[loggers]
keys = root, myapp, sqlalchemy
```

To log the results along with the SQL statements, set the level to DEBUG. This can cause a lot of output! To stop logging the SQL, set the level to WARN or ERROR.

SQLAlchemy has several other loggers you can configure in the same way. "sqlalchemy.pool" level INFO tells when connections are checked out from the engine's connection pool and when they're returned. "sqlalchemy.orm" and buddies log various ORM operations. See "Configuring Logging" in the SQLAlchemy manual.

6.3.13 Multiple application instances

If you're running multiple instances of the _same_ Pylons application in the same WSGI process (e.g., with Paste HTTPServer's "composite" application), you may run into concurrency issues. The problem is that `Session` is thread local but not application-instance local. We're not sure how much this is really an issue if `Session.remove()` is properly called in the base controller, but just in case it becomes an issue, here are possible remedies:

1. Attach the engine(s) to `pylons.g` (aka. `config["pylons.g"]`) rather than to the *meta* module. The globals object is not shared between application instances.
2. Add a scoping function. This prevents the application instances from sharing the same session objects. Add the following function to your model, and pass it as the second argument to *scoped_session*:

```
def pylons_scope():
    import thread
    from pylons import config
    return "Pylons|%s|%s" % (thread.get_ident(), config._current_obj())

Session = scoped_session(sessionmaker(), pylons_scope)
```

If you're affected by this, or think you might be, please bring it up on the pylons-discuss mailing list. We need feedback from actual users in this situation to verify that our advice is correct.

CHAPTER

SEVEN

CONFIGURATION

Pylons comes with two main ways to configure an application:

- The configuration file (*Runtime Configuration*)
- The application's `config` directory

The files in the `config` directory change certain aspects of how the application behaves. Any options that the webmaster should be able to change during deployment should be specified in a configuration file.

Tip: A good indicator of whether an option should be set in the `config` directory code vs. the configuration file is whether or not the option is necessary for the functioning of the application. If the application won't function without the setting, it belongs in the appropriate `config/` directory file. If the option should be changed depending on deployment, it belongs in the *Runtime Configuration*.

The applications `config/` directory includes:

- `config/environment.py` described in *Environment*
- `config/middleware.py` described in *Middleware*
- `config/deployment.ini_tmpl` described in *Production Configuration Files*
- `config/routing.py` described in *URL Configuration*

Each of these files allows developers to change key aspects of how the application behaves.

7.1 Runtime Configuration

When a new project is created a sample configuration file called `development.ini` is automatically produced as one of the project files. This default configuration file contains sensible options for development use, for example when developing a Pylons application it is very useful to be able to see a debug report every time an error occurs. The `development.ini` file includes options to enable debug mode so these errors are shown.

Since the configuration file is used to determine which application is run, multiple configuration files can be used to easily toggle sets of options. Typically a developer might have a `development.ini` configuration file for testing and a `production.ini` file produced by the **paster make-config** command for testing the command produces sensible production output. A `test.ini` configuration is also included in the project for test-specific options.

To specify a configuration file to use when running the application, change the last part of the **paster serve** to include the desired config file:

```
$ paster serve production.ini
```

See Also:

Configuration file format **and options** are described in great detail in the Paste Deploy documentation.

7.1.1 Getting Information From Configuration Files

All information from the configuration file is available in the `pylons.config` object. `pylons.config` also contains application configuration as defined in the project's `config.environment` module.

```
from pylons import config
```

`pylons.config` behaves like a dictionary. For example, if the configuration file has an entry under the `[app:main]` block:

```
cache_dir = %(here)s/data
```

That can then be read in the projects code:

```
from pylons import config
cache_dir = config['cache_dir']
```

Or the current debug status like this:

```
debug = config['debug']
```

Evaluating Non-string Data in Configuration Files

By default, all the values in the configuration file are considered strings. To make it easier to handle boolean values, the Paste library comes with a function that will convert `true` and `false` to proper Python boolean values:

```
from paste.deploy.converters import asbool

debug = asbool(config['debug'])
```

This is used already in the default projects' *Middleware* to toggle middleware that should only be used in development mode (with `debug`) set to true.

7.1.2 Production Configuration Files

To change the defaults of the configuration INI file that should be used when deploying the application, edit the `config/deployment.ini_tmpl` file. This is the file that will be used as a template during deployment, so that the person handling deployment has a starting point of the minimum options the application needs set.

One of the most important options set in the deployment ini is the `debug = true` setting. The email options should be setup so that errors can be e-mailed to the appropriate developers or webmaster in the event of an application error.

Generating the Production Configuration

To generate the production.ini file from the projects' `config/deployment.ini_tmpl` it must first be installed either as an *egg* or under development mode. Assuming the name of the Pylons application is `helloworld`, run:

```
$ paster make-config helloworld production.ini
```

Note: This command will also work from inside the project when its being developed.

It is the responsibility of the developer to ensure that a sensible set of default configuration values exist when the webmaster uses the `paster make-config` command.

Warning: **Always** make sure that the `debug` is set to `false` when deploying a Pylons application.

7.2 Environment

The `config/environment.py` module sets up the basic Pylons environment variables needed to run the application. Objects that should be setup once for the entire application should either be setup here, or in the `lib/app_globals __init__()` method.

It also calls the *URL Configuration* function to setup how the URL's will be matched up to *Controllers*, creates the *app_globals* object, configures which module will be referred to as *h*, and is where the template engine is setup.

When using SQLAlchemy it's recommended that the SQLAlchemy engine be setup in this module. The default SQLAlchemy configuration that Pylons comes with creates the engine here which is then used in `model/__init__.py`.

7.3 URL Configuration

A Python library called Routes handles mapping URLs to controllers and their methods, or their *action* as Routes refers to them. By default, Pylons sets up the following *route*s (found in `config/routing.py`):

```
map.connect('/{controller}/{action}')
map.connect('/{controller}/{action}/{id}')
```

Changed in version 0.9.7: Prior to Routes 1.9, all map.connect statements required variable parts to begin with a `:` like `map.connect(':controller/:action')`. This syntax is now optional, and the new `{}` syntax is recommended. Any part of the path inside the curly braces is a variable (a *variable part*) that will match any text in the URL for that 'part'. A 'part' of the URL is the text between two forward slashes. Every part of the URL must be present for the *route* to match, otherwise a 404 will be returned.

The routes above are translated by the Routes library into regular expressions for high performance URL matching. By default, all the variable parts (except for the special case of `{controller}`) become a matching regular expression of `[^/]+` to match anything except for a forward slash. This can be changed easily, for example to have the `{id}` only match digits:

```
map.connect('/{controller}/{action}/{id:\d+}')
```

If the desired regular expression includes the `{}`, then it should be specified separately for the variable part. To limit the `{id}` to only match at least 2-4 digits:

```
map.connect('/{controller}/{action}/{id}',  requirements=dict(id='\d{2,4}'))
```

The controller and action can also be specified as keyword arguments so that they don't need to be included in the URL:

```
# Archives by 2 digit year -> /archives/08
map.connect('/archives/{year:\d\d}', controller='articles',  action='archives')
```

Any variable part, or keyword argument in the `map.connect` statement will be available for use in the action used. For the route above, which resolves to the *articles* controller:

```
class ArticlesController(BaseController):

    def archives(self, year):
        ...
```

The part of the URL that matched as the year is available by name in the function argument.

Note: Routes also includes the ability to attempt to 'minimize' the URL. This behavior is generally not intuitive, and starting in Pylons 0.9.7 is turned off by default with the `map.minimization=False` setting.

The default mapping can match to any controller and any of their actions which means the following URLs will match:

```
/hello/index       >>    controller: hello, action: index
/entry/view/4      >>    controller: entry, action: view, id:4
/comment/edit/2    >>    controller: comment, action: edit, id:2
```

This simple scheme can be suitable for even large applications when complex URL's aren't needed.

Controllers can be organized into directories as well. For example, if the admins should have a separate `comments` controller:

```
$ paster controller admin/comments
```

Will create the `admin` directory along with the appropriate `comments` controller under it. To get to the comments controller:

```
/admin/comments/index    >>    controller: admin/comments, action: index
```

Note: The `{controller}` match is special, in that it doesn't always stop at the next forward slash (`/`). As the example above demonstrates, it is able to match controllers nested under a directory should they exist.

7.3.1 Adding a route to match /

The controller and action can be specified directly in the `map.connect()` statement, as well as the raw URL should be matched.

```
map.connect('/', controller='main', action='index')
```

will result in `/` being handled by the `index` method of the `main` controller.

7.3.2 Generating URLs

URLs are generated via the callable `routes.util.URLGenerator` object. Pylons provides an instance of this special object at `pylons.url`. It accepts keyword arguments indicating the desired controller, action and additional variables defined in a route.

```
# generates /content/view/2
url(controller='content', action='view', id=2)
```

To generate the URL of the matched route of the current request, call `routes.util.URLGenerator.current()`:

```
# Generates /content/view/3 during a request for /content/view/3
url.current()
```

`routes.util.URLGenerator.current()` also accepts the same arguments as *url()*. This uses Routes memory to generate a small change to the current URL without the need to specify all the relevant arguments:

```
# Generates /content/view/2 during a request for /content/view/3
url.current(id=2)
```

See Also:

Routes manual Full details and source code.

7.4 Middleware

A projects WSGI stack should be setup in the `config/middleware.py` module. Ideally this file should import middleware it needs, and set it up in the *make_app* function.

The default stack that is setup for a Pylons application is described in detail in *WSGI Middleware*.

Default middleware stack:

```
# The Pylons WSGI app
app = PylonsApp()

# Routing/Session/Cache Middleware
app = RoutesMiddleware(app, config['routes.map'])
app = SessionMiddleware(app, config)
app = CacheMiddleware(app, config)

# CUSTOM MIDDLEWARE HERE (filtered by error handling middlewares)

if asbool(full_stack):
    # Handle Python exceptions
    app = ErrorHandler(app, global_conf, **config['pylons.errorware'])

    # Display error documents for 401, 403, 404 status codes (and
    # 500 when debug is disabled)
    if asbool(config['debug']):
        app = StatusCodeRedirect(app)
    else:
        app = StatusCodeRedirect(app, [400, 401, 403, 404, 500])

# Establish the Registry for this application
app = RegistryManager(app)
```

```
if asbool(static_files):
    # Serve static files
    static_app = StaticURLParser(config['pylons.paths']['static_files'])
    app = Cascade([static_app, app])

return app
```

Since each piece of middleware wraps the one before it, the stack needs to be assembled in reverse order from the order in which its called. That is, the very last middleware that wraps the WSGI Application, is the very first that will be called by the server.

The last piece of middleware in the stack, called Cascade, is used to serve static content files during development. For top performance, consider disabling the Cascade middleware via setting the `static_files = false` in the configuration file. Then have the webserver or a *CDN* serve static files.

Warning: When unsure about whether or not to change the middleware, **don't**. The order of the middleware is important to the proper functioning of a Pylons application, and shouldn't be altered unless needed.

7.4.1 Adding custom middleware

Custom middleware should be included in the `config/middleware.py` at comment marker:

```
# CUSTOM MIDDLEWARE HERE (filtered by error handling middlewares)
```

For example, to add a middleware component named *MyMiddleware*, include it in `config/middleware.py`:

```
# The Pylons WSGI app
app = PylonsApp()

# Routing/Session/Cache Middleware
app = RoutesMiddleware(app, config['routes.map'])
app = SessionMiddleware(app, config)
app = CacheMiddleware(app, config)

# CUSTOM MIDDLEWARE HERE (filtered by error handling middlewares)
app = MyMiddleware(app)
```

The app object is simply passed as a parameter to the *MyMiddleware* middleware which in turn should return a wrapped WSGI application.

Care should be taken when deciding in which layer to place custom middleware. In most cases middleware should be placed before the Pylons WSGI application and its supporting Routes/Session/Cache middlewares, however if the middleware should run *after* the CacheMiddleware:

```
# Routing/Session/Cache Middleware
app = RoutesMiddleware(app, config['routes.map'])
app = SessionMiddleware(app, config)

# MyMiddleware can only see the cache object, nothing *above* here
app = MyMiddleware(app)

app = CacheMiddleware(app, config)
```

7.4.2 What is full_stack?

In the Pylons ini file {`development.ini` or `production.ini`} this block determines if the flag full_stack is set to true or false:

```
[app:main]
use = egg:app_name
full_stack = true
```

The full_stack flag determines if the ErrorHandler and StatusCodeRedirect is included as a layer in the middleware wrapping process. The only condition in which this option would be set to *false* is if multiple Pylons applications are running and will be wrapped in the appropriate middleware elsewhere.

7.5 Application Setup

There are two kinds of 'Application Setup' that are occasionally referenced with regards to a project using Pylons.

- Setting up a new application
- Configuring project information and package dependencies

7.5.1 Setting Up a New Application

To make it easier to setup a new instance of a project, such as setting up the basic database schema, populating necessary defaults, etc. a setup script can be created.

In a Pylons project, the setup script to be run is located in the projects' `websetup.py` file. The default script loads the projects configuration to make it easier to write application setup steps:

```
import logging

from helloworld.config.environment import load_environment

log = logging.getLogger(__name__)

def setup_app(command, conf, vars):
    """Place any commands to setup helloworld here"""
    load_environment(conf.global_conf, conf.local_conf)
```

Note: If the project was configured during creation to use SQLAlchemy this file will include some commands to setup the database connection to make it easier to setup database tables.

To run the setup script using the development configuration:

```
$ paster setup-app development.ini
```

7.5.2 Configuring the Package

A newly created project with Pylons is a standard Python package. As a Python package, it has a `setup.py` file that records meta-information about the package. Most of the options in it are fairly self-explanatory, the most important being the 'install_requires' option:

```
install_requires=[
    "Pylons>=0.9.7",
],
```

These lines indicate what packages are required for the proper functioning of the application, and should be updated as needed. To re-parse the `setup.py` line for new dependencies:

```
$ python setup.py develop
```

In addition to updating the packages as needed so that the dependency requirements are made, this command will ensure that this package is active in the system (without requiring the traditional **python setup.py install**).

See Also:

Declaring Dependencies

CHAPTER

EIGHT

LOGGING

8.1 Logging messages

As of Pylons 0.9.6, Pylons controllers (created via `paster controller/restcontroller`) and `websetup.py` create their own Logger objects via Python's logging module.

For example, in the helloworld project's hello controller (`helloworld/controllers/hello.py`):

```
import logging

from pylons import request, response, session, tmpl_context as c
from pylons.controllers.util import abort, redirect_to

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        ...
```

Python's special `__name__` variable refers to the current module's fully qualified name; in this case, `helloworld.controllers.hello`.

To log messages, simply use methods available on that Logger object:

```
import logging

from pylons import request, response, session, tmpl_context as c
from pylons.controllers.util import abort, redirect_to

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        content_type = 'text/plain'
        content = 'Hello World!'

        log.debug('Returning: %s (content-type: %s)', content, content_type)
        response.content_type = content_type
        return content
```

Which will result in the following printed to the console, on stderr:

```
16:20:20,440 DEBUG [helloworld.controllers.hello] Returning: Hello World!
                   (content-type: text/plain)
```

8.2 Basic Logging configuration

As of Pylons 0.9.6, the default ini files include a basic configuration for the logging module. Paste ini files use the Python standard ConfigParser format; the same format used for the Python logging module's Configuration file format.

`paster`, when loading an application via the `paster serve`, `shell` or `setup-app` commands, calls the logging.fileConfig function on that specified ini file if it contains a 'loggers' entry. `logging.fileConfig` reads the logging configuration from a `ConfigParser` file.

Logging configuration is provided in both the default `development.ini` and the production ini file (created via `paster make-config <package_name> <ini_file>`). The production ini's logging setup is a little simpler than the `development.ini`'s, and is as follows:

```
# Logging configuration
[loggers]
keys = root

[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] %(message)s
```

One root Logger is created that logs only 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 [helloworld.controllers.hello] Loading resource, id: 86
```

For those familiar with the `logging.basicConfig` function, this configuration is equivalent to the code:

```
logging.basicConfig(level=logging.INFO,
format='%(asctime)s %(levelname)-5.5s [%(name)s] %(message)s')
```

The default `development.ini`'s logging section has a couple of differences: it uses a less verbose timestamp, and defaults your application's log messages to the `DEBUG` level (described in the next section).

Pylons and many other libraries (such as Beaker, SQLAlchemy, Paste) log a number of messages for debugging purposes. Switching the root Logger level to `DEBUG` reveals them:

```
[logger_root]
#level = INFO
level = DEBUG
handlers = console
```

8.3 Filtering log messages

Often there's too much log output to sift through, such as when switching the root Logger's level to DEBUG.

An 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, 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:

```
# Logging configuration
[loggers]
keys = root, helloworld
```

```
[logger_helloworld]
level = DEBUG
handlers =
qualname = helloworld
```

All of the child Loggers of the helloworld Logger will inherit the DEBUG level unless they're explicitly set differently. Meaning the `helloworld.controllers.hello`, `helloworld.websetup` (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 Filter object; however it cannot be used directly from the configuration file.

8.4 Advanced Configuration

To capture log output to a separate file, use a FileHandler (or a RotatingFileHandler):

```
[handler_accesslog]
class = FileHandler
args = ('access.log','a')
level = INFO
formatter = generic
```

Before it's recognized, it needs to be added to the list of Handlers:

```
[handlers]
keys = console, accesslog
```

and finally utilized by a Logger.

```
[logger_root]
level = INFO
handlers = console, accesslog
```

These final 3 lines of configuration directs all of the root Logger's output to the access.log as well as the console; we'll want to disable this for the next section.

8.5 Request logging with Paste's TransLogger

Paste provides the TransLogger middleware for logging requests using the Apache Combined Log Format. TransLogger combined with a FileHandler can be used 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 in the `[app:main]` section of the ini file:

```
filter-with = translogger

[filter:translogger]
use = egg:Paste#translogger
setup_console_handler = False
```

This is equivalent to wrapping your app in a TransLogger instance via the bottom of your project's `config/middleware.py` file:

```
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 we've configured our own logging Handlers, we need to disable that option 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 [helloworld.controllers.hello] Returning: Hello World!
                  (content-type: text/plain)
00:50:53,695 INFO [wsgi] 192.168.1.111 - - [11/Aug/2007:20:09:33 -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 the `access.log` FileHandler defined above, we need to add that FileHandler to the wsgi Logger's list of Handlers:

```
# Logging configuration
[loggers]
keys = root, wsgi
```

```
[logger_wsgi]
level = INFO
```

```
handlers = handler_accesslog
qualname = wsgi
propagate = 0
```

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:

```
[formatters]
keys = generic, accesslog
```

```
[formatter_accesslog]
format = %(message)s
```

Then wire this new `accesslog` Formatter into the FileHandler:

```
[handler_accesslog]
class = FileHandler
args = ('access.log','a')
level = INFO
formatter = accesslog
```

8.6 Logging to wsgi.errors

Pylons provides a custom logging Handler class, pylons.log.WSGIErrorsHandler, for logging output to `environ['wsgi.errors']`: the WSGI server's error stream (see the WSGI Spefification, PEP 333 for more information). `wsgi.errors` can be useful to log to in certain situations, such as when deployed under Apache mod_wsgi/mod_python, where the `wsgi.errors` stream is the Apache error log.

To configure logging of only `ERROR` (and `CRITICAL`) messages to `wsgi.errors`, add the following to the ini file:

```
[handlers]
keys = console, wsgierrors
```

```
[handler_wsgierrors]
class = pylons.log.WSGIErrorsHandler
args = ()
level = ERROR
format = generic
```

then add the new Handler name to the list of Handlers used by the root Logger:

```
[logger_root]
level = INFO
handlers = console, wsgierrors
```

Warning: `WSGIErrorsHandler` does not receive log messages created during application startup. This is due to the `wsgi.errors` stream only being available through the `environ` dictionary; which isn't available until a request is made.

8.6.1 Lumberjacking with log4j's Chainsaw

Java's `log4j` project provides the Java GUI application Chainsaw for viewing and managing log messages. Among its features are the ability to filter log messages on the fly, and customizable color highlighting of log messages.

We can configure Python's logging module to output to a format parsable by Chainsaw, `log4j`'s XMLLayout format.

To do so, we first need to install the Python XMLLayout package:

```
$ easy_install XMLLayout
```

It provides a log Formatter that generates `XMLLayout` XML. It also provides `RawSocketHandler`; like the logging module's `SocketHandler`, it sends log messages across the network, but does not pickle them.

The following is an example configuration for sending `XMLLayout` log messages across the network to Chainsaw, if it were listening on *localhost* port *4448*:

```
[handlers]
keys = console, chainsaw

[formatters]
keys = generic, xmllayout

[logger_root]
level = INFO
handlers = console, chainsaw
```

```
[handler_chainsaw]
class = xmllayout.RawSocketHandler
args = ('localhost', 4448)
level = NOTSET
formatter = xmllayout
```

```
[formatter_xmllayout]
class = xmllayout.XMLLayout
```

This configures any log messages handled by the root Logger to also be sent to Chainsaw. The default `development.ini` configures the root Logger to the `INFO` level, however in the case of using Chainsaw, it is preferable to configure the root Logger to `NOTSET` so *all* log messages are sent to Chainsaw. Instead, we can restrict the console handler to the `INFO` level:

```
[logger_root]
level = NOTSET
handlers = console

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = INFO
formatter = generic
```

Chainsaw can be downloaded from its home page, but can also be launched directly from a Java-enabled browser via the link: Chainsaw web start.

It can be configured from the GUI, but it also supports reading its configuration from a `log4j.xml` file.

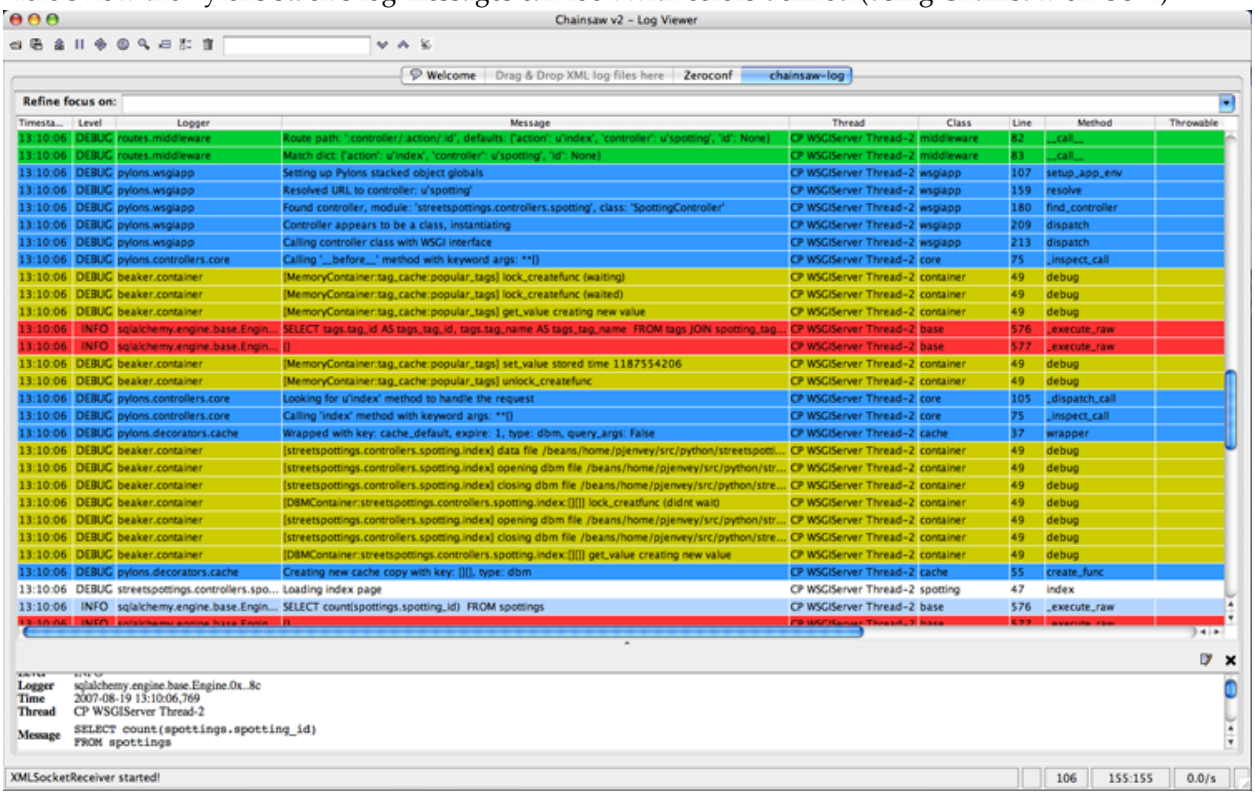
The following `log4j.xml` file configures Chainsaw to listen on port *4448* for `XMLLayout` style log messages. It also hides Chainsaw's own logging messages under the `WARN` level, so only your app's log mes-

sages are displayed:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration>
<configuration xmlns="http://logging.apache.org/">

<plugin name="XMLSocketReceiver" class="org.apache.log4j.net.XMLSocketReceiver">
    <param name="decoder" value="org.apache.log4j.xml.XMLDecoder"/>
    <param name="port" value="4448"/>
</plugin>

<logger name="org.apache.log4j">
    <level value="warn"/>
</logger>

<root>
    <level value="debug"/>
</root>

</configuration>
```

Chainsaw will prompt for a configuration file upon startup. The configuration can also be loaded later by clicking *File/Load Log4J File...*. You should see an XMLSocketReceiver instance loaded in Chainsaw's Receiver list, configured at port *4448*, ready to receive log messages.

Here's how the Pylons stack's log messages can look with colors defined (using Chainsaw on OS X):



8.6.2 Alternate Logging Configuration style

Pylons' default ini files include a basic configuration for Python's logging module. Its format matches the standard Python `logging` module's config file format . If a more concise format is preferred, here is Max

Ischenko's demonstration of an alternative style to setup logging.

The following function is called at the application start up (e.g. Global ctor):

```
def setup_logging():
    logfile = config['logfile']
    if logfile == 'STDOUT': # special value, used for unit testing
        logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
               #format='%(name)s %(levelname)s %(message)s',
               #format='%(asctime)s,%(msecs)d %(levelname)s %(message)s',
               format='%(asctime)s,%(msecs)d %(name)s %(levelname)s %(message)s',
               datefmt='%H:%M:%S')
    else:
        logdir = os.path.dirname(os.path.abspath(logfile))
        if not os.path.exists(logdir):
            os.makedirs(logdir)
        logging.basicConfig(filename=logfile, mode='at+',
             level=logging.DEBUG,
             format='%(asctime)s,%(msecs)d %(name)s %(levelname)s %(message)s',
             datefmt='%Y-%b-%d %H:%M:%S')
    setup_thirdparty_logging()
```

The setup_thirdparty_logging function searches through the certain keys of the application `.ini` file which specify logging level for a particular logger (module).

```
def setup_thirdparty_logging():
    for key in config:
        if not key.endswith('logging'):
            continue
        value = config.get(key)
        key = key.rstrip('.logging')
        loglevel = logging.getLevelName(value)
        log.info('Set %s logging for %s', logging.getLevelName(loglevel), key)
        logging.getLogger(key).setLevel(loglevel)
```

Relevant section of the .ini file (example):

```
sqlalchemy.logging = WARNING
sqlalchemy.orm.unitofwork.logging = INFO
sqlalchemy.engine.logging = DEBUG
sqlalchemy.orm.logging = INFO
routes.logging = WARNING
```

This means that routes logger (and all sub-loggers such as routes.mapper) only passes through messages of at least WARNING level; sqlalachemy defaults to WARNING level but some loggers are configured with more verbose level to aid debugging.

CHAPTER

NINE

HELPERS

Helpers are functions intended for usage in templates, to assist with common HTML and text manipulation, higher level constructs like a HTML tag builder (that safely escapes variables), and advanced functionality like Pagination of data sets.

The majority of the helpers available in Pylons are provided by the `webhelpers` package. Some of these helpers are also used in controllers to prepare data for use in the template by other helpers, such as the `secure_form_tag()` function which has a corresponding `authenticate_form()`.

To make individual helpers available for use in templates under *h*, the appropriate functions need to be imported in `lib/helpers.py`. All the functions available in this file are then available under *h* just like any other module reference.

By customizing the `lib/helpers.py` module you can quickly add custom functions and classes for use in your templates.

Helper functions are organized into modules by theme. All HTML generators are under the `webhelpers_html` package, except for a few third-party modules which are directly under `webhelpers`. The webhelpers modules are separately documented, see `webhelpers`.

9.1 Pagination

Note: The *paginate* module is not compatible to the deprecated *pagination* module that was provided with former versions of the Webhelpers package.

9.1.1 Purpose of a paginator

When you display large amounts of data like a result from an SQL query then usually you cannot display all the results on a single page. It would simply be too much. So you divide the data into smaller chunks. This is what a paginator does. It shows one page of chunk of data at a time. Imagine you are providing a company phonebook through the web and let the user search the entries. Assume the search result contains 23 entries. You may decide to display no more than 10 entries per page. The first page contains entries 1-10, the second 11-20 and the third 21-23. And you also show a navigational element like `Page 1 of 3: [1] 2 3` that allows the user to switch between the available pages.

9.1.2 The Page class

The webhelpers package provides a *paginate* module that can be used for this purpose. It can create pages from simple Python lists as well as SQLAlchemy queries and SQLAlchemy select objects. The module provides a Page object that represents a single page of items from a larger result set. Such a Page mainly behaves like a list of items on that page. Let's take the above example of 23 items spread across 3 pages:

```
# Create a list of items from 1 to 23
>>> items = range(1,24)

# Import the paginate module
>>> import webhelpers.paginate

# Create a Page object from the 'items' for the second page
>>> page2 = webhelpers.paginate.Page(items, page=2, items_per_page=10)

# The Page object can be printed (__repr__) to show details on the page
>>> page2

    Page:
    Collection type:  <type 'list'>
    (Current) page:   2
    First item:       11
    Last item:        20
    First page:       1
    Last page:        3
    Previous page:    1
    Next page:        3
    Items per page:   10
    Number of items:  23
    Number of pages:  3

# Show the items on this page
>>> list(page2)

    [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

# Print the items in a for loop
>>> for i in page2: print "This is entry", i

    This is entry 11
    This is entry 12
    This is entry 13
    This is entry 14
    This is entry 15
    This is entry 16
    This is entry 17
    This is entry 18
    This is entry 19
    This is entry 20
```

There are further parameters to invoking a Page object. Please see webhelpers.paginate.Page

---

Note: Page numbers and item numbers start from 1. If you are accessing the items on the page by their index please note that the first item is item[1] instead of item[0].

9.1.3 Switching between pages using a *pager*

The user needs a way to get to another page. This is usually done with a list of links like `Page 3 of 41 - 1 2 [3] 4 5 .. 41`. Such a list can be created by the Page's `pager()` method. Take the above example again:

```
>>> page2.pager()

    <a class="pager_link" href="/content?page=1">1</a>
    <span class="pager_curpage">2</span>
    <a class="pager_link" href="/content?page=3">3</a>
```

Without the HTML tags it looks like `1 [2] 3`. The links point to a URL where the respective page is found. And the current page (2) is highlighted.

The appearance of a pager can be customized. By default the format string is `~2~` which means it shows adjacent pages from the current page with a maximal radius of 2. In a larger set this would look like `1 .. 34 35 [36] 37 38 .. 176`. The radius of 2 means that two pages before and after the current page 36 are shown.

Several special variables can be used in the format string. See `pager()` for a complete list. Some examples for a pager of 20 pages while being on page 10 currently:

```
>>> page.pager()

    1 .. 8 9 [10] 11 12 .. 20

>>> page.pager('~4~')

    1 .. 6 7 8 9 [10] 11 12 13 14 .. 20

>>> page.pager('Page $page of $page_count - ~3~')

    Page 10 of 20 - 1 .. 7 8 9 [10] 11 12 13 .. 20

>>> page.pager('$link_previous $link_next ~2~')

    < > 1 .. 8 9 [10] 11 12 .. 20

>>> page.pager('Items $first_item - $last_item / ~2~')

    Items 91 - 100 / 1 .. 8 9 [10] 11 12 .. 20
```

9.1.4 Paging over an SQLAlchemy query

If the data to page over comes from a database via SQLAlchemy then the `paginate` module can access a `query` object directly. This is useful when using ORM-mapped models. Example:

```
>>> employee_query = Session.query(Employee)
>>> page2 = webhelpers.paginate.Page(
        employee_query,
        page=2,
        items_per_page=10)
>>> for employee in page2: print employee.first_name

    John
    Jack
    Joseph
```

```
    Kay
    Lars
    Lynn
    Pamela
    Sandra
    Thomas
    Tim
```

The *paginate* module is smart enough to only query the database for the objects that are needed on this page. E.g. if a page consists of the items 11-20 then SQLAlchemy will be asked to fetch exactly that 10 rows through *LIMIT* and *OFFSET* in the actual SQL query. So you must not load the complete result set into memory and pass that. Instead always pass a *query* when creating a *Page*.

9.1.5 Paging over an SQLAlchemy select

SQLAlchemy also allows to run arbitrary SELECTs on database tables. This is useful for non-ORM queries. *paginate* can use such select objects, too. Example:

```
>>> selection = sqlalchemy.select([Employee.c.first_name])
>>> page2 = webhelpers.paginate.Page(
        selection,
        page=2,
        items_per_page=10,
        sqlalchemy_session=model.Session)
>>> for first_name in page2: print first_name

    John
    Jack
    Joseph
    Kay
    Lars
    Lynn
    Pamela
    Sandra
    Thomas
    Tim
```

The only difference to using SQLAlchemy *query* objects is that you need to pass an SQLAlchemy *session* via the `sqlalchemy_session` parameter. A bare `select` does not have a database connection assigned. But the session has.

9.1.6 Usage in a Pylons controller and template

A simple example to begin with.

Controller:

```
def list(self):
    c.employees = webhelpers.paginate.Page(
        model.Session.query(model.Employee),
        page = int(request.params['page']),
        items_per_page = 5)
    return render('/employees/list.mako')
```

Template:

```
${c.employees.pager('Page $page: $link_previous $link_next ~4~')}
<ul>
% for employee in c.employees:
    <li>${employee.first_name} ${employee.last_name}</li>
% endfor
</ul>
```

The *pager()* creates links to the previous URL and just sets the *page* parameter appropriately. That's why you need to pass the requested page number (`request.params['page']`) when you create a *Page*.

9.1.7 Partial updates with AJAX

Updating a page partially is easy. All it takes is a little Javascript that - instead of loading the complete page - updates just the part of the page containing the paginated items. The `pager()` method accepts an `onclick` parameter for that purpose. This value is added as an `onclick` parameter to the A-HREF tags. So the `href` parameter points to a URL that loads the complete page while the `onclick` parameter provides Javascript that loads a partial page. An example (using the jQuery Javascript library for simplification) may help explain that.

Controller:

```
def list(self):
    c.employees = webhelpers.paginate.Page(
        model.Session.query(model.Employee),
        page = int(request.params['page']),
        items_per_page = 5)
    if 'partial' in request.params:
        # Render the partial page
        return render('/employees/list-partial.mako')
    else:
        # Render the full page
        return render('/employees/list-full.mako')
```

Template `list-full.mako`:

```
<html>
    <head>
        ${webhelpers.html.tags.javascript_link('/public/jQuery.js')}
    </head>
    <body>
        <div id="page-area">
            <%include file="list-partial.mako"/>
        </div>
    </body>
</html>
```

Template `list-partial.mako`:

```
${c.employees.pager(
    'Page $page: $link_previous $link_next ~4~',
    onclick="$('#my-page-area').load('%s'); return false;")}
<ul>
% for employee in c.employees:
    <li>${employee.first_name} ${employee.last_name}</li>
% endfor
</ul>
```

To avoid code duplication in the template the full template includes the partial template. If a partial page load is requested then just the `list-partial.mako` gets rendered. And if a full page load is requested then the `list-full.mako` is rendered which in turn includes the `list-partial.mako`.

The `%s` variable in the `onclick` string gets replaced with a URL pointing to the respective page with a `partial=1` added (the name of the parameter can be customized through the `partial_param` parameter). Example:

- `href` parameter points to `/employees/list?page=3`
- `onclick` parameter contains Javascript loading `/employees/list?page=3&partial=1`

jQuery's syntax to load a URL into a certain DOM object (e.g. a DIV) is simply:

```
$('#some-id').load('/the/url')
```

The advantage of this technique is that it degrades gracefully. If the user does not have Javascript enabled then a full page is loaded. And if Javascript works then a partial load is done through the `onclick` action.

9.2 Secure Form Tag Helpers

For prevention of Cross-site request forgery (CSRF) attacks.

Generates form tags that include client-specific authorization tokens to be verified by the destined web app.

Authorization tokens are stored in the client's session. The web app can then verify the request's submitted authorization token with the value in the client's session.

This ensures the request came from the originating page. See the wikipedia entry for Cross-site request forgery for more information.

Pylons provides an `authenticate_form` decorator that does this verification on the behalf of controllers.

These helpers depend on Pylons' `session` object. Most of them can be easily ported to another framework by changing the API calls.

The helpers are implemented in such a way that it should be easy for developers to create their own helpers if using helpers for AJAX calls.

`authentication_token()` returns the current authentication token, creating one and storing it in the session if it doesn't already exist.

`auth_token_hidden_field()` creates a hidden field containing the authentication token.

`secure_form()` is `form()` plus `auth_token_hidden_field()`.

CHAPTER

TEN

FORMS

10.1 The basics

When a user submits a form on a website the data is submitted to the URL specified in the *action* attribute of the <*form*> tag. The data can be submitted either via HTTP *GET* or *POST* as specified by the *method* attribute of the <*form*> tag. If your form doesn't specify an *action*, then it's submitted to the current URL, generally you'll want to specify an *action*. When a file upload field such as <*input type="file" name="file" />* is present, then the HTML <*form*> tag must also specify *enctype="multipart/form-data"* and *method* must be *POST*.

10.2 Getting Started

Add two actions that looks like this:

```
# in the controller

def form(self):
    return render('/form.mako')

def email(self):
    return 'Your email is: %s' % request.params['email']
```

Add a new template called *form.mako* in the *templates* directory that contains the following:

```
<form name="test" method="GET" action="/hello/email">
Email Address: <input type="text" name="email" />
<input type="submit" name="submit" value="Submit" />
</form>
```

If the server is still running (see the *Getting Started Guide*) you can visit http://localhost:5000/hello/form and you will see the form. Try entering the email address *test@example.com* and clicking Submit. The URL should change to `http://localhost:5000/hello/email?email=test%40example.com` and you should see the text *Your email is test@example.com*.

In Pylons all form variables can be accessed from the `request.params` object which behaves like a dictionary. The keys are the names of the fields in the form and the value is a string with all the characters entity decoded. For example note how the @ character was converted by the browser to *%40* in the URL and was converted back ready for use in `request.params`.

Note: *request* and *response* are objects from the *WebOb* library. Full documentation on their attributes and methods is here.

If you have two fields with the same name in the form then using the dictionary interface will return the first string. You can get all the strings returned as a list by using the *.getall()* method. If you only expect one value and want to enforce this you should use *.getone()* which raises an error if more than one value with the same name is submitted.

By default if a field is submitted without a value, the dictionary interface returns an empty string. This means that using *.get(key, default)* on *request.params* will only return a default if the value was not present in the form.

10.2.1 POST vs GET and the Re-Submitted Data Problem

If you change the *form.mako* template so that the method is *POST* and you re-run the example you will see the same message is displayed as before. However, the URL displayed in the browser is simply http://localhost:5000/hello/email without the query string. The data is sent in the body of the request instead of the URL, but Pylons makes it available in the same way as for GET requests through the use of *request.params*.

Note: If you are writing forms that contain password fields you should usually use POST to prevent the password being visible to anyone who might be looking at the user's screen.

When writing form-based applications you will occasionally find users will press refresh immediately after submitting a form. This has the effect of repeating whatever actions were performed the first time the form was submitted but often the user will expect that the current page be shown again. If your form was submitted with a POST, most browsers will display a message to the user asking them if they wish to re-submit the data, this will not happen with a GET so POST is preferable to GET in those circumstances.

Of course, the best way to solve this issue is to structure your code differently so:

```
# in the controller

def form(self):
    return render('/form.mako')

def email(self):
    # Code to perform some action based on the form data
    # ...
    redirect_to(action='result')

def result(self):
    return 'Your data was successfully submitted'
```

In this case once the form is submitted the data is saved and an HTTP redirect occurs so that the browser redirects to http://localhost:5000/hello/result. If the user then refreshes the page, it simply redisplays the message rather than re-performing the action.

10.3 Using the Helpers

Creating forms can also be done using WebHelpers, which comes with Pylons. Here is the same form created in the previous section but this time using the helpers:

```
${h.form(h.url(action='email'), method='get')}
Email Address: ${h.text('email')}
```

```
${h.submit('Submit')}
${h.end_form()}
```

Before doing this you'll have to import the helpers you want to use into your project's *lib/helpers.py* file; then they'll be available under Pylons' `h` global. Most projects will want to import at least these:

```
from webhelpers.html import escape, HTML, literal, url_escape
from webhelpers.html.tags import *
```

There are many other helpers for text formatting, container objects, statistics, and for dividing large query results into pages. See the `WebHelpers documentation` to choose the helpers you'll need.

10.4 File Uploads

File upload fields are created by using the *file* input field type. The *file_field* helper provides a shortcut for creating these form fields:

```
${h.file_field('myfile')}
```

The HTML form must have its *enctype* attribute set to *multipart/form-data* to enable the browser to upload the file. The *form* helper's *multipart* keyword argument provides a shortcut for setting the appropriate *enctype* value:

```
${h.form(h.url(action='upload'), multipart=True)}
Upload file: ${h.file_field('myfile')} <br />
File description: ${h.text_field('description')} <br />
${h.submit('Submit')}
${h.end_form()}
```

When a file upload has succeeded, the *request.POST* (or *request.params*) *MultiDict* will contain a *cgi.FieldStorage* object as the value of the field.

FieldStorage objects have three important attributes for file uploads:

filename The name of file uploaded as it appeared on the uploader's filesystem.

file A file(-like) object from which the file's data can be read: A python *tempfile* or a *StringIO* object.

value The content of the uploaded file, eagerly read directly from the file object.

The easiest way to gain access to the file's data is via the *value* attribute: it returns the entire contents of the file as a string:

```
def upload(self):
    myfile = request.POST['myfile']
    return 'Successfully uploaded: %s, size: %i, description: %s' % \
        (myfile.filename, len(myfile.value), request.POST['description'])
```

However reading the entire contents of the file into memory is undesirable, especially for large file uploads. A common means of handling file uploads is to store the file somewhere on the filesystem. The *FieldStorage* typically reads the file onto filesystem, however to a non permanent location, via a python *tempfile* object (though for very small uploads it stores the file in a *StringIO* object instead).

Python *tempfiles* are secure file objects that are automatically destroyed when they are closed (including an implicit close when the object is garbage collected). One of their security features is that their path cannot be determined: a simple *os.rename* from the *tempfile's* path isn't possible. Alternatively, *shutil.copyfileobj* can perform an efficient copy of the file's data to a permanent location:

```
permanent_store = '/uploads/'

class Uploader(BaseController):
    def upload(self):
        myfile = request.POST['myfile']
        permanent_file = open(os.path.join(permanent_store,
                                   myfile.filename.lstrip(os.sep)),
                                   'w')

    shutil.copyfileobj(myfile.file, permanent_file)
    myfile.file.close()
    permanent_file.close()

    return 'Successfully uploaded: %s, description: %s' % \
        (myfile.filename, request.POST['description'])
```

Warning: The previous basic example allows any file uploader to overwrite any file in the *permanent_store* directory that your web application has permissions to.

Also note the use of *myfile.filename.lstrip(os.sep)* here: without it, *os.path.join* is unsafe. *os.path.join* won't join absolute paths (beginning with *os.sep*), i.e. *os.path.join('/uploads/', '/uploaded_file.txt') == '/uploaded_file.txt'*. Always check user submitted data to be used with *os.path.join*.

10.5 Validating user input with FormEncode

10.5.1 Validation the Quick Way

At the moment you could enter any value into the form and it would be displayed in the message, even if it wasn't a valid email address. In most cases this isn't acceptable since the user's input needs validating. The recommended tool for validating forms in Pylons is FormEncode.

For each form you create you also create a validation schema. In our case this is fairly easy:

```
import formencode

class EmailForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    email = formencode.validators.Email(not_empty=True)
```

Note: We usually recommend keeping form schemas together so that you have a single place you can go to update them. It's also convenient for inheritance since you can make new form schemas that build on existing ones. If you put your forms in a *models/form.py* file, you can easily use them throughout your controllers as *model.form.EmailForm* in the case shown.

Our form actually has two fields, an email text field and a submit button. If extra fields are submitted FormEncode's default behavior is to consider the form invalid so we specify *allow_extra_fields = True*. Since we don't want to use the values of the extra fields we also specify *filter_extra_fields = True*. The final line specifies that the email field should be validated with an *Email()* validator. In creating the validator we also specify *not_empty=True* so that the email field will require input.

Pylons comes with an easy to use *validate* decorator, if you wish to use it import it in your *lib/base.py* like this:

```
# other imports

from pylons.decorators import validate
```

Using it in your controller is pretty straight-forward:

```
# in the controller

def form(self):
    return render('/form.mako')

@validate(schema=EmailForm(), form='form')
def email(self):
    return 'Your email is: %s' % self.form_result.get('email')
```

Validation only occurs on POST requests so we need to alter our form definition so that the method is a POST:

```
${h.form(h.url(action='email'), method='post')}
```

If validation is successful, the valid result dict will be saved as *self.form_result* so it can be used in the action. Otherwise, the action will be re-run as if it was a GET request to the controller action specified in *form*, and the output will be filled by FormEncode's htmlfill to fill in the form field errors. For simple cases this is really handy because it also avoids having to write code in your templates to display error messages if they are present.

This does exactly the same thing as the example above but works with the original form definition and in fact will work with any HTML form regardless of how it is generated because the validate decorator uses *formencode.htmlfill* to find HTML fields and replace them with the values were originally submitted.

Note: Python 2.3 doesn't support decorators so rather than using the *@validate()* syntax you need to put *email = validate(schema=EmailForm(), form='form')(email)* after the email function's declaration.

10.5.2 Validation the Long Way

The *validate* decorator covers up a bit of work, and depending on your needs it's possible you could need direct access to FormEncode abilities it smoothes over.

Here's the longer way to use the *EmailForm* schema:

```
# in the controller

def email(self):
    schema = EmailForm()
    try:
        form_result = schema.to_python(request.params)
    except formencode.validators.Invalid, error:
        return 'Invalid: %s' % error
    else:
        return 'Your email is: %s' % form_result.get('email')
```

If the values entered are valid, the schema's *to_python()* method returns a dictionary of the validated and coerced *form_result*. This means that you can guarantee that the *form_result* dictionary contains values that are valid and correct Python objects for the data types desired.

In this case the email address is a string so *request.params['email']* happens to be the same as *form_result['email']*. If our form contained a field for age in years and we had used a *formen-*

code.validators.Int() validator, the value in *form_result* for the age would also be the correct type; in this case a Python integer.

FormEncode comes with a useful set of validators but you can also easily create your own. If you do create your own validators you will find it very useful that all FormEncode schemas' *.to_python()* methods take a second argument named *state*. This means you can pass the Pylons *c* object into your validators so that you can set any variables that your validators need in order to validate a particular field as an attribute of the *c* object. It can then be passed as the *c* object to the schema as follows:

```
c.domain = 'example.com'
form_result = schema.to_python(request.params, c)
```

The schema passes *c* to each validator in turn so that you can do things like this:

```
class SimpleEmail(formencode.validators.Email):
    def _to_python(self, value, c):
        if not value.endswith(c.domain):
            raise formencode.validators.Invalid(
                'Email addresses must end in: %s' % \
                    c.domain, value, c)
        return formencode.validators.Email._to_python(self, value, c)
```

For this to work, make sure to change the *EmailForm* schema you've defined to use the new *SimpleEmail* validator. In other words,

```
email = formencode.validators.Email(not_empty=True)
# becomes:
email = SimpleEmail(not_empty=True)
```

In reality the invalid error message we get if we don't enter a valid email address isn't very useful. We really want to be able to redisplay the form with the value entered and the error message produced. Replace the line:

```
return 'Invalid: %s' % error
```

with the lines:

```
c.form_result = error.value
c.form_errors = error.error_dict or {}
return render('/form.mako')
```

Now we will need to make some tweaks to *form.mako*. Make it look like this:

```
${h.form(h.url(action='email'), method='get')}

% if c.form_errors:
<h2>Please correct the errors</h2>
% else:
<h2>Enter Email Address</h2>
% endif

% if c.form_errors:
Email Address: ${h.text_field('email', value=c.form_result['email'] or '')}
<p>${c.form_errors['email']}</p>
% else:
Email Address: ${h.text_field('email')}
% endif

${h.submit('Submit')}
${h.end_form()}
```

Now when the form is invalid the *form.mako* template is re-rendered with the error messages.

10.6 Other Form Tools

If you are going to be creating a lot of forms you may wish to consider using FormBuild to help create your forms. To use it you create a custom Form object and use that object to build all your forms. You can then use the API to modify all aspects of the generation and use of all forms built with your custom Form by modifying its definition without any need to change the form templates.

Here is an one example of how you might use it in a controller to handle a form submission:

```
# in the controller

def form(self):
    results, errors, response = formbuild.handle(
        schema=Schema(), # Your FormEncode schema for the form
                         # to be validated
        template='form.mako', # The template containg the code
                              # that builds your form
        form=Form # The FormBuild Form definition you wish to use
    )
    if response:
        # The form validation failed so re-display
        # the form with the auto-generted response
        # containing submitted values and errors or
        # do something with the errors
        return response
    else:
        # The form validated, do something useful with results.
        ...
```

Full documentation of all features is available in the FormBuild manual which you should read before looking at Using FormBuild in Pylons

Looking forward it is likely Pylons will soon be able to use the TurboGears widgets system which will probably become the recommended way to build forms in Pylons.

CHAPTER

ELEVEN

INTERNATIONALIZATION AND LOCALIZATION

11.1 Introduction

Internationalization and localization are means of adapting software for non-native environments, especially for other nations and cultures.

Parts of an application which might need to be localized might include:

- Language
- Date/time format
- Formatting of numbers e.g. decimal points, positioning of separators, character used as separator
- Time zones (UTC in internationalized environments)
- Currency
- Weights and measures

The distinction between internationalization and localization is subtle but important. Internationalization is the adaptation of products for potential use virtually everywhere, while localization is the addition of special features for use in a specific locale.

For example, in terms of language used in software, internationalization is the process of marking up all strings that might need to be translated whilst localization is the process of producing translations for a particular locale.

Pylons provides built-in support to enable you to internationalize language but leaves you to handle any other aspects of internationalization which might be appropriate to your application.

Note: Internationalization is often abbreviated as I18N (or i18n or I18n) where the number 18 refers to the number of letters omitted. Localization is often abbreviated L10n or l10n in the same manner. These abbreviations also avoid picking one spelling (internationalisation vs. internationalization, etc.) over the other.

In order to represent characters from multiple languages, you will need to utilize Unicode. This document assumes you have read the *Understanding Unicode*.

By now you should have a good idea of what Unicode is, how to use it in Python and which areas of you application need to pay specific attention to decoding and encoding Unicode data.

This final section will look at the issue of making your application work with multiple languages.

Pylons uses the Python gettext module for internationalization. It is based off the GNU gettext API.

11.2 Getting Started

Everywhere in your code where you want strings to be available in different languages you wrap them in the `_()` function. There are also a number of other translation functions which are documented in the API reference at http://pylonshq.com/docs/module-pylons.i18n.translation.html

---

Note: The `_()` function is a reference to the `ugettext()` function. `_()` is a convention for marking text to be translated and saves on keystrokes. `ugettext()` is the Unicode version of `gettext()`; it returns unicode strings.

In our example we want the string `'Hello'` to appear in three different languages: English, French and Spanish. We also want to display the word `'Hello'` in the default language. We'll then go on to use some plural words too.

Lets call our project `translate_demo`:

```
$ paster create -t pylons translate_demo
```

Now lets add a friendly controller that says hello:

```
$ cd translate_demo
$ paster controller hello
```

Edit `controllers/hello.py` to make use of the `_()` function everywhere where the string `Hello` appears:

```
import logging

from pylons.i18n import get_lang, set_lang

from translate_demo.lib.base import *

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        response.write('Default: %s<br />' % _('Hello'))
        for lang in ['fr','en','es']:
            set_lang(lang)
        response.write("%s: %s<br />" % (get_lang(), _('Hello')))
```

When writing wrapping strings in the gettext functions, it is important not to piece sentences together manually; certain languages might need to invert the grammars. Don't do this:

```
# BAD!
msg = _("He told her ")
msg += _("not to go outside.")
```

but this is perfectly acceptable:

```
# GOOD
msg = _("He told her not to go outside")
```

The controller has now been internationalized, but it will raise a `LanguageError` until we have setup the alternative language catalogs.

GNU gettext use three types of files in the translation framework.

11.2.1 POT (Portable Object Template) files

The first step in the localization process. A program is used to search through your project's source code and pick out every string passed to one of the translation functions, such as `_()`. This list is put together in a specially-formatted template file that will form the basis of all translations. This is the `.pot` file.

11.2.2 PO (Portable Object) files

The second step in the localization process. Using the POT file as a template, the list of messages are translated and saved as a `.po` file.

11.2.3 MO (Machine Object) files

The final step in the localization process. The PO file is run through a program that turns it into an optimized machine-readable binary file, which is the `.mo` file. Compiling the translations to machine code makes the localized program much faster in retrieving the translations while it is running.

GNU gettext provides a suite of command line programs for extracting messages from source code and working with the associated gettext catalogs. The Babel project provides pure Python alternative versions of these tools. Unlike the GNU gettext tool *xgettext*, Babel supports extracting translatable strings from Python templating languages (currently Mako and Genshi).

11.3 Using Babel

To use Babel, you must first install it via easy_install. Run the command:

```
$ easy_install Babel
```

Pylons (as of 0.9.6) includes some sane defaults for Babel's distutils commands in the setup.cfg file.

It also includes an extraction method mapping in the setup.py file. It is commented out by default, to avoid distutils warning about it being an unrecognized option when Babel is not installed. These lines should be uncommented before proceeding with the rest of this walk through:

```
message_extractors = {'translate_demo': [
        ('**.py', 'python', None),
        ('templates/**.mako', 'mako', None),
        ('public/**', 'ignore', None)]},
```

We'll use Babel to extract messages to a `.pot` file in your project's `i18n` directory. First, the directory needs to be created. Don't forget to add it to your revision control system if one is in use:

```
$ cd translate_demo
$ mkdir translate_demo/i18n
$ svn add translate_demo/i18n
```

Next we can extract all messages from the project with the following command:

```
$ python setup.py extract_messages
running extract_messages
extracting messages from translate_demo/__init__.py
extracting messages from translate_demo/websetup.py
...
extracting messages from translate_demo/tests/functional/test_hello.py
writing PO template file to translate_demo/i18n/translate_demo.pot
```

This will create a `.pot` file in the `i18n` directory that looks something like this:

```
# Translations template for translate_demo.
# Copyright (C) 2007 ORGANIZATION
# This file is distributed under the same license as the translate_demo project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2007.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: translate_demo 0.0.0\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2007-08-02 18:01-0700\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 0.9dev-r215\n"

#: translate_demo/controllers/hello.py:10 translate_demo/controllers/hello.py:13
msgid "Hello"
msgstr ""
```

The `.pot` details that appear here can be customized via the `extract_messages` configuration in your project's `setup.cfg` (See the Babel Command-Line Interface Documentation for all configuration options).

Next, we'll initialize a catalog (`.po` file) for the Spanish language:

```
$ python setup.py init_catalog -l es
running init_catalog
creating catalog 'translate_demo/i18n/es/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
```

Then we can edit the last line of the new Spanish `.po` file to add a translation of `"Hello"`:

```
msgid "Hello"
msgstr "¡Hola!"
```

Finally, to utilize these translations in our application, we need to compile the `.po` file to a `.mo` file:

```
$ python setup.py compile_catalog
running compile_catalog
1 of 1 messages (100%) translated in 'translate_demo/i18n/es/LC_MESSAGES/translate_demo.po'
```

```
compiling catalog 'translate_demo/i18n/es/LC_MESSAGES/translate_demo.po' to
'translate_demo/i18n/es/LC_MESSAGES/translate_demo.mo'
```

We can also use the `update_catalog` command to merge new messages from the `.pot` to the `.po` files. For example, if we later added the following line of code to the end of HelloController's index method:

```
response.write('Goodbye: %s' % _('Goodbye'))
```

We'd then need to re-extract the messages from the project, then run the `update_catalog` command:

```
$ python setup.py extract_messages
running extract_messages
extracting messages from translate_demo/__init__.py
extracting messages from translate_demo/websetup.py
...
extracting messages from translate_demo/tests/functional/test_hello.py
writing PO template file to translate_demo/i18n/translate_demo.pot
$ python setup.py update_catalog
running update_catalog
updating catalog 'translate_demo/i18n/es/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
```

We'd then edit our catalog to add a translation for "Goodbye", and recompile the `.po` file as we did above.

For more information, see the Babel documentation and the GNU Gettext Manual.

11.4 Back To Work

Next we'll need to repeat the process of creating a `.mo` file for the `en` and `fr` locales:

```
$ python setup.py init_catalog -l en
running init_catalog
creating catalog 'translate_demo/i18n/en/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
$ python setup.py init_catalog -l fr
running init_catalog
creating catalog 'translate_demo/i18n/fr/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
```

Modify the last line of the `fr` catalog to look like this:

```
#: translate_demo/controllers/hello.py:10 translate_demo/controllers/hello.py:13
msgid "Hello"
msgstr "Bonjour"
```

Since our original messages are already in English, the `en` catalog can stay blank; gettext will fallback to the original.

Once you've edited these new `.po` files and compiled them to `.mo` files, you'll end up with an `i18n` directory containing:

```
i18n/translate_demo.pot
i18n/en/LC_MESSAGES/translate_demo.po
i18n/en/LC_MESSAGES/translate_demo.mo
i18n/es/LC_MESSAGES/translate_demo.po
i18n/es/LC_MESSAGES/translate_demo.mo
i18n/fr/LC_MESSAGES/translate_demo.po
i18n/fr/LC_MESSAGES/translate_demo.mo
```

11.5 Testing the Application

Start the server with the following command:

```
$ paster serve --reload development.ini
```

Test your controller by visiting http://localhost:5000/hello. You should see the following output:

```
Default: Hello
fr: Bonjour
en: Hello
es: ¡Hola!
```

You can now set the language used in a controller on the fly.

For example this could be used to allow a user to set which language they wanted your application to work in. You could save the value to the session object:

```
session['lang'] = 'en'
session.save()
```

then on each controller call the language to be used could be read from the session and set in your controller's `__before__()` method so that the pages remained in the same language that was previously set:

```
def __before__(self):
    if 'lang' in session:
        set_lang(session['lang'])
```

Pylons also supports defining the default language to be used in the configuration file. Set a `lang` variable to the desired default language in your `development.ini` file, and Pylons will automatically call `set_lang` with that language at the beginning of every request.

E.g. to set the default language to Spanish, you would add `lang = es` to your `development.ini`:

```
[app:main]
use = egg:translate_demo
lang = es
```

If you are running the server with the `--reload` option the server will automatically restart if you change the `development.ini` file. Otherwise restart the server manually and the output would this time be as follows:

```
Default: ¡Hola!
fr: Bonjour
en: Hello
es: ¡Hola!
```

11.6 Fallback Languages

If your code calls `_()` with a string that doesn't exist at all in your language catalog, the string passed to `_()` is returned instead.

Modify the last line of the hello controller to look like this:

```
response.write("%s %s, %s" % (_('Hello'), _('World'), _('Hi!')))
```

Warning: Of course, in real life breaking up sentences in this way is very dangerous because some grammars might require the order of the words to be different.

If you run the example again the output will be:

```
Default: ¡Hola!
fr: Bonjour World!
en: Hello World!
es: ¡Hola! World!
```

This is because we never provided a translation for the string `'World!'` so the string itself is used.

Pylons also provides a mechanism for fallback languages, so that you can specify other languages to be used if the word is omitted from the main language's catalog.

In this example we choose `fr` as the main language but `es` as a fallback:

```
import logging

from pylons.i18n import set_lang

from translate_demo.lib.base import *

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        set_lang(['fr', 'es'])
        return "%s %s, %s" % (_('Hello'), _('World'), _('Hi!'))
```

If `Hello` is in the `fr .mo` file as `Bonjour`, `World` is only in `es` as `Mundo` and none of the catalogs contain `Hi!`, you'll get the multilingual message: `Bonjour Mundo, Hi!`. This is a combination of the French, Spanish and original (English in this case, as defined in our source code) words.

You can also add fallback languages after calling `set_lang` via the `pylons.i18n.add_fallback` function. Translations will be tested in the order you add them.

Note: Fallbacks are reset after calling `set_lang(lang)` – that is, fallbacks are associated with the currently selected language.

One case where using fallbacks in this way is particularly useful is when you wish to display content based on the languages requested by the browser in the `HTTP_ACCEPT_LANGUAGE` header. Typically the browser may submit a number of languages so it is useful to be add fallbacks in the order specified by the browser so that you always try to display words in the language of preference and search the other languages in order if a translation cannot be found. The languages defined in the `HTTP_ACCEPT_LANGUAGE` header are available in Pylons as `request.languages` and can be used like this:

```
for lang in request.languages:
    add_fallback(lang)
```

11.7 Translations Within Templates

You can also use the `_()` function within templates in exactly the same way you do in code. For example, in a Mako template:

```
${_('Hello')}
```

would produce the string `'Hello'` in the language you had set.

Babel currently supports extracting gettext messages from Mako and Genshi templates. The Mako extractor also provides support for translator comments. Babel can be extended to extract messages from other sources via a custom extraction method plugin.

Pylons (as of 0.9.6) automatically configures a Babel extraction mapping for your Python source code and Mako templates. This is defined in your project's setup.py file:

```
message_extractors = {'translate_demo': [
        ('**.py', 'python', None),
        ('templates/**.mako', 'mako', None),
        ('public/**', 'ignore', None)]},
```

For a project using Genshi instead of Mako, the Mako line might be replaced with:

```
('templates/**.html, 'genshi', None),
```

See Babel's documentation on Message Extraction for more information.

11.8 Lazy Translations

Occasionally you might come across a situation when you need to translate a string when it is accessed, not when the `_()` or other functions are called.

Consider this example:

```
import logging

from pylons.i18n import get_lang, set_lang

from translate_demo.lib.base import *

log = logging.getLogger(__name__)

text = _('Hello')

class HelloController(BaseController):

    def index(self):
        response.write('Default: %s<br />' % _('Hello'))
        for lang in ['fr','en','es']:
            set_lang(lang)
        response.write("%s: %s<br />" % (get_lang(), _('Hello')))
        response.write('Text: %s<br />' % text)
```

If we run this we get the following output:

```
Default: Hello
['fr']: Bonjour
['en']: Good morning
['es']: Hola
Text: Hello
```

This is because the function `_('Hello')` just after the imports is called when the default language is `en` so the variable `text` gets the value of the English translation even though when the string was used the

default language was Spanish.

The rule of thumb in these situations is to try to avoid using the translation functions in situations where they are not executed on each request. For situations where this isn't possible, perhaps because you are working with legacy code or with a library which doesn't support internationalization, you need to use lazy translations.

If we modify the above example so that the import statements and assignment to `text` look like this:

```
from pylons.i18n import get_lang, lazy_gettext, set_lang

from helloworld.lib.base import *

log = logging.getLogger(__name__)

text = lazy_gettext('Hello')
```

then we get the output we expected:

```
Default: Hello
['fr']: Bonjour
['en']: Good morning
['es']: Hola
Text: Hola
```

There are lazy versions of all the standard Pylons translation functions.

There is one drawback to be aware of when using the lazy translation functions: they are not actually strings. This means that if our example had used the following code it would have failed with an error `cannot concatenate 'str' and 'LazyString' objects`:

```
response.write('Text: ' + text + '<br />')
```

For this reason you should only use the lazy translations where absolutely necessary and should always ensure they are converted to strings by calling `str()` or `repr()` before they are used in operations with real strings.

11.9 Producing a Python Egg

Finally you can produce an egg of your project which includes the translation files like this:

```
$ python setup.py bdist_egg
```

The `setup.py` automatically includes the `.mo` language catalogs your application needs so that your application can be distributed as an egg. This is done with the following line in your `setup.py` file:

```
package_data={'translate_demo': ['i18n/*/LC_MESSAGES/*.mo']},
```

11.10 Plural Forms

Pylons also provides the `ungettext()` function. It's designed for internationalizing plural words, and can be used as follows:

```
ungettext('There is %(num)d file here', 'There are %(num)d files here',
          n) % {'num': n}
```

Plural forms have a different type of entry in `.pot/.po` files, as described in The Format of PO Files in GNU Gettext's Manual:

```
#: translate_demo/controllers/hello.py:12
#, python-format
msgid "There is %(num)d file here"
msgid_plural "There are %(num)d files here"
msgstr[0] ""
msgstr[1] ""
```

One thing to keep in mind is that other languages don't have the same plural forms as English. While English only has 2 plural forms, singular and plural, Slovenian has 4! That means that you *must* use ugettext for proper pluralization. Specifically, the following will not work:

```
# BAD!
if n == 1:
    msg = _("There was no dog.")
else:
    msg = _("There were no dogs.")
```

11.11 Summary

This document only covers the basics of internationalizing and localizing a web application.

GNU Gettext is an extensive library, and the GNU Gettext Manual is highly recommended for more information.

Babel also provides support for interfacing to the CLDR (Common Locale Data Repository), providing access to various locale display names, localized number and date formatting, etc.

You should also be able to internationalize and then localize your application using Pylons' support for GNU gettext.

11.12 Further Reading

http://en.wikipedia.org/wiki/Internationalization

Please feel free to report any mistakes to the Pylons mailing list or to the author. Any corrections or clarifications would be gratefully received.

Note: This is a work in progress. We hope the internationalization, localization and Unicode support in Pylons is now robust and flexible but we would appreciate hearing about any issues we have. Just drop a line to the pylons-discuss mailing list on Google Groups.

11.13 babel.core – Babel core classes

11.13.1 Module Contents

11.14 babel.localedata — Babel locale data

11.15 babel.dates – Babel date classes

11.15.1 Module Contents

11.16 babel.numbers – Babel number classes

11.16.1 Module Contents

CHAPTER

TWELVE

SESSIONS

12.1 Sessions

Note: The session code is due an extensive rewrite. It uses the Caching container API in Beaker which is optimized for use patterns that are more common in caching (infrequent updates / frequent reads). Unlike caching, a session is a single load, then a single save and multiple simultaneous writes to the same session occur only rarely. In consequence, the excessive but necessary locking that the cache interface currently performs is just a waste of performance where sessions are concerned.

12.2 Session Objects

12.2.1 SessionObject

This session proxy / lazy creator object handles access to the real session object. If the session hasn't been used before a session object will automatically be created and set up. Using a proxy in this fashion to handle access to the real session object avoids creating and loading the session from persistent store unless it is actually used during the request.

12.2.2 CookieSession

Pure cookie-based session. The options recognized when using cookie-based sessions are slightly more restricted than general sessions.

- **`key`** The name the cookie should be set to.
- **`timeout`** How long session data is considered valid. This is used regardless of the cookie being present or not to determine whether session data is still valid.
- **`encrypt_key`** The key to use for the session encryption, if not provided the session will not be encrypted.
- **`validate_key`** The key used to sign the encrypted session
- **`cookie_domain`** Domain to use for the cookie.
- **`secure`** Whether or not the cookie should only be sent over SSL.

12.3 Beaker

```
beaker.session.key = wiki
beaker.session.secret = ${app_instance_secret}
```

Pylons comes with caching middleware enabled that is part of the same package that provides the session handling, Beaker. Beaker supports several different types of cache back-end: memory, filesystem, memcached and database. The supported database packages are: SQLite, SQLAlchemy and Google BigTable.

Beaker's cache and session options are configured via a dictionary.

Note: When used with the Paste package, all Beaker options should be prefixed with `beaker.` so that Beaker can discriminate its options from other application configuration options.

12.3.1 General Config Options

Config options should be prefixed with either `session.` or `cache.`

data_dir

Accepts: string *Default:* None

The data directory where cache data will be stored. If this argument is not present, the regular data_dir parameter is used, with the path "./sessions" appended to it.

type

Accepts: string *Default:* dbm

Type of storage used for the session, current types are "dbm", "file", "memcached", "database", and "memory". The storage uses the Container API that is also used by the cache system.

When using dbm files, each user's session is stored in its own dbm file, via the class :class"*beaker.container.DBMNamespaceManager* class.

When using 'database' or 'memcached', additional configuration options are required as documented in the appropriate section below.

For sessions only, there is an additional choice of a "cookie" type, which requires the Sessions "secret" option to be set as well.

12.3.2 Database Configuration

When the type is set to 'database', the following additional options can be used.

url (*required*)

Accepts: string (formatted as required for an SQLAlchemy db uri) *Default:* None

The database URI as formatted for SQLAlchemy to use for the database. The appropriate database packages for the database must also be installed.

table_name

Accepts: string *Default:* beaker_cache

Table name to use for beaker's storage.

optimistic

Accepts: boolean *Default:* False

Use optimistic session locking, note that this will result in an select when updating a cache value to compare version numbers.

sa_opts (*Only for SQLAlchemy 0.3*)

Accepts: dict *Default:* None

A dictionary of values to use that are passed directly to SQLAlchemy's engine. Note that this is only applicable for SQLAlchemy 0.3.

sa.*

Accepts: Valid SQLAlchemy 0.4 database options *Default:* None

When using SQLAlchemy 0.4 and above, all options prefixed with `sa.` are passed to the SQLAlchemy database engine. Common parameters are `pool_size`, `pool_recycle`, etc.

12.3.3 Memcached Options

url (required)

Accepts: string *Default:* None

The url should be a single IP address, or list of semi-colon separated IP addresses that should be used for memcached.

Beaker can use either py-memcached or cmemcache to communicate with memcached, but it should be noted that cmemcache can cause Python to segfault should memcached become unreachable.

12.3.4 Session Options

cookie_expires

Accepts: boolean, datetime, timedelta *Default:* True

The expiration time to use on the session cookie. Defaults to "True" which means, don't specify any expiration time (the cookie will expire when the browser is closed). A value of "False" means, never expire (specifies the maximum date that can be stored in a datetime object and uses that). The value can also be a {{datetime.timedelta()}} object which will be added to the current date and time, or a {{datetime.datetime()}} object.

cookie_domain

Accepts: string *Default:* The entire domain name being used, including sub-domain, etc.

By default, Beaker's sessions are set to the cookie domain of the entire hostname. For sub-domains, this should be set to the top domain the cookie should be valid for.

id

Accepts: string *Default:* None

Session id for this session. When using sessions with cookies, this parameter is not needed as the session automatically creates, writes and retrieves the value from the request. When using a URL-based method for the session, the id should be retreived from the id data member when the session is first created, and then used in writing new URLs.

key

Accepts: string *Default:* beaker_session_id

The key that will be used as a cookie key to identify sessions. Changing this could allow several different applications to have different sessions underneath the same hostname.

secret

Accepts: string *Default:* None

Secret key to enable encrypted session ids. When non-None, the session ids are generated with an MD5-signature created against this value.

When used with the "cookie" Session type, the secret is used for encrypting the contents of the cookie, and should be a reasonably secure randomly generated string of characters no more than 54 characters.

timeout

Accepts: integer *Default:* None

Time in seconds before the session times out. A timeout occurs when the session has not been loaded for more than timeout seconds.

12.3.5 Session Options (For use with cookie-based Sessions)

encrypt_key

Accepts: string *Default:* None

The key to use for the session encryption, if not provided the session will not be encrypted. This will only work if a strong hash scheme is available, such as pycryptopp's or Python 2.5's hashlib.sha256.

validate_key

Accepts: string *Default:* None

The key used to sign the encrypted session, this is used instead of a secret option.

12.4 Custom and caching middleware

Care should be taken when deciding in which layer to place custom middleware. In most cases middleware should be placed between the Pylons WSGI application instantiation and the Routes middleware; however, if the middleware should run *before* the session object or routing is handled:

```
# Routing/Session/Cache Middleware
app = RoutesMiddleware(app, config['routes.map'])
app = SessionMiddleware(app, config)

# MyMiddleware can only see the cache object, nothing *above* here
app = MyMiddleware(app)

app = CacheMiddleware(app, config)
```

Some of the Pylons middleware layers such as the `Session`, `Routes`, and `Cache` middleware, only add objects to the *environ* dict, or add HTTP headers to the response (the Session middleware for example adds the session cookie header). Others, such as the `Status Code Redirect`, and the `Error Handler` may fully intercept the request entirely, and change how its responded to.

12.5 Bulk deletion of expired db-held sessions

The db schema for Session stores a "last accessed time" for each session. This enables bulk deletion of expired sessions through the use of a simple SQL command, run every day, that clears those sessions which have a "last accessed" timestamp > 2 days, or whatever is required.

12.6 Using *Session* in Internationalization

How to set the language used in a controller on the fly.

For example this could be used to allow a user to set which language they wanted your application to work in. Save the value to the session object:

```
session['lang'] = 'en'
session.save()
```

then on each controller call the language to be used could be read from the session and set in the controller's `__before__()` method so that the pages remained in the same language that was previously set:

```
def __before__(self):
    if 'lang' in session:
        set_lang(session['lang'])
```

12.7 Using *Session* in Secure Forms

Authorization tokens are stored in the client's session. The web app can then verify the request's submitted authorization token with the value in the client's session.

This ensures the request came from the originating page. See the wikipedia entry for Cross-site request forgery for more information.

Pylons provides an `authenticate_form` decorator that does this verfication on the behalf of controllers.

These helpers depend on Pylons' `session` object. Most of them can be easily ported to another framework by changing the API calls.

12.8 Hacking the session for no cookies

(From a paste #441 baked by Ben Bangert)

Set the session to not use cookies in the dev.ini file

```
beaker.session.use_cookies = False
```

with this as the *mode d'emploi* in the controller action

```
from beaker.session import Session as BeakerSession

# Get the actual session object through the global proxy
real_session = session._get_current_obj()

# Duplicate the session init options to avoid screwing up other sessions in
# other threads
params = real_session.__dict__['_params']

# Now set the id param used to make a session to our session maker,
# if id is None, a new id will be made automatically
params['id'] = find_id_func()
real_session.__dict__['_sess'] = BeakerSession({}, **params)

# Now we can use the session as usual
session['fred'] = 42
session.save()

# At the end, we need to see if the session was used and handle its id
if session.is_new:
    # do something with session.id to make sure its around next time
    pass
```

12.9 Using middleware (Beaker) with a composite app

How to allow called WSGI apps to share a common session management utility.

(From a paste #616 baked by Mark Luffel)

```
# Here's an example of configuring multiple apps to use a common
# middleware filter
# The [app:home] section is a standard pylons app
# The ''/servicebroker'' and ''/proxy'' apps both want to be able
# to use the same session management

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000

[filter-app:main]
use = egg:Beaker#beaker_session
```

```
next = sessioned
beaker.session.key = my_project_key
beaker.session.secret = i_wear_two_layers_of_socks

[composite:sessioned]
use = egg:Paste#urlmap
/ = home
/servicebroker = servicebroker
/proxy = cross_domain_proxy

[app:servicebroker]
use = egg:Appcelerator#service_broker

[app:cross_domain_proxy]
use = egg:Appcelerator#cross_domain_proxy

[app:home]
use = egg:my_project
full_stack = true
cache_dir = %(here)s/data
```

12.10 storing SA mapped objects in Beaker sessions

Taken from pylons-discuss Google group discussion:

```
> I wouldn't expect a SA object to be serializable.  It just doesn't
> make sense to me.  I don't even want to think about complications with
> the database and ACID, nor do I want to consider the scalability
> concerns (the SA object should be tied to a particular SA session,
> right?).
```

SA objects are serializable (as long as you aren't using `assign_mapper()`, which can complicate things unless you define a custom `__getstate__()` method).

The error above is because the entity is not being detached from its original session. If you are going to serialize, you have to manually shuttle the object to and from the appropriate sessions.

Three ways to get an object out of serialization and back into an SA Session are:

1. A mapped class that has a `__getstate__()` which only copies desired properties and won't copy SA session pointers:

```
beaker.put(key, obj)
...
obj = beaker.get(key)
Session.add(obj)
```

2. A regular old mapped class. Add an `expunge()` step.

```
Session.expunge(obj)
beaker.put(key, obj)
...
obj = beaker.get(key)
Session.add(obj)
```

3. Don't worry about `__getstate__()` or `expunge()` on the original object, use `merge()`. This is "cleaner" than the `expunge()` method shown above but will usually force a load of the object from

the database and therefore is not necessarily as "efficient", also it copies the state of the given object to the target object which may be error-prone.

```
beaker.put(key, obj)
...
obj = beaker.get(key)
obj = Session.merge(obj)
```

CHAPTER

THIRTEEN

CACHING

Inevitably, there will be occasions during applications development or deployment when some task is revealed to be taking a significant amount of time to complete. When this occurs, the best way to speed things up is with *caching*.

Pylons comes with caching middleware enabled that is part of the same package that provides the session handling, Beaker. Beaker supports a variety of caching backends: memory-based, filesystem-based and the specialised *memcached* library.

There are several ways to cache data under Pylons, depending on where the slowdown is occurring:

- Browser-side Caching - HTTP/1.1 supports the *ETag* caching system that allows the browser to use its own cache instead of requiring regeneration of the entire page. ETag-based caching avoids repeated generation of content but if the browser has never seen the page before, the page will still be generated. Therefore using ETag caching in conjunction with one of the other types of caching listed here will achieve optimal throughput and avoid unnecessary calls on resource-intensive operations.

Note: the latter only helps if the entire page can be cached.

- Controllers - The *cache* object can be imported in controllers used for caching anything in Python that can be pickled.
- Templates - The results of an entire rendered template can be cached using the 3 cache keyword arguments to the render calls. These render commands can also be used inside templates.
- Mako/Myghty Templates - Built-in caching options are available for both Mako and Myghty template engines. They allow fine-grained caching of only certain sections of the template as well as caching of the entire template.

The two primary concepts to bear in mind when caching are i) caches have a *namespace* and ii) caches can have *keys* under that namespace. The reason for this is that, for a single template, there might be multiple versions of the template each requiring its own cached version. The keys in the namespace are the `version` and the name of the template is the `namespace`. **Both of these values must be Python strings.**

In templates, the cache `namespace` will automatically be set to the name of the template being rendered. Nothing else is required for basic caching, unless the developer wishes to control for how long the template is cached and/or maintain caches of multiple versions of the template.

See Also:

Stephen Pierzchala's Caching for Performance (stephen@pierzchala.com)

13.1 Using the Cache object

Inside the controller, the *cache* object needs to be imported before being used. If an action or block of code makes heavy use of resources or take a long time to complete, it can be convenient to cache the result. The *cache* object can cache any Python structure that can be pickled.

Consider an action where it is desirable to cache some code that does a time-consuming or resource-intensive lookup and returns an object that can be pickled (list, dict, tuple, etc.):

```
# Add to existing imports
from pylons import cache


# Under the controller class
def some_action(self, day):
    # hypothetical action that uses a 'day' variable as its key

    def expensive_function():
        # do something that takes a lot of cpu/resources
        return expensive_call()

    # Get a cache for a specific namespace, you can name it whatever
    # you want, in this case its 'my_function'
    mycache = cache.get_cache('my_function', type="memory")

    # Get the value, this will create the cache copy the first time
    # and any time it expires (in seconds, so 3600 = one hour)
    c.myvalue = mycache.get_value(key=day, createfunc=expensive_function,
                                  expiretime=3600)

    return render('/some/template.myt')
```

The *createfunc* option requires a callable object or a function which is then called by the cache whenever a value for the provided key is not in the cache, or has expired in the cache.

Because the *createfunc* is called with no arguments, the resource- or time-expensive function must correspondingly also not require any arguments.

13.1.1 Other Cache Options

The cache also supports the removal values from the cache, using the key(s) to identify the value(s) to be removed and it also supports clearing the cache completely, should it need to be reset.

```
# Clear the cache
mycache.clear()

# Remove a specific key
mycache.remove_value('some_key')
```

13.2 Using Cache keywords to *render*

> **Warning:** Needs to be extended to cover the specific render_* calls introduced in Pylons 0.9.7

All render <pylons.templating.render_mako() commmands have caching functionality built in. To use it, merely add the appropriate cache keyword to the render call.

```
class SampleController(BaseController):

    def index(self):
        # Cache the template for 10 mins
        return render('/index.myt', cache_expire=600)

    def show(self, id):
        # Cache this version of the template for 3 mins
        return render('/show.myt', cache_key=id, cache_expire=180)

    def feed(self):
        # Cache for 20 mins to memory
        return render('/feed.myt', cache_type='memory', cache_expire=1200)

    def home(self, user):
        # Cache this version of a page forever (until the cache dir
        # is cleaned)
        return render('/home.myt', cache_key=user, cache_expire='never')
```

13.3 Using the Cache Decorator

Pylons also provides the beaker_cache() decorator for caching in *pylons.cache* the results of a completed function call (memoizing).

The cache decorator takes the same cache arguments (minus their *cache_* prefix), as the *render* function does.

```
from pylons.decorators.cache import beaker_cache

class SampleController(BaseController):

    # Cache this controller action forever (until the cache dir is
    # cleaned)
    @beaker_cache()
    def home(self):
        c.data = expensive_call()
        return render('/home.myt')

    # Cache this controller action by its GET args for 10 mins to memory
    @beaker_cache(expire=600, type='memory', query_args=True)
    def show(self, id):
        c.data = expensive_call(id)
        return render('/show.myt')
```

By default the decorator uses a composite of all of the decorated function's arguments as the cache key. It can alternatively use a composite of the *request.GET* query args as the cache key when the *query_args* option is enabled.

The cache key can be further customized via the *key* argument.

13.4 Caching Arbitrary Functions

Arbitrary functions can use the `beaker_cache()` decorator, but should include an additional option. Since the decorator caches the *response* object, its unlikely the status code and headers for non-controller methods should be cached. To avoid caching that data, the cache_response keyword argument should be set to false.

```
from pylons.decorators.cache import beaker_cache

@beaker_cache(expire=600, cache_response=False)
def generate_data():
    # do expensive data generation
    return data
```

Warning: When caching arbitrary functions, the `query_args` argument should not be used since the result of arbitrary functions shouldn't depend on the request parameters.

13.5 ETag Caching

Caching via ETag involves sending the browser an ETag header so that it knows to save and possibly use a cached copy of the page from its own cache, instead of requesting the application to send a fresh copy.

Because the ETag cache relies on sending headers to the browser, it works in a slightly different manner to the other caching mechanisms described above.

The `etag_cache()` function will set the proper HTTP headers if the browser doesn't yet have a copy of the page. Otherwise, a 304 HTTP Exception will be thrown that is then caught by Paste middleware and turned into a proper 304 response to the browser. This will cause the browser to use its own locally-cached copy.

`etag_cache()` returns `Response` for legacy purposes (`Response` should be used directly instead).

ETag-based caching requires a single key which is sent in the ETag HTTP header back to the browser. The RFC specification for HTTP headers indicates that an ETag header merely needs to be a string. This value of this string does not need to be unique for every URL as the browser itself determines whether to use its own copy, this decision is based on the URL and the ETag key.

```
def my_action(self):
    etag_cache('somekey')
    return render('/show.myt', cache_expire=3600)
```

Or to change other aspects of the response:

```
def my_action(self):
    etag_cache('somekey')
    response.headers['content-type'] = 'text/plain'
    return render('/show.myt', cache_expire=3600)
```

Note: In this example that we are using template caching in addition to ETag caching. If a new visitor comes to the site, we avoid re-rendering the template if a cached copy exists and repeat hits to the page by that user will then trigger the ETag cache. This example also will never change the ETag key, so the browsers cache will always be used if it has one.

The frequency with which an ETag cache key is changed will depend on the web application and the developer's assessment of how often the browser should be prompted to fetch a fresh copy of the page.

Warning: Stolen from Philip Cooper's OpenVest wiki after which it was updated and edited ...

13.6 Inside the Beaker Cache

13.6.1 Caching

First lets start out with some **slow** function that we would like to cache. This function is not slow but it will show us when it was cached so we can see things are working as we expect:

```
import time
def slooow(myarg):
  # some slow database or template stuff here
  return "%s at %s" % (myarg,time.asctime())
```

When we have the cached function, multiple calls will tell us whether are seeing a cached or a new version.

13.6.2 DBMCache

The DBMCache stores (actually pickles) the response in a dbm style database.

What may not be obvious is that the are two levels of keys. They are essentially created as one for the function or template name (called the namespace) and one for the ''keys" within that (called the key). So for *Some_Function_name*, there is a cache created as one dbm file/database. As that function is called with different arguments, those arguments are keys within the dbm file. First lets create and populate a cache. This cache might be a cache for the function *Some_Function_name* called three times with three different arguments: *x, yy, and zzz*:

```
from beaker.cache import CacheManager
cm = CacheManager(type='dbm', data_dir='beaker.cache')
cache = cm.get_cache('Some_Function_name')
# the cache is setup but the dbm file is not created until needed
# so let's populate it with three values:
cache.get_value('x', createfunc=lambda: slooow('x'), expiretime=15)
cache.get_value('yy', createfunc=lambda: slooow('yy'), expiretime=15)
cache.get_value('zzz', createfunc=lambda: slooow('zzz'), expiretime=15)
```

Nothing much new yet. After getting the cache we can use the cache as per the Beaker Documentation.

```
import beaker.container as container
cc = container.ContainerContext()
nsm = cc.get_namespace_manager('Some_Function_name',
                               container.DBMContainer,data_dir='beaker.cache')
filename = nsm.file
```

Now we have the file name. The file name is a *sha* hash of a string which is a join of the container class name and the function name (used in the *get_cache* function call). It would return something like:

```
'beaker.cache/container_dbm/a/a7/a768f120e39d0248d3d2f23d15ee0a20be5226de.dbm'
```

With that file name you could look directly inside the cache database (but only for your education and debugging experience, **not** your cache interactions!)

```
## this file name can be used directly (for debug ONLY)
import anydbm
import pickle
db = anydbm.open(filename)
old_t, old_v = pickle.loads(db['zzz'])
```

The database only contains the old time and old value. Where did the expire time and the function to create/update the value go?. They never make it to the database. They reside in the *cache* object returned from *get_cache* call above.

Note that the createfunc, and expiretime values are stored during the first call to *get_value*. Subsequent calls with (say) a different expiry time will **not** update that value. This is a tricky part of the caching but perhaps is a good thing since different processes may have different policies in effect.

If there are difficulties with these values, remember that one call to `cache.clear()` resets everything.

13.6.3 Database Cache

Using the *ext:database* cache type.

```
from beaker.cache import CacheManager
#cm = CacheManager(type='dbm', data_dir='beaker.cache')
cm = CacheManager(type='ext:database',
                  url="sqlite:///beaker.cache/beaker.sqlite",
                  data_dir='beaker.cache')
cache = cm.get_cache('Some_Function_name')
# the cache is setup but the dbm file is not created until needed
# so let's populate it with three values:
cache.get_value('x', createfunc=lambda: slooow('x'), expiretime=15)
cache.get_value('yy', createfunc=lambda: slooow('yy'), expiretime=15)
cache.get_value('zzz', createfunc=lambda: slooow('zzz'), expiretime=15)
```

This is identical to the cache usage above with the only difference being the creation of the *CacheManager*. It is much easier to view the caches outside the beaker code (again for edification and debugging, not for api usage).

SQLite was used in this instance and the SQLite data file can be directly accessed uaing the SQLite command-line utility or the Firefox plug-in:

```
sqlite3 beaker.cache/beaker.sqlite
# from inside sqlite:
sqlite> .schema
CREATE TABLE beaker_cache (
        id INTEGER NOT NULL,
        namespace VARCHAR(255) NOT NULL,
        key VARCHAR(255) NOT NULL,
        value BLOB NOT NULL,
        PRIMARY KEY (id),
         UNIQUE (namespace, key)
);
select * from beaker_cache;
```

Warning: The data structure is different in Beaker 0.8 ...

```
cache = sa.Table(table_name, meta,
                 sa.Column('id', types.Integer, primary_key=True),
                 sa.Column('namespace', types.String(255), nullable=False),
```

```
                sa.Column('accessed', types.DateTime, nullable=False),
                sa.Column('created', types.DateTime, nullable=False),
                sa.Column('data', types.BLOB(), nullable=False),
                sa.UniqueConstraint('namespace')
)
```

It includes the access time but stores rows on a one-row-per-namespace basis, (storing a pickled dict) rather than one-row-per-namespace/key-combination. This is a more efficient approach when the problem is handling a large number of namespaces with limited keys — like sessions.

13.6.4 Memcached Cache

For large numbers of keys with expensive pre-key lookups memcached it the way to go.

If memcached is running on the the default port of 11211:

```
from beaker.cache import CacheManager
cm = CacheManager(type='ext:memcached', url='127.0.0.1:11211',
                  lock_dir='beaker.cache')
cache = cm.get_cache('Some_Function_name')
# the cache is setup but the dbm file is not created until needed
# so let's populate it with three values:
cache.get_value('x', createfunc=lambda: slooow('x'), expiretime=15)
cache.get_value('yy', createfunc=lambda: slooow('yy'), expiretime=15)
cache.get_value('zzz', createfunc=lambda: slooow('zzz'), expiretime=15)
```

CHAPTER

FOURTEEN

UNIT AND FUNCTIONAL TESTING

14.1 Unit Testing with `webtest`

Pylons provides powerful unit testing capabilities for your web application utilizing webtest to emulate requests to your web application. You can then ensure that the response was handled appropriately and that the controller set things up properly.

To run the test suite for your web application, Pylons utilizes the nose test runner/discovery package. Running `nosetests` in your project directory will run all the tests you create in the tests directory. If you don't have nose installed on your system, it can be installed via setuptools with:

```
$ easy_install -U nose
```

To avoid conflicts with your development setup, the tests use the *test.ini* configuration file when run. This means **you must configure any databases, etc. in your test.ini file or your tests will not be able to find the database configuration**.

> **Warning:** Nose can trigger errors during its attempt to search for doc tests since it will try and import all your modules one at a time *before* your app was loaded. This will cause files under models/ that rely on your app to be running, to fail.

Pylons 0.9.6.1 and later includes a plugin for nose that loads the app before the doctests scan your modules, allowing models to be doctested. You can use this option from the command line with nose:

```
nosetests --with-pylons=test.ini
```

Or by setting up a *[nosetests]* block in your setup.cfg:

```
[nosetests]
verbose=True
verbosity=2
with-pylons=test.ini
detailed-errors=1
with-doctest=True
```

Then just run:

```
python setup.py nosetests
```

to run the tests.

14.2 Example: Testing a Controller

First let's create a new project and controller for this example:

```
$ paster create -t pylons TestExample
$ cd TestExample
$ paster controller comments
```

You'll see that it creates two files when you create a controller. The stub controller, and a test for it under `testexample/tests/functional/`.

Modify the `testexample/controllers/comments.py` file so it looks like this:

```
from testexample.lib.base import *

class CommentsController(BaseController):

    def index(self):
        return 'Basic output'

    def sess(self):
        session['name'] = 'Joe Smith'
        session.save()
        return 'Saved a session'
```

Then write a basic set of tests to ensure that the controller actions are functioning properly, modify `testexample/tests/functional/test_comments.py` to match the following:

```
from testexample.tests import *

class TestCommentsController(TestController):
    def test_index(self):
        response = self.app.get(url(controller='/comments'))
        assert 'Basic output' in response

    def test_sess(self):
        response = self.app.get(url(controller='/comments', action='sess'))
        assert response.session['name'] == 'Joe Smith'
        assert 'Saved a session' in response
```

Run `nosetests` in your main project directory and you should see them all pass:

```
..
----------------------------------------------------------------------
Ran 2 tests in 2.999s

OK
```

Unfortunately, a plain assert does not provide detailed information about the results of an assertion should it fail, unless you specify it a second argument. For example, add the following test to the `test_sess` function:

```
assert response.session.has_key('address') == True
```

When you run `nosetests` you will get the following, not-very-helpful result:

```
.F
======================================================================
FAIL: test_sess (testexample.tests.functional.test_comments.TestCommentsController)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
File "~/TestExample/testexample/tests/functional/test_comments.py", line 12, in test_sess
assert response.session.has_key('address') == True
AssertionError:


----------------------------------------------------------------------
Ran 2 tests in 1.417s

FAILED (failures=1)
```

You can augment this result by doing the following:

```
assert response.session.has_key('address') == True, "address not found in session"
```

Which results in:

```
.F
======================================================================
FAIL: test_sess (testexample.tests.functional.test_comments.TestCommentsController)
----------------------------------------------------------------------
Traceback (most recent call last):
File "~/TestExample/testexample/tests/functional/test_comments.py", line 12, in test_sess
assert response.session.has_key('address') == True
AssertionError: address not found in session


----------------------------------------------------------------------
Ran 2 tests in 1.417s

FAILED (failures=1)
```

But detailing every assert statement could be time consuming. Our TestController subclasses the standard Python `unittest.TestCase` class, so we can use utilize its helper methods, such as `assertEqual`, that can automatically provide a more detailed AssertionError. The new test line looks like this:

```
self.assertEqual(response.session.has_key('address'), True)
```

Which provides the more useful failure message:

```
.F
======================================================================
FAIL: test_sess (testexample.tests.functional.test_comments.TestCommentsController)
----------------------------------------------------------------------
Traceback (most recent call last):
File "~/TestExample/testexample/tests/functional/test_comments.py", line 12, in test_sess
self.assertEqual(response.session.has_key('address'), True)
AssertionError: False != True
```

14.3 Testing Pylons Objects

Pylons will provide several additional attributes for the `webtest` `webtest.TestResponse` object that let you access various objects that were created during the web request:

`session` Session object

`req` Request object

c Object containing variables passed to templates

g Globals object

To use them, merely access the attributes of the response *after* you've used a get/post command:

```
response = app.get('/some/url')
assert response.session['var'] == 4
assert 'REQUEST_METHOD' in response.req.environ
```

Note: The `response` object already has a TestRequest object assigned to it, therefore Pylons assigns its `request` object to the response as `req`.

14.4 Testing Your Own Objects

WebTest's fixture testing allows you to designate your own objects that you'd like to access in your tests. This powerful functionality makes it easy to test the value of objects that are normally only retained for the duration of a single request.

Before making objects available for testing, its useful to know when your application is being tested. WebTest will provide an environ variable called `paste.testing` that you can test for the presence and truth of so that your application only populates the testing objects when it has to.

Populating the `webtest` response object with your objects is done by adding them to the environ dict under the key `paste.testing_variables`. Pylons creates this dict before calling your application, so testing for its existence and adding new values to it is recommended. All variables assigned to the `paste.testing_variables` dict will be available on the response object with the key being the attribute name.

Note: WebTest is an extracted stand-alone version of a Paste component called paste.fixture. For backwards compatibility, WebTest continues to honor the `paste.testing_variables` key in the environ.

Example:

```
# testexample/lib/base.py

from pylons import request
from pylons.controllers import WSGIController
from pylons.templating import render_mako as render

class BaseController(WSGIController):
    def __call__(self, environ, start_response):
        # Create a custom email object
        email = MyCustomEmailObj()
        email.name = 'Fred Smith'
        if 'paste.testing_variables' in request.environ:
            request.environ['paste.testing_variables']['email'] = email
        return WSGIController.__call__(self, environ, start_response)


# testexample/tests/functional/test_controller.py
from testexample.tests import *

class TestCommentsController(TestController):
    def test_index(self):
```

```
        response = self.app.get(url(controller='/'))
        assert response.email.name == 'Fred Smith'
```

See Also:

WebTest Documentation Documentation covering webtest and its usage

`WebTest Module docs` Module API reference for methods available for use when testing the application

14.5 Unit Testing

XXX: Describe unit testing an applications models, libraries

14.6 Functional Testing

XXX: Describe functional/integrated testing, WebTest

CHAPTER

FIFTEEN

TROUBLESHOOTING & DEBUGGING

15.1 Interactive debugging

Things break, and when they do, quickly pinpointing what went wrong and why makes a huge difference. By default, Pylons uses a customized version of Ian Bicking's EvalException middleware that also includes full Mako/Myghty Traceback information.

15.2 The Debugging Screen

The debugging screen has three tabs at the top:

`Traceback` Provides the raw exception trace with the interactive debugger

`Extra Data` Displays CGI, WSGI variables at the time of the exception, in addition to configuration information

`Template` Human friendly traceback for Mako or Myghty templates

Since Mako and Myghty compile their templates to Python modules, it can be difficult to accurately figure out what line of the template resulted in the error. The *Template* tab provides the full Mako or Myghty traceback which contains accurate line numbers for your templates, and where the error originated from. If your exception was triggered before a template was rendered, no Template information will be available in this section.

15.3 Example: Exploring the Traceback

Using the interactive debugger can also be useful to gain a deeper insight into objects present only during the web request like the `session` and `request` objects.

To trigger an error so that we can explore what's happening just raise an exception inside an action you're curious about. In this example, we'll raise an error in the action that's used to display the page you're reading this on. Here's what the docs controller looks like:

```
class DocsController(BaseController):
    def view(self, url):
        if request.path_info.endswith('docs'):
            redirect_to('/docs/')
        return render('/docs/' + url)
```

Since we want to explore the `session` and `request`, we'll need to bind them first. Here's what our action now looks like with the binding and raising an exception:

```
def view(self, url):
    raise "hi"
    if request.path_info.endswith('docs'):
        redirect_to('/docs/')
    return render('/docs/' + url)
```

Here's what exploring the Traceback from the above example looks like (Excerpt of the relevant portion):

15.4 Email Options

You can make all sorts of changes to how the debugging works. For example if you disable the `debug` variable in the config file Pylons will email you an error report instead of displaying it as long as you provide your email address at the top of the config file:

```
error_email_from = you@example.com
```

This is very useful for a production site. Emails are sent via SMTP so you need to specify a valid SMTP server too.

15.4.1 Error Handling Options

A number of error handling options can be specified in the config file. These are described in the *Interactive debugging* documentation but the important point to remember is that debug should always be set to `false` in production environments otherwise if an error occurs the visitor will be presented with the developer's interactive traceback which they could use to execute malicious code.

CHAPTER

SIXTEEN

UPGRADING

Upgrading your project is slightly different depending on which versions you're upgrading from and to. It's recommended that upgrades be done in minor revision steps, as deprecation warnings are added between revisions to help in the upgrade process.

For example, if you're running 0.9.4, first upgrade to 0.9.5, then 0.9.6, then finally 0.9.7 when desired. The change to 0.9.7 can be done in two steps unlike the older upgrades which should follow the process documented here after the 0.9.7 upgrade.

16.1 Upgrading from 0.9.6 -> 0.9.7

Pylons 0.9.7 changes several implicit behaviors of 0.9.6, as well as toggling some new options of Routes, and using automatic HTML escaping in Mako. These changes can be done in waves, and do not need to be completed all at once for a 0.9.6 project to run under 0.9.7.

16.1.1 Minimal Steps to run a 0.9.6 project under 0.9.7

Add the following lines to `config/middleware.py`:

```
# Add these imports to the top
from beaker.middleware import CacheMiddleware, SessionMiddleware
from routes.middleware import RoutesMiddleware

# Add these below the 'CUSTOM MIDDLEWARE HERE' line, or if you removed
# that, add them immediately after the PylonsApp initialization
app = RoutesMiddleware(app, config['routes.map'])
app = SessionMiddleware(app, config)
app = CacheMiddleware(app, config)
```

The Rails helpers from WebHelpers are no longer automatically imported in the webhelpers package. To use them 'lib/helpers.py' should be changed to import them:

```
from webhelpers.rails import *
```

Your Pylons 0.9.6 project should now run without issue in Pylons 0.9.7. Note that some deprecation warnings will likely be thrown reminding you to upgrade other parts.

16.1.2 Moving to use the new features of 0.9.7

To use the complete set of new features in 0.9.7, such as the automatic HTML escaping, new webhelpers, and new error middleware, follow the What's new in Pylons 0.9.7 overview to determine how to change the other files in your project to use the new features.

16.2 Moving from a pre-0.9.6 to 0.9.6

Pylons projects should be updated using the paster command create. In addition to creating new projects, paster create when run over an existing project will provide several ways to update the project template to the latest version.

Using this tool properly can make upgrading a fairly minor task. For the purpose of this document, the project being upgraded will be called 'demoapp' and all commands will use that name.

16.2.1 Running `paster create` to upgrade

First, navigate to the directory *above* the project's main directory. The main directory is the one that contains the `setup.py`, `setup.cfg`, and `development.ini` files.

```
/home/joe/demoapp $ cd ..
/home/joe $
```

Then run paster create on the project directory:

```
/home/joe $ paster create demoapp -t pylons
```

paster will issue prompts to allow the handling conflicts and updates to the existing project files. The options available are (hit the key in the parens to perform the operation):

```
(d)iff them, and show the changes between the project files and the ones
that have changed in Pylons

(b)ackup the file and copy the new version into its place. The backup file that
is created will have a ``.bak`` extension.

(y)es to overwrite the existing file with the new one. This approach is generally
not recommended as it does not allow the developer to view the content of the file
that will be replaced and it offers no opportunity for later recovery of the content.
The option can be made less intrepid by first viewing the diff to ascertain if any
changes will be lost in the overwriting.

(n)o to overwrite, retain the existing file. Safe if nothing has changed.
```

It's recommended when upgrading your project that you always look at the diff first to see what has changed. Then either overwrite your existing one if you are not going to lose changes you want, or backup yours and write the new one in. You can then manually compare and add your changes back in.

CHAPTER

SEVENTEEN

PACKAGING AND DEPLOYMENT OVERVIEW

TODO: some of this is redundant to the (more current) *Configuration* doc – should be consolidated and cross-referenced

This document describes how a developer can take advantage of Pylons' application setup functionality to allow webmasters to easily set up their application.

Installation refers to the process of downloading and installing the application with *easy_install* whereas setup refers to the process of setting up an instance of an installed application so it is ready to be deployed.

For example, a wiki application might need to create database tables to use. The webmaster would only install the wiki `.egg` file once using *easy_install* but might want to run 5 wikis on the site so would setup the wiki 5 times, each time specifying a different database to use so that 5 wikis can run from the same code, but store their data in different databases.

17.1 Egg Files

Before you can understand how a user configures an application you have to understand how Pylons applications are distributed. All Pylons applications are distributed in `.egg` format. An egg is simply a Python executable package that has been put together into a single file.

You create an egg from your project by going into the project root directory and running the command:

```
$ python setup.py bdist_egg
```

If everything goes smoothly a `.egg` file with the correct name and version number appears in a newly created `dist` directory.

When a webmaster wants to install a Pylons application he will do so by downloading the egg and then installing it.

17.2 Installing as a Non-root User

It's quite possible when using shared hosting accounts that you do not have root access to install packages. In this case you can install *setuptools* based packages like Pylons and Pylons web applications in your home directory using a *virtualenv* setup. This way you can install all the packages you want to use without super-user access.

17.3 Understanding the Setup Process

Say you have written a Pylons wiki application called `wiki`. When a webmaster wants to install your wiki application he will run the following command to generate a config file:

```
$ paster make-config wiki wiki_production.ini
```

He will then edit the config file for his production environment with the settings he wants and then run this command to setup the application:

```
$ paster setup-app wiki_production.ini
```

Finally he might choose to deploy the wiki application through the paste server like this (although he could have chosen CGI/FastCGI/SCGI etc):

```
$ paster serve wiki_production.ini
```

The idea is that an application only needs to be installed once but if necessary can be set up multiple times, each with a different configuration.

All Pylons applications are installed in the same way, so you as the developer need to know how the above commands work.

17.3.1 Make Config

The `paster make-config` command looks for the file `deployment.ini_tmpl` and uses it as a basis for generating a new `.ini` file.

Using our new wiki example again, the `wiki/config/deployment.ini_tmpl` file contains the text:

```
[DEFAULT]
debug = true
email_to = you@yourdomain.com
smtp_server = localhost
error_email_from = paste@localhost

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000

[app:main]
use = egg:wiki
full_stack = true
static_files = true
cache_dir = %(here)s/data
beaker.session.key = wiki
beaker.session.secret = ${app_instance_secret}
app_instance_uuid = ${app_instance_uuid}

# If you'd like to fine-tune the individual locations of the cache data dirs
# for the Cache data, or the Session saves, un-comment the desired settings
# here:
#beaker.cache.data_dir = %(here)s/data/cache
#beaker.session.data_dir = %(here)s/data/sessions

# WARNING: *THE LINE BELOW MUST BE UNCOMMENTED ON A PRODUCTION ENVIRONMENT*
# Debug mode will enable the interactive debugging tool, allowing ANYONE to
```

```
# execute malicious code after an exception is raised.
set debug = false


# Logging configuration
[loggers]
keys = root

[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] %(message)s
```

When the command `paster make-config wiki wiki_production.ini` is run, the contents of this file are produced so you should tweak this file to provide sensible default configuration for production deployment of your app.

17.3.2 Setup App

The `paster setup-app` command references the newly created `.ini` file and calls the function `wiki.websetup.setup_app()` to set up the application. If your application needs to be set up before it can be used, you should edit the `websetup.py` file.

Here's an example which just prints the location of the cache directory via Python's logging facilities:

```
"""Setup the helloworld application"""
import logging

from pylons import config
from helloworld.config.environment import load_environment

log = logging.getLogger(__name__)

def setup_app(command, conf, vars):
    """Place any commands to setup helloworld here"""
    load_environment(conf.global_conf, conf.local_conf)
    log.info("Using cache dirctory %s" % config['cache.dir'])
```

For a more useful example, say your application needs a database set up and loaded with initial data. The user will specify the location of the database to use by editing the config file before running the `paster setup-app` command. The `setup_app()` function will then be able to load the configuration and act on it in the function body. This way, the `setup_app()` function can be used to initialize the database when `paster setup-app` is run. Using the optional *SQLAlchemy* project template support when creating a

Pylons project will set all of this up for you in a basic way. The *quickwiki_tutorial* illustrates an example of this configuration.

17.4 Deploying the Application

Once the application is setup it is ready to be deployed. There are lots of ways of deploying an application, one of which is to use the `paster serve` command which takes the configuration file that has already been used to setup the application and serves it on a local HTTP server for production use:

```
$ paster serve wiki_production.ini
```

More information on Paste deployment options is available on the Paste website at http://pythonpaste.org. See *Running Pylons Apps with Other Web Servers* for alternative Pylons deployment scenarios.

17.5 Advanced Usage

So far everything we have done has happened through the `paste.script.appinstall.Installer` class which looks for the `deployment.ini_tmpl` and `websetup.py` file and behaves accordingly.

If you need more control over how your application is installed you can use your own installer class. Create a file, for example `wiki/installer.py` and code your new installer class in the file by deriving it from the existing one:

```
from paste.script.appinstall import Installer
class MyInstaller(Installer):
    pass
```

You then override the functionality as necessary (have a look at the source code for `Installer` as a basis. You then change your application's `setup.py` file so that the `paste.app_install` entry point `main` points to your new installer:

```
entry_points="""
...
[paste.app_install]
main=wiki.installer:MyInstaller
...
""",
```

Depending on how you code your `MyInstaller` class you may not even need your `websetup.py` or `deployment.ini_tmpl` as you might have decided to create the `.ini` file and setup the application in an entirely different way.

CHAPTER

EIGHTEEN

RUNNING PYLONS APPS WITH OTHER WEB SERVERS

This document assumes that you have already installed a Pylons web application, and *Runtime Configuration* for it. Pylons applications use PasteDeploy to start up your Pylons WSGI application, and can use the flup package to provide a Fast-CGI, SCGI, or AJP connection to it.

18.1 Using Fast-CGI

Fast-CGI is a gateway to connect web severs like Apache and lighttpd to a CGI-style application. Out of the box, Pylons applications can run with Fast-CGI in either a threaded or forking mode. (Threaded is the recommended choice)

Setting a Pylons application to use Fast-CGI is very easy, and merely requires you to change the config line like so:

```
# default
[server:main]
use = egg:Paste#http

# Use Fastcgi threaded
[server:main]
use = egg:PasteScript#flup_fcgi_thread
host = 0.0.0.0
port = 6500
```

Note that you will need to install the flup package, which can be installed via easy_install:

```
$ easy_install -U flup
```

The options in the config file are passed onto flup. The two common ways to run Fast CGI is either using a socket to listen for requests, or listening on a port/host which allows a webserver to send your requests to web applications on a different machine.

To configure for a socket, your `server:main` section should look like this:

```
[server:main]
use = egg:PasteScript#flup_fcgi_thread
socket = /location/to/app.socket
```

If you want to listen on a host/port, the configuration cited in the first example will do the trick.

18.2 Apache Configuration

For this example, we will assume you're using Apache 2, though Apache 1 configuration will be very similar. First, make sure that you have the Apache mod_fastcgi module installed in your Apache.

There will most likely be a section where you declare your FastCGI servers, and whether they're external:

```
<IfModule mod_fastcgi.c>
FastCgiIpcDir /tmp
FastCgiExternalServer /some/path/to/app/myapp.fcgi -host some.host.com:6200
</IfModule>
```

In our example we'll assume you're going to run a Pylons web application listening on a host/port. Changing `-host` to `-socket` will let you use a Pylons web application listening on a socket.

The filename you give in the second option does not need to physically exist on the webserver, URIs that Apache resolve to this filename will be handled by the FastCGI application.

The other important line to ensure that your Apache webserver has is to indicate that fcgi scripts should be handled with Fast-CGI:

```
AddHandler fastcgi-script .fcgi
```

Finally, to configure your website to use the Fast CGI application you will need to indicate the script to be used:

```
<VirtualHost *:80>
    ServerAdmin george@monkey.com
    ServerName monkey.com
    ServerAlias www.monkey.com
    DocumentRoot /some/path/to/app

    ScriptAliasMatch ^(/.*)$ /some/path/to/app/myapp.fcgi$1
</VirtualHost>
```

Other useful directives should be added as needed, for example, the ErrorLog directive, etc. This configuration will result in all requests being sent to your FastCGI application.

18.3 PrefixMiddleware

`PrefixMiddleware` provides a way to manually override the root prefix (`SCRIPT_NAME`) of your application for certain situations.

When running an application under a prefix (such as '`/james`') in FastCGI/apache, the `SCRIPT_NAME` environment variable is automatically set to to the appropriate value: '`/james`'. Pylons' URL generators such as `url` always take the `SCRIPT_NAME` value into account.

One situation where `PrefixMiddleware` is required is when an application is accessed via a reverse proxy with a prefix. The application is accessed through the reverse proxy via the the URL prefix '`/james`', whereas the reverse proxy forwards those requests to the application at the prefix '`/`'.

The reverse proxy, being an entirely separate web server, has no way of specifying the `SCRIPT_NAME` variable; it must be manually set by a `PrefixMiddleware` instance. Without setting `SCRIPT_NAME`, `url` will generate URLs such as: '`/purchase_orders/1`', when it should be generating: '`/james/purchase_orders/1`'.

To filter your application through a `PrefixMiddleware` instance, add the following to the '`[app:main]`' section of your .ini file:

```
filter-with = proxy-prefix

[filter:proxy-prefix]
use = egg:PasteDeploy#prefix
prefix = /james
```

The name `proxy-prefix` simply acts as an identifier of the filter section; feel free to rename it.

These .ini settings are equivalent to adding the following to the end of your application's `config/middleware.py`, right before the `return app` line:

```
# This app is served behind a proxy via the following prefix (SCRIPT_NAME)
app = PrefixMiddleware(app, global_conf, prefix='/james')
```

This requires the additional import line:

```
from paste.deploy.config import PrefixMiddleware
```

Whereas the modification to `config/middleware.py` will setup an instance of `PrefixMiddleware` under every environment (.ini).

CHAPTER

NINETEEN

DOCUMENTING YOUR APPLICATION

TODO: this needs to be rewritten – Pudge is effectively dead

While the information in this document should be correct, it may not be entirely complete... Pudge is somewhat unruly to work with at this time, and you may need to experiment to find a working combination of package versions. In particular, it has been noted that an older version of Kid, like 0.9.1, may be required. You might also need to install {{RuleDispatch}} if you get errors related to {{FormEncode}} when attempting to build documentation.

Apologies for this suboptimal situation. Considerations are being taken to fix Pudge or supplant it for future versions of Pylons.

19.1 Introduction

Pylons comes with support for automatic documentation generation tools like Pudge.

Automatic documentation generation allows you to write your main documentation in the docs directory of your project as well as throughout the code itself using docstrings.

When you run a simple command all the documentation is built into sophisticated HTML.

19.2 Tutorial

First create a project as described in *Getting Started*.

You will notice a docs directory within your main project directory. This is where you should write your main documentation.

There is already an `index.txt` file in `docs` so you can already generate documentation. First we'll install Pudge and buildutils. By default, Pylons sets an option to use Pygments for syntax-highlighting of code in your documentation, so you'll need to install it too (unless you wish to remove the option from `setup.cfg`):

```
$ easy_install pudge buildutils
$ easy_install Pygments
```

then run the following command from your project's main directory where the `setup.py` file is:

```
$ python setup.py pudge
```

Note: The `pudge` command is currently disabled by default. Run the following command first to enable it:

..code-block:: bash

    $ python setup.py addcommand -p buildutils.pudge_command

Thanks to Yannick Gingras for the tip.

Pudge will produce output similar to the following to tell you what it is doing and show you any problems:

```
running pudge
generating documentation
copying: pudge\template\pythonpaste.org\rst.css -> do/docs/html\rst.css
copying: pudge\template\base\pudge.css -> do/docs/html\pudge.css
copying: pudge\template\pythonpaste.org\layout.css -> do/docs/html\layout.css
rendering: pudge\template\pythonpaste.org\site.css.kid -> site.css
colorizing: do/docs/html\do/__init__.py.html
colorizing: do/docs/html\do/tests/__init__.py.html
colorizing: do/docs/html\do/i18n/__init__.py.html
colorizing: do/docs/html\do/lib/__init__.py.html
colorizing: do/docs/html\do/controllers/__init__.py.html
colorizing: do/docs/html\do/model.py.html
```

Once finished you will notice a `docs/html` directory. The `index.html` is the main file which was generated from `docs/index.txt`.

19.3 Learning ReStructuredText

Python programs typically use a rather odd format for documentation called reStructuredText. It is designed so that the text file used to generate the HTML is as readable as possible but as a result can be a bit confusing for beginners.

Read the reStructuredText tutorial which is part of the docutils project.

Once you have mastered reStructuredText you can write documentation until your heart's content.

19.4 Using Docstrings

Docstrings are one of Python's most useful features if used properly. They are described in detail in the Python documentation but basically allow you to document any module, class, method or function, in fact just about anything. Users can then access this documentation interactively.

Try this:

```
>>> import pylons
>>> help(pylons)
...
```

As you can see if you tried it you get detailed information about the pylons module including the information in the docstring.

Docstrings are also extracted by Pudge so you can describe how to use all the controllers, actions and modules that make up your application. Pudge will extract that information and turn it into useful API documentation automatically.

Try clicking the `Modules` link in the HTML documentation you generated earlier or look at the Pylons source code for some examples of how to use docstrings.

19.5 Using doctest

The final useful thing about docstrings is that you can use the `doctest` module with them. `doctest` again is described in the Python documentation but it looks through your docstrings for things that look like Python code written at a Python prompt. Consider this example:

```
>>> a = 2
>>> b = 3
>>> a + b
5
```

If `doctest` was run on this file it would have found the example above and executed it. If when the expression `a + b` is executed the result was not `5`, `doctest` would raise an Exception.

This is a very handy way of checking that the examples in your documentation are actually correct.

To run `doctest` on a module use:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

The `if __name__ == "__main__":` part ensures that your module won't be tested if it is just imported, only if it is run from the command line

To run `doctest` on a file use:

```
import doctest
doctest.testfile("docs/index.txt")
```

You might consider incorporating this functionality in your `tests/test.py` file to improve the testing of your application.

19.6 Summary

So if you write your documentation in reStructuredText, in the `docs` directory and in your code's docstrings, liberally scattered with example code, Pylons provides a very useful and powerful system for you.

If you want to find out more information have a look at the Pudge documentation or try tinkering with your project's `setup.cfg` file which contains the Pudge settings.

CHAPTER

TWENTY

DISTRIBUTING YOUR APPLICATION

TODO: this assumes helloworld tutorial context that is no longer present, and could be consolidated with packaging info in *Packaging and Deployment Overview*

As mentioned earlier eggs are a convenient format for packaging applications. You can create an egg for your project like this:

```
$ cd helloworld
$ python setup.py bdist_egg
```

Your egg will be in the `dist` directory and will be called `helloworld-0.0.0dev-py2.4.egg`.

You can change options in `setup.py` to change information about your project. For example change version to `version="0.1.0",` and run `python setup.py bdist_egg` again to produce a new egg with an updated version number.

You can then register your application with the Python Package Index (PyPI) with the following command:

```
$ python setup.py register
```

Note: You should not do this unless you actually want to register a package!

If users want to install your software and have installed *easy_install* they can install your new egg as follows:

```
$ easy_install helloworld==0.1.0
```

This will retrieve the package from PyPI and install it. Alternatively you can install the egg locally:

```
$ easy_install -f C:\path\with\the\egg\files\in helloworld==0.1.0
```

In order to use the egg in a website you need to use Paste. You have already used Paste to create your Pylons template and to run a test server to test the tutorial application.

Paste is a set of tools available at http://pythonpaste.org for providing a uniform way in which all compatible Python web frameworks can work together. To run a paste application such as any Pylons application you need to create a Paste configuration file. The idea is that the your paste configuration file will contain all the configuration for all the different Paste applications you run. A configuration file suitable for development is in the `helloworld/development.ini` file of the tutorial but the idea is that the person using your egg will add relevant configuration options to their own Paste configuration file so that your egg behaves they way they want. See the section below for more on this configuration.

Paste configuration files can be run in many different ways, from CGI scripts, as standalone servers, with FastCGI, SCGI, mod_python and more. This flexibility means that your Pylons application can be run in virtually any environment and also take advantage of the speed benefits that the deployment option offers.

See Also:

Running Pylons Apps with Other Web Servers

20.1 Running Your Application

In order to run your application your users will need to install it as described above but then generate a config file and setup your application before deploying it. This is described in *Runtime Configuration* and *Packaging and Deployment Overview*.

CHAPTER

TWENTYONE

PYTHON 2.3 INSTALLATION INSTRUCTIONS

21.1 Advice of end of support for Python 2.3

Warning: **END OF SUPPORT FOR PYTHON 2.3** This is the **LAST** version to support Python 2.3 **BEGIN UPGRADING OR DIE**

21.2 Preparation

First, please note that Python 2.3 users on Windows will need to install subprocess.exe before beginning the installation (whereas Python 2.4 users on Windows do not). All windows users also should read the section *Windows Notes* after installation. Users of Ubuntu/debian will also likely need to install the python-dev package.

21.3 System-wide Install

To install Pylons so it can be used by everyone (you'll need root access).

If you already have easy install:

```
$ easy_install Pylons==0.9.7
```

Note: On rare occasions, the python.org Cheeseshop goes down. It is still possible to install Pylons and its dependencies however by specifying our local package directory for installation with:

```
$ easy_install -f http://pylonshq.com/download/ Pylons==0.9.7
```

Which will use the packages necessary for the latest release. If you're using an older version of Pylons, you can get the packages that went with it by specifying the version desired:

```
$ easy_install -f http://pylonshq.com/download/0.9.7/ Pylons==0.9.7
```

Otherwise:

1. Download the easy install setup file from http://peak.telecommunity.com/dist/ez_setup.py

2. Run:

```
$ python ez_setup.py Pylons==0.9.7
```

Warning: **END OF SUPPORT FOR PYTHON 2.3** This is the **LAST** version to support Python 2.3 **BEGIN UPGRADING OR DIE**

CHAPTER

TWENTYTWO

WINDOWS NOTES

Python scripts installed as part of the Pylons install process will be put in the `Scripts` directory of your Python installation, typically in `C:\Python24\Scripts`. By default on Windows, this directory is not in your `PATH`; this can cause the following error message when running a command such as `paster` from the command prompt:

```
C:\Documents and Settings\James>paster
'paster' is not recognized as an internal or external command,
operable program or batch file.
```

To run the scripts installed with Pylons either the full path must be specified:

```
C:\Documents and Settings\James>C:\Python24\Scripts\paster
Usage: C:\Python24\Scripts\paster-script.py COMMAND
usage: paster-script.py [paster_options] COMMAND [command_options]

options:
  --version          show program's version number and exit
  --plugin=PLUGINS   Add a plugin to the list of commands (plugins are Egg
                     specs; will also require() the Egg)
  -h, --help         Show this help message

... etc ...
```

or (the preferable solution) the `Scripts` directory must be added to the `PATH` as described below.

22.1 For Win2K or WinXP

1. From the desktop or Start Menu, right click My Computer and click Properties.
2. In the System Properties window, click on the Advanced tab.
3. In the Advanced section, click the Environment Variables button.
4. Finally, in the Environment Variables window, highlight the path variable in the Systems Variable section and click edit. Add or modify the path lines with the paths you wish the computer to access. Each different directory is separated with a semicolon as shown below:

```
C:\Program Files;C:\WINDOWS;C:\WINDOWS\System32
```

1. Add the path to your scripts directory:

```
C:\Program Files;C:\WINDOWS;C:\WINDOWS\System32;C:\Python24\Scripts
```

See Finally below.

22.2 For Windows 95, 98 and ME

Edit `autoexec.bat`, and add the following line to the end of the file:

```
set PATH=%PATH%;C:\Python24\Scripts
```

See Finally below.

22.3 Finally

Restarting your computer may be required to enable the change to the `PATH`. Then commands may be entered from any location:

```
C:\Documents and Settings\James>paster
Usage: C:\Python24\Scripts\paster-script.py COMMAND
usage: paster-script.py [paster_options] COMMAND [command_options]

options:
  --version          show program's version number and exit
  --plugin=PLUGINS   Add a plugin to the list of commands (plugins are Egg
                     specs; will also require() the Egg)
  -h, --help         Show this help message

... etc ...
```

All documentation assumes the `PATH` is setup correctly as described above.

CHAPTER

TWENTYTHREE

SECURITY POLICY FOR BUGS

23.1 Receiving Security Updates

The Pylons team have set up a mailing list at wsgi-security-announce@googlegroups.com to which any security vulnerabilities that affect Pylons will be announced. Anyone wishing to be notified of vulnerabilities in Pylons should subscribe to this list. Security announcements will only be made once a solution to the problem has been discovered.

23.2 Reporting Security Issues

Please report security issues by email to both the lead developers of Pylons at the following addresses:

ben@groovie.org

security@3aims.com

Please DO NOT announce the vulnerability to any mailing lists or on the ticket system because we would not want any malicious person to be aware of the problem before a solution is available.

In the event of a confirmed vulnerability in Pylons itself, we will take the following actions:

- Acknowledge to the reporter that we've received the report and that a fix is forthcoming. We'll give a rough timeline and ask the reporter to keep the issue confidential until we announce it.
- Halt all other development as long as is needed to develop a fix, including patches against the current release.
- Publicly announce the vulnerability and the fix as soon as it is available to the WSGI security list at wsgi-security-announce@googlegroups.com.

This will probably mean a new release of Pylons, but in some cases it may simply be the release of documentation explaining how to avoid the vulnerability.

In the event of a confirmed vulnerability in one of the components that Pylons uses, we will take the following actions:

- Acknowledge to the reporter that we've received the report and ask the reporter to keep the issue confidential until we announce it.
- Contact the developer or maintainer of the package containing the vulnerability.
- If the developer or maintainer fails to release a new version in a reasonable time-scale and the vulnerability is serious we will either create documentation explaining how to avoid the problem or as a last resort, create a patched version.

- Publicly announce the vulnerability and the fix as soon as it is available to the WSGI security list at wsgi-security-announce@googlegroups.com.

23.3 Minimising Risk

- Only use official production versions of Pylons released publicly on the Python Package Index.
- Only use stable releases of third party software not development, alpha, beta or release candidate code.
- Do not assume that related software is of the same quality as Pylons itself, even if Pylons users frequently make use of it.
- Subscribe to the wsgi-security-announce@googlegroups.com mailing list to be informed of security issues and their solutions.

CHAPTER

TWENTYFOUR

WSGI SUPPORT

The Web Server Gateway Interface defined in PEP 333 is a standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers.

Pylons supports the Web Server Gateway Interface (or WSGI for short, pronounced "wizgy") throughout its stack. This is important for developers because it means that as well coming with all the features you would expect of a modern web framework, Pylons is also extremely flexible. With the WSGI it is possible to change any part of the Pylons stack to add new functionality or modify a request or a response without having to take apart the whole framework.

24.1 Paste and WSGI

Most of Pylons' WSGI capability comes from its close integration with Paste. Paste provides all the tools and middleware necessary to deploy WSGI applications. It can be thought of as a low-level WSGI framework designed for other web frameworks to build upon. Pylons is an example of a framework which makes full use of the possibilities of Paste.

If you want to, you can get the WSGI application object from your Pylons configuration file like this:

```
from paste.deploy import loadapp
wsgi_app = loadapp('config:/path/to/config.ini')
```

You can then serve the file using a WSGI server. Here is an example using the WSGI Reference Implementation to be included with Python 2.5:

```
from paste.deploy import loadapp
wsgi_app = loadapp('config:/path/to/config.ini')

from wsgiref import simple_server
httpd = simple_server.WSGIServer(('',8000), simple_server.WSGIRequestHandler)
httpd.set_app(wsgi_app)
httpd.serve_forever()
```

The `paster serve` command you will be used to using during the development of Pylons projects combines these two steps of creating a WSGI app from the config file and serving the resulting file to give the illusion that it is serving the config file directly.

Because the resulting Pylons application is a WSGI application it means you can do the same things with it that you can do with any WSGI application. For example add a middleware chain to it or serve it via FastCGI/SCGI/CGI/mod_python/AJP or standalone.

You can also configure extra WSGI middleware, applications and more directly using the configuration file. The various options are described in the Paste Deploy Documentation so we won't repeat them here.

24.2 Using a WSGI Application as a Pylons 0.9 Controller

In Pylons 0.9 controllers are derived from `pylons.controllers.WSGIController` and are also valid WSGI applications. Unless your controller is derived from the legacy `pylons.controllers.Controller` class it is also assumed to be a WSGI application. This means that you don't actually need to use a Pylons controller class in your controller, any WSGI application will work as long as you give it the same name.

For example, if you added a `hello` controller by executing `paster controller hello`, you could modify it to look like this:

```
def HelloController(environ, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return ['Hello World!']
```

or use `yield` statements like this:

```
def HelloController(environ, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    yield 'Hello '
    yield 'World!'
```

or use the standard Pylons `Response` object which is a valid WSGI response which takes care of calling `start_response()` for you:

```
def HelloController(environ, start_response):
    return Response('Hello World!')
```

and you could use the `render()` and `render_response()` objects exactly like you would in a normal controller action.

As well as writing your WSGI application as a function you could write it as a class:

```
class HelloController:

    def __call__(self, environ, start_response):
        start_response('200 OK', [('Content-Type','text/html')])
        return ['Hello World!']
```

All the standard Pylons middleware defined in `config/middleware.py` is still available.

24.3 Running a WSGI Application From Within a Controller

There may be occasions where you don't want to replace your entire controller with a WSGI application but simply want to run a WSGI application from with a controller action. If your project was called `test` and you had a WSGI application called `wsgi_app` you could even do this:

```
from test.lib.base import *

def wsgi_app(environ, start_response):
    start_response('200 OK',[('Content-type','text/html')])
    return ['<html>\n<body>\nHello World!\n</body>\n</html>']
```

```
class HelloController(BaseController):
    def index(self):
        return wsgi_app(request.environ, self.start_response)
```

24.4 Configuring Middleware Within a Pylons Application

A Pylons application middleware stack is directly exposed in the project's `config/middleware.py` file. This means that you can add and remove pieces from the stack as you choose.

> **Warning:** If you remove any of the default middleware you are likely to find that various parts of Pylons stop working!

As an example, if you wanted to add middleware that added a new key to the environ dictionary you might do this:

```
# YOUR MIDDLEWARE
# Put your own middleware here, so that any problems are caught by the error
# handling middleware underneath

class KeyAdder:
def __init__(self, app, key, value):
    self.app = app
    if '.' not in key:
        raise Exception("WSGI environ keys must contain a '.' character")
    self.key = key
    self.value = value

def __call__(self, environ, start_response):
    environ[self.key] = self.value
    return self.app(environ, start_response)

app = KeyAdder(app, 'test.hello', 'Hello World')
```

Then in your controller you could write:

```
return Response(request.environ['test.hello'])
```

and you would see your `Hello World!` message.

Of course, this isn't a particularly useful thing to do. Middleware classes can do one of four things or a combination of them:

- Change the environ dictionary
- Change the status
- Change the HTTP headers
- Change the response body of the application

With the ability to do these things as a middleware you can create authentication code, error handling middleware and more but the great thing about WSGI is that someone probably already has so you can consult the wsgi.org middleware list or have a look at the Paste project and reuse an exisiting piece of middleware.

24.5 The Cascade

Towards the end of the middleware stack in your project's `config/middleware.py` file you will find a special piece of middleware called the cascade:

```
app = Cascade([static_app, javascripts_app, app])
```

Passed a list of applications, `Cascade` will try each of them in turn. If one returns a 404 status code then the next application is tried until one of the applications returns a code other than `200` in which case its response is returned. If all applications fail, then the last application's failure response is used.

The three WSGI applications in the cascade serve files from your project's `public` directory first then if nothing matches, the WebHelpers module JavaScripts are searched and finally if no JavaScripts are found your Pylons app is tried. This is why the `public/index.html` file is served before your controller is executed and why you can put `/javascripts/` into your HTML and the files will be found.

You are free to change the order of the cascade or add extra WSGI applications to it before `app` so that other locations are checked before your Pylons application is executed.

24.6 Useful Resources

Whilst other frameworks have put WSGI adapters at the end of their stacks so that their applications can be served by WSGI servers, we hope you can see how fully Pylons embraces WSGI throughout its design to be the most flexible and extensible of the main Python web frameworks.

To find out more about the Web Server Gateway Interface you might find the following resources useful:

- PEP 333
- The WSGI website at wsgi.org
- XML.com articles: Introducing WSGI - Pythons Secret Web Weapon.html Part 1 Part 2

CHAPTER

TWENTYFIVE

ADVANCED PYLONS

25.1 WSGI, CLI scripts

25.1.1 Working with `wsgiwrappers.WSGIRequest`

Pylons uses a specialised *WSGIRequest* class that is accessible via the `paste.wsgiwrappers` module.

The `wsgiwrappers.WSGIRequest` object represents a WSGI request that has a more programmer-friendly interface. This interface does not expose every detail of the WSGI environment *(why?)* and does not attempt to express anything beyond what is available in the environment dictionary.

The only state maintained in this object is the desired `charset`, an associated errors handler and a `decode_param_names` option.

> *Unicode notes*
>
> When `charset` is set, the incoming parameter values will be automatically coerced to unicode objects of the charset encoding.
>
> When unicode is expected, `charset` will be overridden by the the value of the charset parameter set in the Content-Type header, if one was specified by the client.
>
> The incoming parameter names are not decoded to unicode unless the decode_param_names option is enabled.

The class variable `defaults` specifies default values for charset, errors, and language. These default values can be overridden for the current request via the registry *(what's a registry?)*.

The language default value is considered the fallback during i18n translations to ensure in odd cases that mixed languages don't occur should the language file contain the string but not another language in the accepted languages list. The language value only applies when getting a list of accepted languages from the HTTP Accept header.

This behavior is duplicated from Aquarium, and may seem strange but is very useful. Normally, everything in the code is in "en-us". However, the "en-us" translation catalog is usually empty. If the user requests ["en-us", "zh-cn"] and a translation isn't found for a string in "en-us", you don't want gettext to fallback to "zh-cn". You want it to just use the string itself. Hence, if a string isn't found in the language catalog, the string in the source code will be used.

All other state is kept in the environment dictionary; this is essential for interoperability.

You are free to subclass this object.

25.1.2 Attributes

GET

A dictionary-like object representing the QUERY_STRING parameters. Always present, possibly empty.

If the same key is present in the query string multiple times, a list of its values can be retrieved from the `MultiDict` via the :meth:`getall` method.

Returns a `MultiDict` container or, when charset is set, a `UnicodeMultiDict`.

POST

A dictionary-like object representing the `POST` body.

Most values are encoded strings, or unicode strings when charset is set. There may also be FieldStorage objects representing file uploads. If this is not a POST request, or the body is not encoded fields (e.g., an XMLRPC request) then this will be empty.

This will consume wsgi.input when first accessed if applicable, but the raw version will be put in environ['paste.parsed_formvars'].

Returns a MultiDict container or a UnicodeMultiDict when charset is set.

cookies

A dictionary of cookies, keyed by cookie name.

Just a plain dictionary, may be empty but not None.

defaults

```
{'errors': 'replace',
 'decode_param_names': False,
 'charset': None,
 'language': 'en-us'}
```

host

The host name, as provided in `HTTP_HOST` with a fall-back to `SERVER_NAME`

is_xhr

Returns a boolean if `X-Requested-With` is present and is a `XMLHttpRequest`

languages

Returns a (possibly empty) list of preferred languages, most preferred first.

params

A dictionary-like object of keys from `POST`, `GET`, `URL` dicts

Return a key value from the parameters, they are checked in the following order: POST, GET, URL

25.1.3 Additional methods supported:

getlist(key)

Returns a list of all the values by that key, collected from POST, GET, URL dicts

Returns a `MultiDict` container or a `UnicodeMultiDict` when `charset` is set.

urlvars

Return any variables matched in the URL (e.g. wsgiorg.routing_args).

25.1.4 Methods

__init__(self, environ)

determine_browser_charset(self)

Determine the encoding as specified by the browser via the Content-Type's `charset parameter`, if one is set

match_accept(self, mimetypes)

Return a list of specified mime-types that the browser's HTTP Accept header allows in the order provided.

25.2 Adding commands to Paster

25.2.1 Paster command

The command line will be `paster my-command arg1 arg2` if the current directory is the application egg, or `paster --plugin=MyPylonsApp my-command arg1 arg2` otherwise. In the latter case, `MyPylonsApp` must have been installed via `easy_install` or `python setup.py develop`.

Make a package directory for your commands:

```
$ mkdir myapp/commands
$ touch myapp/commands/__init__.py
```

Create a module `myapp/commands/my_command.py` like this:

```
from paste.script.command import Command

class MyCommand(Command):
        # Parser configuration
        summary = "--NO SUMMARY--"
```

```
        usage = "--NO USAGE--"
        group_name = "myapp"
        parser = Command.standard_parser(verbose=False)

        def command(self):
                import pprint
                print "Hello, app script world!"
                print
                print "My options are:"
                print "    ", pprint.pformat(vars(self.options))
                print "My args are:"
                print "    ", pprint.pformat(self.args)
                print
                print "My parser help is:"
                print
                print self.parser.format_help()
```

Note: The class _must_ define `.command`, `.parser`, and `.summary`

Modify the `entry_points` argument in `setup.py` to contain:

```
[paste.paster_command]
my-command = myapp.commands.my_command:MyCommand
```

Run `python setup.py develop` or `easy_install .` to update the entry points in the egg in sys.path.

Now you should be able to run:

```
$ paster --plugin=MyApp my-command arg1 arg2
Hello, MyApp script world!

My options are:
         {'interactive': False, 'overwrite': False, 'quiet': 0, 'verbose': 0}
My args are:
         ['arg1', 'arg2']

My parser help is:

Usage: /usr/local/bin/paster my-command [options] --NO USAGE--
--NO SUMMARY--

Options:
  -h, --help  show this help message and exit

$ paster --plugin=MyApp --help
Usage: paster [paster_options] COMMAND [command_options]

...
myapp:
  my-command      --NO SUMMARY--

pylons:
  controller      Create a Controller and accompanying functional test
  restcontroller  Create a REST Controller and accompanying functional test
  shell           Open an interactive shell with the Pylons app loaded
```

25.2.2 Required class attributes

In addition to the `.command` method, the class should define `.parser` and `.summary`.

25.2.3 Command-line options

`Command.standard_parser()` returns a Python `OptionParser`. Calling `parser.add_option` enables the developer to add as many options as desired. Inside the `.command` method, the user's options are available under `self.options`, and any additional arguments are in `self.args`.

There are several other class attributes that affect the parser; see them defined in `paste.script.command:Command`. The most useful attributes are `.usage`, `.description`, `.min_args`, and `.max_args`. `.usage` is the part of the usage string _after_ the command name. The `.standard_parser()` method has several optional arguments to add standardized options; some of these got added to my parser although I don't see how.

See the `paster shell` command, `pylons.commands:ShellCommand`, for an example of using command-line options and loading the `.ini file` and model.

Also see "paster setup-app" where it is defined in `paste.script.appinstall.SetupCommand`. This is evident from the entry point in PasteScript (`PasteScript-VERSION.egg/EGG_INFO/entry_points.txt`). It is a complex example of reading a config file and delegating to another entry point.

The code for calling `myapp.websetup:setup_config` is in `paste.script.appinstall`.

The `Command` class also has several convenience methods to handle console prompts, enable logging, verify directories exist and that files have expected content, insert text into a file, run a shell command, add files to Subversion, parse "var=value" arguments, add variables to an .ini file.

25.2.4 Using paster to access a Pylons app

Paster provides `request` and `post` commands for running requests on an application. These commands will be run in the full configuration context of a normal application. Useful for cron jobs, the error handler will also be in place and you can get email reports of failed requests.

Because arguments all just go in `QUERY_STRING`, `request.GET` and `request.PARAMS` won't look like you expect. But you can parse them with something like:

```
parser = optparse.OptionParser()
parser.add_option(etc)

args = [item[0] for item in
        cgi.parse_qsl(request.environ['QUERY_STRING'])]

options, args = parser.parse_args(args)
```

paster request / post

Usage: paster request / post [options] CONFIG_FILE URL [OPTIONS/ARGUMENTS]

Run a request for the described application

This command makes an artifical request to a web application that uses a `paste.deploy` configuration file for the server and application. Use 'paster request config.ini /url' to request `/url`.

Use 'paster post config.ini /url < data' to do a POST with the given request body.

If the URL is relative (i.e. doesn't begin with /) it is interpreted as relative to /.command/.

The variable `environ['paste.command_request']` will be set to True in the request, so your application can distinguish these calls from normal requests.

Note that you can pass options besides the options listed here; any unknown options will be passed to the application in `environ['QUERY_STRING']`.

```
Options:
  -h, --help            show this help message and exit
  -v, --verbose
  -q, --quiet
  -n NAME, --app-name=NAME
                        Load the named application (default main)
  --config-var=NAME:VALUE
                        Variable to make available in the config for %()s
                        substitution (you can use this option multiple times)
  --header=NAME:VALUE   Header to add to request (you can use this option
                        multiple times)
  --display-headers     Display headers before the response body
```

Future development

A Pylons controller that handled some of this would probably be quite useful. Probably even nicer with additions to the current template, so that `/.command/` all gets routed to a single controller that uses actions for the various sub-commands, and can provide a useful response to `/.command/?-h`, etc.

25.3 Creating Paste templates

25.3.1 Introduction

Python Paste is an extremely powerful package that isn't just about WSGI middleware. The related document *Using Entry Points to Write Plugins* demonstrates how to use entry_points to create simple plugins. This document describes how to write just such a plugin for use Paste's project template creation facility and how to add a command to Paste's `paster` script.

The example task is to create a template for an imaginary content management system. The template is going to produce a project directory structure for a Python package, so we need to be able to specify a package name.

25.3.2 Creating The Directory Structure and Templates

The directory structure for the new project needs to look like this:

```
- default_project
    - +package+
        - __init__.py
        - static
            - layout
            - region
            - renderer
        - service
            - layout
                - __init__.py
```

```
            - region
                - __init__.py
            - renderer
                - __init__.py
    - setup.py_tmpl
    - setup.cfg_tmpl
    - development.ini_tmpl
    - README.txt_tmpl
    - ez_setup.py
```

Of course, the actual project's directory structure might look very different. In fact the `paster create` command can even be used to generate directory structures which *aren't* project templates — although this wasn't what it was designed for.

When the `paster create` command is run, any directories with `+package+` in their name will have that portion of the name replaced by a simplified package name and likewise any directories with `+egg+` in their name will have that portion replaced by the name of the egg directory, although we don't make use of that feature in this example.

All of the files with `_tmpl` at the end of their filenames are treated as templates and will have the variables they contain replaced automatically. All other files will remain unchanged.

Note: The small templating language used with `paster create` in files ending in `_tmpl` is described in detail in the Paste util module documentation

---

When specifying a package name it can include capitalisation and _ characters but it should be borne in mind that the actual name of the package will be the *lowercase* package name with the _ characters removed. If the package name contains an _, the egg name will contain a _ character so occasionally the `+egg+` name is different to the `+package+` name.

To avoid difficulty always recommend to users that they stick with package names that contain no _ characters so that the names remain unique when made lowercase.

25.3.3 Implementing the Code

Now that the directory structure has been defined, the next step is to implement the commands that will convert this to a ready-to-run project. The template creation commands are implemented by a class derived from `paste.script.templates.Template`. This is how our example appears:

```
from paste.script.templates import Template, var

vars = [
    var('version', 'Version (like 0.1)'),
    var('description', 'One-line description of the package'),
    var('long_description', 'Multi-line description (in reST)'),
    var('keywords', 'Space-separated keywords/tags'),
    var('author', 'Author name'),
    var('author_email', 'Author email'),
    var('url', 'URL of homepage'),
    var('license_name', 'License name'),
    var('zip_safe', 'True/False: if the package can be distributed as a .zip file',
        default=False),
]

class ArtProjectTemplate(Template):
    _template_dir = 'templates/default_project'
```

```
    summary = 'Art project template'
    vars = vars
```

The `vars` arguments can all be set at run time and will be available to be used as (in this instance) Cheetah template variables in the files which end `_tmpl`. For example the `setup.py_tmpl` file for the `default_project` might look like this:

```
from setuptools import setup, find_packages

version = ${repr(version)|"0.0"}

setup(name=${repr(project)},
    version=version,
    description="${description|nothing}",
    long_description="""\
    ${long_description|nothing}""",
    classifiers=[],
    keywords=${repr(keywords)|empty},
    author=${repr(author)|empty},
    author_email=${repr(author_email)|empty},
    url=${repr(url)|empty},
    license=${repr(license_name)|empty},
    packages=find_packages(exclude=['ez_setup']),
    include_package_data=True,
    zip_safe=${repr(bool(zip_safe))|False},
    install_requires=[
      # Extra requirements go here #
    ],
    entry_points="""
        [paste.app_factory]
        main=${package}:make_app
    """,
)
```

Note how the variables specified in `vars` earlier are used to generate the actual `setup.py` file.

In order to use the new templates they must be hooked up to the `paster create` command by means of an entry point. In the `setup.py` file of the project (in which created the project template is going to be stored) we need to add the following:

```
entry_points="""
    [paste.paster_create_template]
    art_project=art.entry.template:ArtProjectTemplate
""",
```

We also need to add `PasteScript>=1.3` to the `install_requires` line.

```
install_requires=["PasteScript>=1.3"],
```

We just need to install the entry points now by running:

```
python setup.py develop
```

We should now be able to see a list of available templates with this command:

```
$ paster create --list-templates
```

Note: Windows users will need to add their Python scripts directory to their path or enter the full version of the command, similar to this:

```
C:\Python24\Scripts\paster.exe create --list-templates
```

You should see the following:

```
Available templates:
art_project:              Art project template
basic_package:            A basic setuptools-enabled package
```

There may be other projects too.

25.3.4 Troubleshooting

If the Art entries don't show up, check whether it is possible to import the `template.py` file because any errors are simply ignored by the paster create command rather than output as a warning.

If the code is correct, the issue might be that the entry points data hasn't been updated. Examine the Python `site-packages` directory and delete the `Art.egg-link` files, any `Art*.egg` files or directories and remove any entries for art from `easy_install.pth` (replacing `Art` with the name chosen for the project of course). Then re-run `python setup.py develop` to install the correct information.

If problems are still evident, then running the following code will print out a list of all entry points. It might help track the problem down:

```
import pkg_resources
    for x in pkg_resources.iter_group_name(None, None):
        print x
```

25.3.5 Using the Template

Now that the entry point is working, a new project can be created:

```
$ paster create --template=art TestProject
```

Paster will ask lots of questions based on the variables set up in `vars` earlier. Pressing `return` will cause the default to be used. The final result is a nice project template ready for people to start coding with.

25.3.6 Implementing Pylons Templates

If the development context is subject to a frequent need to create lots of Pylons projects, each with a slightly different setup from the standard Pylons defaults then it is probably desirable to create a customised Pylons template to use when generating projects. This can be done in exactly the way described in this document.

First, set up a new Python package, perhaps called something like `CustomPylons` (obviously, don't use the Pylons name because Pylons itself is already using it). Then check out the Pylons source code and copy the pylons/templates/default_project directory into the new project as a starting point. The next stage is to add the custom `vars` and `Template` class and set up the entry points in the `CustomPylons` `setup.py` file.

After those tasks have been completed, it is then possible to create customised templates (ultimately based on the Pylons one) by using the `CustomPylons` package.

25.4 Using Entry Points to Write Plugins

25.4.1 Introduction

An entry point is a Python object in a project's code that is identified by a string in the project's `setup.py` file. The entry point is referenced by a group and a name so that the object may be discoverabe. This means that another application can search for all the installed software that has an entry point with a particular group name, and then access the Python object associated with that name.

This is extremely useful because it means it is possible to write plugins for an appropriately-designed application that can be loaded at run time. This document describes just such an application.

It is important to understand that entry points are a feature of the new Python eggs package format and are *not* a standard feature of Python. To learn about eggs, their benefits, how to install them and how to set them up, see:

- Python Eggs
- Easy Install
- Setuptools

If reading the above documentation is inconvenient, suffice it to say that eggs are created via a similar `setup.py` file to the one used by Python's own distutils module — except that eggs have some powerful extra features such as entry points and the ability to specify module dependencies and have them automatically installed by `easy_install` when the application itself is installed.

For those developers unfamiliar with `distutils`: it is the standard mechanism by which Python packages should be distributed. To use it, add a `setup.py` file to the desired project, insert the required metadata and specify the important files. The `setup.py` file can be used to issue various commands which create distributions of the pacakge in various formats for users to install.

25.4.2 Creating Plugins

This document describes how to use entry points to create a plugin mechansim which allows new types of content to be added to a content management system but we are going to start by looking at the plugin.

Say the standard way the CMS creates a plugin is with the `make_plugin()` function. In order for a plugin to be a plugin it must therefore have the function which takes the same arguments as the `make_plugin()` function and returns a plugin. We are going to add some image plugins to the CMS so we setup a project with the following directory structure:

```
+ image_plugins
  + __init__.py
+ setup.py
```

The `image_plugins/__init__.py` file looks like this:

```
def make_jpeg_image_plugin():
    return "This would return the JPEG image plugin"

def make_png_image_plugin():
    return "This would return the PNG image plugin"
```

We have now defined our plugins so we need to define our entry points. First lets write a basic `setup.py` for the project:

```
from setuptools import setup, find_packages

setup(
    name='ImagePlugins',
    version="1.0",
    description="Image plugins for the imaginary CMS 1.0 project",
    author="James Gardner",
    packages=find_packages(),
    include_package_data=True,
)
```

When using `setuptools` we can specify the `find_packages()` function and `include_package_data=True` rather than having to manually list all the modules and package data like we had to do in the old `distutils` `setup.py`.

Because the plugin is designed to work with the (imaginary) CMS 1.0 package, we need to specify that the plugin requires the CMS to be installed too and so we add this line to the `setup()` function:

```
install_requires=["CMS>=1.0"],
```

Now when the plugins are installed, CMS 1.0 or above will be installed automatically if it is not already present.

There are lots of other arguments such as `author_email` or `url` which you can add to the `setup.py` function too.

We are interested in adding the entry points. We need to decide on a group name for the entry points. It is traditional to use the name of the package using the entry point, separated by a `.` character and then use a name that describes what the entry point does. For our example `cms.plugin` might be an appropriate name for the entry point. Since the `image_plugin` module contains two plugins we will need two entries. Add the following to the `setup.py` function:

```
entry_points="""
    [cms.plugin]
    jpg_image=image_plugin:make_jpeg_image_plugin
    png_image=image_plugin:make_jpeg_image_plugin
""",
```

Group names are specified in square brackets, plugin names are specified in the format `name=module.import.path:object_within_the_module`. The object doesn't have to be a function and can have any valid Python name. The module import path doesn't have to be a top level component as it is in this example and the name of the entry point doesn't have to be the same as the name of the object it is pointing to.

The developer can add as many entries as desired in each group as long as the names are different and the same holds for adding groups. It is also possible to specify the entry points as a Python dictionary rather than a string if that approach is preferred.

There are two more things we need to do to complete the plugin. The first is to include an `ez_setup` module so that if the user installing the plugin doesn't have `setuptools` installed, it will be installed for them. We do this by adding the follwoing to the very top of the `setup.py` file before the import:

```
from ez_setup import use_setuptools
use_setuptools()
```

We also need to download the `ez_setup.py` file into our project directory at the same level as `setup.py`.

Note: If you keep your project in SVN there is a trick you can use with the 'SVN:externals to keep the `ez_setup.py` file up to date.

Finally in order for the CMS to find the plugins we need to install them. We can do this with:

```
$ python setup.py install
```

as usual or, since we might go on to develop the plugins further we can install them using a special development mode which sets up the paths to run the plugins from the source rather than installing them to Python's `site-packages` directory:

```
$ python setup.py develop
```

Both commands will download and install `setuptools` if you don't already have it installed.

25.4.3 Using Plugins

Now that the plugin is written we need to write the code in the CMS package to load it. Luckily this is even easier.

There are actually lots of ways of discovering plugins. For example: by distribution name and version requirement (such as `ImagePlugins>=1.0`) or by the entry point group and name (eg `jpg_image`). For this example we are choosing the latter, here is a simple script for loading the plugins:

```
from pkg_resources import iter_entry_points
for object in iter_entry_points(group='cms.plugin', name=None):
    print object()

from pkg_resources import iter_entry_points
available_methods = []
for method_handler in iter_entry_points(group='authkit.method', name=None):
    available_methods.append(method_handler.load())
```

Executing this short script, will result in the following output:

```
This would return the JPEG image plugin
This would return the PNG image plugin
```

The `iter_entry_points()` function has looped though all the objects in the `cms.plugin` group and returned the function they were associated with. The application then called the function that the enrty point was pointing to.

We hope that we have demonstrated the power of entry points for building extensible code and developers are encouraged to read the pkg_resources module documentation to learn about some more features of the eggs format.

CHAPTER

TWENTYSIX

PYLONS MODULES

26.1 pylons.commands – Command line functions

26.1.1 Module Contents

26.2 pylons.configuration – Configuration object and defaults setup

26.2.1 Module Contents

26.3 pylons.controllers – Controllers

This module makes available the `WSGIController` and `XMLRPCController` for easier importing.

26.4 pylons.controllers.core – WSGIController Class

26.4.1 Module Contents

26.5 pylons.controllers.util – Controller Utility functions

26.5.1 Module Contents

26.6 pylons.controllers.xmlrpc – XMLRPCController Class

26.6.1 Module Contents

26.7 pylons.decorators – Decorators

26.7.1 Module Contents

26.8 pylons.decorators.cache – Cache Decorators

26.8.1 Module Contents

26.9 pylons.decorators.rest – REST-ful Decorators

26.9.1 Module Contents

26.10 pylons.decorators.secure – Secure Decorators

26.10.1 Module Contents

26.11 pylons.error – Error handling support

26.12 pylons.i18n.translation – Translation/Localization functions

26.12.1 Module Contents

26.13 pylons.log – Logging for WSGI errors

26.13.1 Module Contents

26.14 pylons.middleware – WSGI Middleware

26.14.1 Module Contents

Note:

The `errorware` dictionary is constructed from the settings in the *DEFAULT* section of development.ini. the recognised

- `error_email` = conf.get('email_to')
- `error_log` = conf.get('error_log', None)
- `smtp_server` = conf.get('smtp_server','localhost')
- `error_subject_prefix` = conf.get('error_subject_prefix', 'WebApp Error: ')
- `from_address` = conf.get('from_address', conf.get('error_email_from', 'pylons@yourapp.com'))
- `error_message` = conf.get('error_message', 'An internal server error occurred')

26.14.2 Referenced classes

Pylons middleware uses `WebError` to effect the error-handling. The two classes implicated are:

ErrorMiddleware

`weberror.errormiddleware` `weberror.errormiddleware.ErrorMiddleware`

EvalException

`weberror.evalexception` `weberror.evalexception.EvalException`

26.14.3 Legacy

Changed in version 0.9.7: These functions were deprecated in Pylons 0.9.7, and have been superceded by the `StatusCodeRedirect` middleware.

26.15 pylons.templating – Render functions and helpers

26.15.1 Module Contents

Legacy Render Functions

Legacy Buffet Functions

26.16 pylons.test – Test related functionality

26.16.1 Module Contents

26.17 pylons.util – Paste Template and Pylons utility functions

26.17.1 Module Contents

26.18 pylons.wsgiapp – PylonsWSGI App Creator

26.18.1 Module Contents

CHAPTER

TWENTYSEVEN

THIRD-PARTY COMPONENTS

27.1 `beaker` – Beaker Caching

27.1.1 Cache

This package contains the "front end" classes and functions for Beaker caching.

Included are the `Cache` and `CacheManager` classes, as well as the function decorators `region_decorate()`, `region_invalidate()`.

class `beaker.cache.`**`Cache`**(*namespace*, *type='memory'*, *expiretime=None*, *starttime=None*, *expire=None*, ***nsargs*)

Front-end to the containment API implementing a data cache.

Parameters

- **namespace** – the namespace of this Cache
- **type** – type of cache to use
- **expire** – seconds to keep cached data
- **expiretime** – seconds to keep cached data (legacy support)
- **starttime** – time when cache was cache was

`clear`()

Clear all the values from the namespace

`get`(*key*, ***kw*)

Retrieve a cached value from the container

`get_value`(*key*, ***kw*)

Retrieve a cached value from the container

class `beaker.cache.`**`CacheManager`**(***kwargs*)

`cache`(**args*, ***kwargs*)

Decorate a function to cache itself with supplied parameters

Parameters

- **args** – Used to make the key unique for this function, as in region() above.
- **kwargs** – Parameters to be passed to get_cache(), will override defaults

Example:

```
        # Assuming a cache object is available like:
        cache = CacheManager(dict_of_config_options)


        def populate_things():

            @cache.cache('mycache', expire=15)
            def load(search_term, limit, offset):
                return load_the_data(search_term, limit, offset)

            return load('rabbits', 20, 0)
```

Note: The function being decorated must only be called with positional arguments.

---

invalidate(*func*, **args*, ***kwargs*)
Invalidate a cache decorated function

This function only invalidates cache spaces created with the cache decorator.

Parameters

- **func** – Decorated function to invalidate
- **args** – Used to make the key unique for this function, as in region() above.
- **kwargs** – Parameters that were passed for use by get_cache(), note that this is only required if a `type` was specified for the function

Example:

```
        # Assuming a cache object is available like:
        cache = CacheManager(dict_of_config_options)


        def populate_things(invalidate=False):

            @cache.cache('mycache', type="file", expire=15)
            def load(search_term, limit, offset):
                return load_the_data(search_term, limit, offset)

            # If the results should be invalidated first
            if invalidate:
                cache.invalidate(load, 'mycache', 'rabbits', 20, 0, type="file")
            return load('rabbits', 20, 0)
```

region(*region*, **args*)
Decorate a function to cache itself using a cache region

The region decorator requires arguments if there are more than two of the same named function, in the same module. This is because the namespace used for the functions cache is based on the functions name and the module.

Example:

```
        # Assuming a cache object is available like:
        cache = CacheManager(dict_of_config_options)


        def populate_things():
```

```
        @cache.region('short_term', 'some_data')
        def load(search_term, limit, offset):
            return load_the_data(search_term, limit, offset)

        return load('rabbits', 20, 0)
```

Note: The function being decorated must only be called with positional arguments.

region_invalidate(*namespace*, *region*, **args*)
Invalidate a cache region namespace or decorated function

This function only invalidates cache spaces created with the cache_region decorator.

Parameters

- **namespace** – Either the namespace of the result to invalidate, or the cached function
- **region** – The region the function was cached to. If the function was cached to a single region then this argument can be None
- **args** – Arguments that were used to differentiate the cached function as well as the arguments passed to the decorated function

Example:

```
    # Assuming a cache object is available like:
    cache = CacheManager(dict_of_config_options)

    def populate_things(invalidate=False):

        @cache.region('short_term', 'some_data')
        def load(search_term, limit, offset):
            return load_the_data(search_term, limit, offset)

        # If the results should be invalidated first
        if invalidate:
            cache.region_invalidate(load, None, 'some_data',
                                    'rabbits', 20, 0)
        return load('rabbits', 20, 0)
```

27.1.2 Container

Container and Namespace classes

beaker.container.**ContainerContext**
alias of dict

class beaker.container.**Container**
Implements synchronization and value-creation logic for a 'value' stored in a NamespaceManager.

Container and its subclasses are deprecated. The Value class is now used for this purpose.

class beaker.container.**MemoryContainer**

class beaker.container.**DBMContainer**

class beaker.container.**NamespaceManager**(*namespace*)
Handles dictionary operations and locking for a namespace of values.

NamespaceManager provides a dictionary-like interface, implementing __getitem__(), __setitem__(), and __contains__(), as well as functions related to lock acquisition.

The implementation for setting and retrieving the namespace data is handled by subclasses.

NamespaceManager may be used alone, or may be accessed by one or more Value objects. Value objects provide per-key services like expiration times and automatic recreation of values.

Multiple NamespaceManagers created with a particular name will all share access to the same underlying datasource and will attempt to synchronize against a common mutex object. The scope of this sharing may be within a single process or across multiple processes, depending on the type of NamespaceManager used.

The NamespaceManager itself is generally threadsafe, except in the case of the DBMNamespaceManager in conjunction with the gdbm dbm implementation.

class beaker.container.**MemoryNamespaceManager**(*namespace*, ***kwargs*)
NamespaceManager that uses a Python dictionary for storage.

class beaker.container.**DBMNamespaceManager**(*namespace*, *dbmmodule=None*, *data_dir=None*, *dbm_dir=None*, *lock_dir=None*, *digest_filenames=True*, ***kwargs*)
NamespaceManager that uses dbm files for storage.

class beaker.container.**FileContainer**

class beaker.container.**FileNamespaceManager**(*namespace*, *data_dir=None*, *file_dir=None*, *lock_dir=None*, *digest_filenames=True*, ***kwargs*)
NamespaceManager that uses binary files for storage.

Each namespace is implemented as a single file storing a dictionary of key/value pairs, serialized using the Python pickle module.

exception beaker.container.**CreationAbortedError**
Deprecated.

27.1.3 Database

class beaker.ext.database.**DatabaseNamespaceManager**(*namespace*, *url=None*, *sa_opts=None*, *optimistic=False*, *table_name='beaker_cache'*, *data_dir=None*, *lock_dir=None*, *schema_name=None*, ***params*)

class beaker.ext.database.**DatabaseContainer**

27.1.4 Google

class beaker.ext.google.**GoogleNamespaceManager**(*namespace*, *table_name='beaker_cache'*, ***params*)

class beaker.ext.google.**GoogleContainer**

27.1.5 Memcached

class beaker.ext.memcached.**MemcachedNamespaceManager**(*namespace*, *url*, *memcache_module='auto'*, *data_dir=None*, *lock_dir=None*, ***kw*)

Provides the NamespaceManager API over a memcache client library.

class beaker.ext.memcached.**MemcachedContainer**

Container class which invokes MemcacheNamespaceManager.

Middleware

class beaker.middleware.**CacheMiddleware**(*app*, *config=None*, *environ_key='beaker.cache'*, ***kwargs*)

class beaker.middleware.**SessionMiddleware**(*wrap_app*, *config=None*, *environ_key='beaker.session'*, ***kwargs*)

27.1.6 Session

class beaker.session.**SignedCookie**(*secret*, *input=None*)

Extends python cookie to give digital signature support

class beaker.session.**Session**(*request*, *id=None*, *invalidate_corrupt=False*, *use_cookies=True*, *type=None*, *data_dir=None*, *key='beaker.session.id'*, *timeout=None*, *cookie_expires=True*, *cookie_domain=None*, *cookie_path='/'*, *secret=None*, *secure=False*, *namespace_class=None*, *httponly=False*, *encrypt_key=None*, *validate_key=None*, ***namespace_args*)

Session object that uses container package for storage.

Parameters

- **invalidate_corrupt** (*bool*) – How to handle corrupt data when loading. When set to True, then corrupt data will be silently invalidated and a new session created, otherwise invalid data will cause an exception.
- **use_cookies** (*bool*) – Whether or not cookies should be created. When set to False, it is assumed the user will handle storing the session on their own.
- **type** – What data backend type should be used to store the underlying session data
- **key** – The name the cookie should be set to.
- **timeout** (*int*) – How long session data is considered valid. This is used regardless of the cookie being present or not to determine whether session data is still valid.
- **cookie_expires** – Expiration date for cookie
- **cookie_domain** – Domain to use for the cookie.
- **cookie_path** – Path to use for the cookie.
- **secure** – Whether or not the cookie should only be sent over SSL.
- **httponly** – Whether or not the cookie should only be accessible by the browser not by JavaScript.
- **encrypt_key** – The key to use for the local session encryption, if not provided the session will not be encrypted.
- **validate_key** – The key used to sign the local encrypted session

class beaker.session.**SessionObject**(*environ*, ***params*)
Session proxy/lazy creator

This object proxies access to the actual session object, so that in the case that the session hasn't been used before, it will be setup. This avoid creating and loading the session from persistent storage unless its actually used during the request.

27.1.7 Synchronization

Synchronization functions.

File- and mutex-based mutual exclusion synchronizers are provided, as well as a name-based mutex which locks within an application based on a string name.

class beaker.synchronization.**NameLock**(*identifier=None*, *reentrant=False*)
a proxy for an RLock object that is stored in a name based registry.

Multiple threads can get a reference to the same RLock based on the name alone, and synchronize operations related to that name.

class beaker.synchronization.**SynchronizerImpl**
Base class for a synchronization object that allows multiple readers, single writers.

class beaker.synchronization.**FileSynchronizer**(*identifier*, *lock_dir*)
A synchronizer which locks using flock().

class beaker.synchronization.**ConditionSynchronizer**(*identifier*)
a synchronizer using a Condition.

27.1.8 Util

Beaker utilities

class beaker.util.**SyncDict**
An efficient/threadsafe singleton map algorithm, a.k.a. "get a value based on this key, and create if not found or not valid" paradigm:

exists && isvalid ? get : create

Designed to work with weakref dictionaries to expect items to asynchronously disappear from the dictionary.

Use python 2.3.3 or greater ! a major bug was just fixed in Nov. 2003 that was driving me nuts with garbage collection/weakrefs in this section.

class beaker.util.**WeakValuedRegistry**

class beaker.util.**ThreadLocal**
stores a value on a per-thread basis

beaker.util.**verify_directory**(*dir*)
verifies and creates a directory. tries to ignore collisions with other threads and processes.

beaker.util.**encoded_path**(*root*, *identifiers*, *extension='.enc'*, *depth=3*, *digest_filenames=True*)
Generate a unique file-accessible path from the given list of identifiers starting at the given root directory.

beaker.util.**verify_options**(*opt*, *types*, *error*)

beaker.util.**verify_rules**(*params*, *ruleset*)

beaker.util.**coerce_session_params**(*params*)

beaker.util.**coerce_cache_params**(*params*)

27.2 FormEncode

FormEncode is a validation and form generation package. The validation can be used separately from the form generation. The validation works on compound data structures, with all parts being nestable. It is separate from HTTP or any other input mechanism.

These module API docs are divided into section by category.

27.2.1 Core API

formencode.api

These functions are used mostly internally by FormEncode. Core classes for validation.

formencode.api.**is_validator**(*obj*)
Returns whether obj is a validator object or not.

class formencode.api.**Invalid**(*msg*, *value*, *state*, *error_list=None*, *error_dict=None*)
This is raised in response to invalid input. It has several public attributes:

msg: The message, *without* values substituted. For instance, if you want HTML quoting of values, you can apply that.

substituteArgs: The arguments (a dictionary) to go with *msg*.

str(self): The message describing the error, with values substituted.

value: The offending (invalid) value.

state: The state that went with this validator. This is an application-specific object.

error_list: If this was a compound validator that takes a repeating value, and sub-validator(s) had errors, then this is a list of those exceptions. The list will be the same length as the number of values – valid values will have None instead of an exception.

error_dict: Like *error_list*, but for dictionary compound validators.

__init__(*msg*, *value*, *state*, *error_list=None*, *error_dict=None*)

unpack_errors(*encode_variables=False*, *dict_char='.'*, *list_char='-'*)
Returns the error as a simple data structure – lists, dictionaries, and strings.

If encode_variables is true, then this will return a flat dictionary, encoded with variable_encode

class formencode.api.**Validator**(**args*, ***kw*)
The base class of most validators. See *IValidator* for more, and *FancyValidator* for the more common (and more featureful) class.

Messages

classmethod all_messages()
Return a dictionary of all the messages of this validator, and any subvalidators if present. Keys are message names, values may be a message or list of messages. This is really just intended for documentation purposes, to show someone all the messages that a validator or compound validator (like Schemas) can produce.

@@: Should this produce a more structured set of messages, so that messages could be unpacked into a rendered form to see the placement of all the messages? Well, probably so.

if_missing
alias of `NoDefault`

classmethod subvalidators()
Return any validators that this validator contains. This is not useful for functional, except to inspect what values are available. Specifically the `.all_messages()` method uses this to accumulate all possible messages.

class `formencode.api.`**FancyValidator**(**args*, ***kw*)
FancyValidator is the (abstract) superclass for various validators and converters. A subclass can validate, convert, or do both. There is no formal distinction made here.

Validators have two important external methods:

- .to_python(value, state): Attempts to convert the value. If there is a problem, or the value is not valid, an Invalid exception is raised. The argument for this exception is the (potentially HTML-formatted) error message to give the user.
- .from_python(value, state): Reverses to_python.

There are five important methods for subclasses to override, however none of these *have* to be overridden, only the ones that are appropriate for the validator:

- __init__(): if the *declarative.Declarative* model doesn't work for this.
- .validate_python(value, state): This should raise an error if necessary. The value is a Python object, either the result of to_python, or the input to from_python.
- .validate_other(value, state): Validates the source, before to_python, or after from_python. It's more common to use *.validate_python()* however.
- ._to_python(value, state): This returns the converted value, or raises an Invalid exception if there is an error. The argument to this exception should be the error message.
- ._from_python(value, state): Should undo .to_python() in some reasonable way, returning a string.

Validators should have no internal state besides the values given at instantiation. They should be reusable and reentrant.

All subclasses can take the arguments/instance variables:

- if_empty: If set, then this value will be returned if the input evaluates to false (empty list, empty string, None, etc), but not the 0 or False objects. This only applies to `.to_python()`.
- not_empty: If true, then if an empty value is given raise an error. (Both with `.to_python()` and also `.from_python()` if `.validate_python` is true).
- strip: If true and the input is a string, strip it (occurs before empty tests).
- if_invalid: If set, then when this validator would raise Invalid during `.to_python()`, instead return this value.
- if_invalid_python: If set, when the Python value (converted with `.from_python()`) is invalid, this value will be returned.
- accept_python: If True (the default), then `.validate_python()` and `.validate_other()` will not be called when `.from_python()` is used.

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

base64encode(*value*)
Encode a string in base64, stripping whitespace and removing newlines.

if_empty
alias of `NoDefault`

if_invalid
alias of `NoDefault`

if_invalid_python
alias of `NoDefault`

validate_other(*value*, *state*)
A validation method that doesn't do anything.

validate_python(*value*, *state*)
A validation method that doesn't do anything.

formencode.schema

The FormEncode schema is one of the most important parts of using FormEncode, as it lets you organize validators into parts that can be re-used between schemas. Generally, a single schema will represent an entire form, but may inherit other schemas for re-usable validation parts (ie, maybe multiple forms all requires first and last name).

class formencode.schema.**Schema**(**args*, ***kw*)
A schema validates a dictionary of values, applying different validators (be key) to the different values. If allow_extra_fields=True, keys without validators will be allowed; otherwise they will raise Invalid. If filter_extra_fields is set to true, then extra fields are not passed back in the results.

Validators are associated with keys either with a class syntax, or as keyword arguments (class syntax is usually easier). Something like:

```
class MySchema(Schema):
    name = Validators.PlainText()
    phone = Validators.PhoneNumber()
```

These will not be available as actual instance variables, but will be collected in a dictionary. To remove a validator in a subclass that is present in a superclass, set it to None, like:

```
class MySubSchema(MySchema):
    name = None
```

Note that missing fields are handled at the Schema level. Missing fields can have the 'missing' message set to specify the error message, or if that does not exist the *schema* message 'missingValue' is used.

Messages

badDictType: The input must be dict-like (not a `%(type)s`: `%(value)r`)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field `%(name)s` was not expected.

class `formencode.schema.`**SimpleFormValidator**(**args*, ***kw*)
This validator wraps a simple function that validates the form.

The function looks something like this:

```
>>> def validate(form_values, state, validator):
...     if form_values.get('country', 'US') == 'US':
...         if not form_values.get('state'):
...             return dict(state='You must enter a state')
...     if not form_values.get('country'):
...         form_values['country'] = 'US'
```

This tests that the field 'state' must be filled in if the country is US, and defaults that country value to 'US'. The `validator` argument is the SimpleFormValidator instance, which you can use to format messages or keep configuration state in if you like (for simple ad hoc validation you are unlikely to need it).

To create a validator from that function, you would do:

```
>>> from formencode.schema import SimpleFormValidator
>>> validator = SimpleFormValidator(validate)
>>> validator.to_python({'country': 'US', 'state': ''}, None)
Traceback (most recent call last):
    ...
Invalid: state: You must enter a state
>>> validator.to_python({'state': 'IL'}, None)
{'country': 'US', 'state': 'IL'}
```

The validate function can either return a single error message (that applies to the whole form), a dictionary that applies to the fields, None which means the form is valid, or it can raise Invalid.

Note that you may update the value_dict *in place*, but you cannot return a new value.

Another way to instantiate a validator is like this:

```
>>> @SimpleFormValidator.decorate()
... def MyValidator(value_dict, state):
...     return None # or some more useful validation
```

After this `MyValidator` will be a `SimpleFormValidator` instance (it won't be your function).

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

27.2.2 Validators

Validator/Converters for use with FormEncode.

class `formencode.validators.`**Bool**(**args*, ***kw*)
Always Valid, returns True or False based on the value and the existance of the value.

If you want to convert strings like `'true'` to booleans, then use `StringBool`.

Examples:

```
>>> Bool.to_python(0)
False
>>> Bool.to_python(1)
True
>>> Bool.to_python('')
False
>>> Bool.to_python(None)
False
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**CIDR**(**args*, ***kw*)
Formencode validator to check whether a string is in correct CIDR notation (IP address, or IP address plus /mask).

Examples:

```
>>> cidr = CIDR()
>>> cidr.to_python('127.0.0.1')
'127.0.0.1'
>>> cidr.to_python('299.0.0.1')
Traceback (most recent call last):
    ...
Invalid: The octets must be within the range of 0-255 (not '299')
>>> cidr.to_python('192.168.0.1/1')
Traceback (most recent call last):
    ...
Invalid: The network size (bits) must be within the range of 8-32 (not '1')
>>> cidr.to_python('asdf')
Traceback (most recent call last):
    ...
Invalid: Please enter a valid IP address (a.b.c.d) or IP network (a.b.c.d/e)
```

Messages

badFormat: Please enter a valid IP address (a.b.c.d) or IP network (a.b.c.d/e)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

illegalBits: The network size (bits) must be within the range of 8-32 (not `%(bits)r`)

illegalOctets: The octets must be within the range of 0-255 (not `%(octet)r`)

noneType: The input must be a string (not None)

class formencode.validators.**CreditCardValidator**(**args*, ***kw*)
Checks that credit card numbers are valid (if not real).

You pass in the name of the field that has the credit card type and the field with the credit card number. The credit card type should be one of "visa", "mastercard", "amex", "dinersclub", "discover", "jcb".

You must check the expiration date yourself (there is no relation between CC number/types and expiration dates).

```
>>> cc = CreditCardValidator()
>>> cc.to_python({'ccType': 'visa', 'ccNumber': '4111111111111111'})
{'ccNumber': '4111111111111111', 'ccType': 'visa'}
>>> cc.to_python({'ccType': 'visa', 'ccNumber': '411111111111111'})
Traceback (most recent call last):
    ...
Invalid: ccNumber: You did not enter a valid number of digits
>>> cc.to_python({'ccType': 'visa', 'ccNumber': '411111111111112'})
Traceback (most recent call last):
    ...
Invalid: ccNumber: You did not enter a valid number of digits
>>> cc().to_python({})
Traceback (most recent call last):
    ...
Invalid: The field ccType is missing
```

Messages

badLength: You did not enter a valid number of digits

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalidNumber: That number is not valid

missing_key: The field `%(key)s` is missing

noneType: The input must be a string (not None)

notANumber: Please enter only the number, no other characters

class formencode.validators.**CreditCardExpires**(**args*, ***kw*)
Checks that credit card expiration date is valid relative to the current date.

You pass in the name of the field that has the credit card expiration month and the field with the credit card expiration year.

```
>>> ed = CreditCardExpires()
>>> ed.to_python({'ccExpiresMonth': '11', 'ccExpiresYear': '2250'})
{'ccExpiresYear': '2250', 'ccExpiresMonth': '11'}
>>> ed.to_python({'ccExpiresMonth': '10', 'ccExpiresYear': '2005'})
Traceback (most recent call last):
    ...
Invalid: ccExpiresMonth: Invalid Expiration Date<br>
ccExpiresYear: Invalid Expiration Date
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalidNumber: Invalid Expiration Date

noneType: The input must be a string (not None)

notANumber: Please enter numbers only for month and year

class formencode.validators.**CreditCardSecurityCode**(**args*, ***kw*)
Checks that credit card security code has the correct number of digits for the given credit card type.

You pass in the name of the field that has the credit card type and the field with the credit card security code.

```
>>> code = CreditCardSecurityCode()
>>> code.to_python({'ccType': 'visa', 'ccCode': '111'})
{'ccType': 'visa', 'ccCode': '111'}
>>> code.to_python({'ccType': 'visa', 'ccCode': '1111'})
Traceback (most recent call last):
    ...
Invalid: ccCode: Invalid credit card security code length
```

Messages

badLength: Invalid credit card security code length

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

notANumber: Please enter numbers only for credit card security code

class `formencode.validators.`**`DateConverter`**`(`*args*, ***kw*`)`

Validates and converts a string date, like mm/yy, dd/mm/yy, dd-mm-yy, etc. Using `month_style` you can support `'mm/dd/yyyy'` or `'dd/mm/yyyy'`. Only these two general styles are supported.

Accepts English month names, also abbreviated. Returns value as a datetime object (you can get mx.DateTime objects if you use `datetime_module='mxDateTime'`). Two year dates are assumed to be within 1950-2020, with dates from 21-49 being ambiguous and signaling an error.

Use accept_day=False if you just want a month/year (like for a credit card expiration date).

```
>>> d = DateConverter()
>>> d.to_python('12/3/09')
datetime.date(2009, 12, 3)
>>> d.to_python('12/3/2009')
datetime.date(2009, 12, 3)
>>> d.to_python('2/30/04')
Traceback (most recent call last):
    ...
Invalid: That month only has 29 days
>>> d.to_python('13/2/05')
Traceback (most recent call last):
    ...
Invalid: Please enter a month from 1 to 12
>>> d.to_python('1/1/200')
Traceback (most recent call last):
    ...
Invalid: Please enter a four-digit year after 1899
```

If you change `month_style` you can get European-style dates:

```
>>> d = DateConverter(month_style='dd/mm/yyyy')
>>> date = d.to_python('12/3/09')
>>> date
datetime.date(2009, 3, 12)
>>> d.from_python(date)
'12/03/2009'
```

Messages

badFormat: Please enter the date in the form `%(format)s`

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

dayRange: That month only has `%(days)i` days

empty: Please enter a value

fourDigitYear: Please enter a four-digit year after 1899

invalidDate: That is not a valid day (`%(exception)s`)

invalidDay: Please enter a valid day

invalidYear: Please enter a number for the year

monthRange: Please enter a month from 1 to 12

noneType: The input must be a string (not None)

unknownMonthName: Unknown month name: `%(month)s`

wrongFormat: Please enter the date in the form `%(format)s`

class `formencode.validators.`**`DateValidator`**(**args*, ***kw*)

Validates that a date is within the given range. Be sure to call DateConverter first if you aren't expecting mxDateTime input.

`earliest_date` and `latest_date` may be functions; if so, they will be called each time before validating.

`after_now` means a time after the current timestamp; note that just a few milliseconds before now is invalid! `today_or_after` is more permissive, and ignores hours and minutes.

Examples:

```
>>> from datetime import datetime, timedelta
>>> d = DateValidator(earliest_date=datetime(2003, 1, 1))
>>> d.to_python(datetime(2004, 1, 1))
datetime.datetime(2004, 1, 1, 0, 0)
>>> d.to_python(datetime(2002, 1, 1))
Traceback (most recent call last):
    ...
Invalid: Date must be after Wednesday, 01 January 2003
>>> d.to_python(datetime(2003, 1, 1))
datetime.datetime(2003, 1, 1, 0, 0)
>>> d = DateValidator(after_now=True)
>>> now = datetime.now()
>>> d.to_python(now+timedelta(seconds=5)) == now+timedelta(seconds=5)
True
>>> d.to_python(now-timedelta(days=1))
Traceback (most recent call last):
    ...
Invalid: The date must be sometime in the future
>>> d.to_python(now+timedelta(days=1)) > now
True
>>> d = DateValidator(today_or_after=True)
>>> d.to_python(now) == now
True
```

Messages

after: Date must be after `%(date)s`

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

before: Date must be before `%(date)s`

date_format: %%A, %%d %%B %%Y

empty: Please enter a value

future: The date must be sometime in the future

noneType: The input must be a string (not None)

class formencode.validators.**DictConverter**(**args*, ***kw*)
Converts values based on a dictionary which has values as keys for the resultant values.

If `allowNull` is passed, it will not balk if a false value (e.g., '' or None) is given (it will return None in these cases).

to_python takes keys and gives values, from_python takes values and gives keys.

If you give hideDict=True, then the contents of the dictionary will not show up in error messages.

Examples:

```
>>> dc = DictConverter({1: 'one', 2: 'two'})
>>> dc.to_python(1)
'one'
>>> dc.from_python('one')
1
>>> dc.to_python(3)
Traceback (most recent call last):
    ....
Invalid: Enter a value from: 1; 2
>>> dc2 = dc(hideDict=True)
>>> dc2.hideDict
True
>>> dc2.dict
{1: 'one', 2: 'two'}
>>> dc2.to_python(3)
Traceback (most recent call last):
    ....
Invalid: Choose something
>>> dc.from_python('three')
Traceback (most recent call last):
    ....
Invalid: Nothing in my dictionary goes by the value 'three'.  Choose one of: 'one'; 'two'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

chooseKey: Enter a value from: `%(items)s`

chooseValue: Nothing in my dictionary goes by the value `%(value)s`. Choose one of: `%(items)s`

empty: Please enter a value

keyNotFound: Choose something

noneType: The input must be a string (not None)

valueNotFound: That value is not known

class formencode.validators.**Email**(**args*, ***kw*)
Validate an email address.

If you pass `resolve_domain=True`, then it will try to resolve the domain name to make sure it's valid. This takes longer, of course. You must have the pyDNS modules installed to look up DNS (MX and A) records.

```
>>> e = Email()
>>> e.to_python(' test@foo.com ')
'test@foo.com'
>>> e.to_python('test')
Traceback (most recent call last):
    ...
Invalid: An email address must contain a single @
>>> e.to_python('test@foobar')
Traceback (most recent call last):
    ...
Invalid: The domain portion of the email address is invalid (the portion after the @: foobar)
>>> e.to_python('test@foobar.com.5')
Traceback (most recent call last):
    ...
Invalid: The domain portion of the email address is invalid (the portion after the @: foobar.com
>>> e.to_python('test@foo..bar.com')
Traceback (most recent call last):
    ...
Invalid: The domain portion of the email address is invalid (the portion after the @: foo..bar.c
>>> e.to_python('test@.foo.bar.com')
Traceback (most recent call last):
    ...
Invalid: The domain portion of the email address is invalid (the portion after the @: .foo.bar.c
>>> e.to_python('nobody@xn--m7r7ml7t24h.com')
'nobody@xn--m7r7ml7t24h.com'
>>> e.to_python('o*reilly@test.com')
'o*reilly@test.com'
>>> e = Email(resolve_domain=True)
>>> e.resolve_domain
True
>>> e.to_python('doesnotexist@colorstudy.com')
'doesnotexist@colorstudy.com'
>>> e.to_python('test@nyu.edu')
'test@nyu.edu'
>>> # NOTE: If you do not have PyDNS installed this example won't work:
>>> e.to_python('test@thisdomaindoesnotexistithinkforsure.com')
Traceback (most recent call last):
    ...
Invalid: The domain of the email address does not exist (the portion after the @: thisdomaindoes
>>> e.to_python(u'test@google.com')
u'test@google.com'
>>> e = Email(not_empty=False)
>>> e.to_python('')
```

Messages

badDomain: The domain portion of the email address is invalid (the portion after the @: `%(domain)s`)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

badUsername: The username portion of the email address is invalid (the portion before the @: `%(username)s`)

domainDoesNotExist: The domain of the email address does not exist (the portion after the @: `%(domain)s`)

empty: Please enter an email address

noAt: An email address must contain a single @

noneType: The input must be a string (not None)

socketError: An error occured when trying to connect to the server: `%(error)s`

class formencode.validators.**Empty**(**args*, ***kw*)
Invalid unless the value is empty. Use cleverly, if at all.

Examples:

```
>>> Empty.to_python(0)
Traceback (most recent call last):
  ...
Invalid: You cannot enter a value here
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

notEmpty: You cannot enter a value here

class formencode.validators.**FieldsMatch**(**args*, ***kw*)
Tests that the given fields match, i.e., are identical. Useful for password+confirmation fields. Pass the list of field names in as *field_names*.

```
>>> f = FieldsMatch('pass', 'conf')
>>> f.to_python({'pass': 'xx', 'conf': 'xx'})
{'conf': 'xx', 'pass': 'xx'}
>>> f.to_python({'pass': 'xx', 'conf': 'yy'})
Traceback (most recent call last):
    ...
Invalid: conf: Fields do not match
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Fields do not match (should be `%(match)s`)

invalidNoMatch: Fields do not match

noneType: The input must be a string (not None)

notDict: Fields should be a dictionary

class formencode.validators.**FieldStorageUploadConverter**(**args*, ***kw*)
Handles cgi.FieldStorage instances that are file uploads.

This doesn't do any conversion, but it can detect empty upload fields (which appear like normal fields, but have no filename when no upload was given).

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**FileUploadKeeper**(**args*, ***kw*)
Takes two inputs (a dictionary with keys `static` and `upload`) and converts them into one value on

the Python side (a dictionary with `filename` and `content` keys). The upload takes priority over the static value. The filename may be None if it can't be discovered.

Handles uploads of both text and `cgi.FieldStorage` upload values.

This is basically for use when you have an upload field, and you want to keep the upload around even if the rest of the form submission fails. When converting *back* to the form submission, there may be extra values `'original_filename'` and `'original_content'`, which may want to use in your form to show the user you still have their content around.

To use this, make sure you are using variabledecode, then use something like:

```
<input type="file" name="myfield.upload">
<input type="hidden" name="myfield.static">
```

Then in your scheme:

```
class MyScheme(Scheme):
    myfield = FileUploadKeeper()
```

Note that big file uploads mean big hidden fields, and lots of bytes passed back and forth in the case of an error.

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class `formencode.validators.`**`FormValidator`**`(`*args*, ***kw*`)`
A FormValidator is something that can be chained with a Schema.

Unlike normal chaining the FormValidator can validate forms that aren't entirely valid.

The important method is .validate(), of course. It gets passed a dictionary of the (processed) values from the form. If you have .validate_partial_form set to True, then it will get the incomplete values as well – check with the "in" operator if the form was able to process any particular field.

Anyway, .validate() should return a string or a dictionary. If a string, it's an error message that applies to the whole form. If not, then it should be a dictionary of fieldName: errorMessage. The special key "form" is the error message for the form as a whole (i.e., a string is equivalent to {"form": string}).

Returns None on no errors.

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class `formencode.validators.`**`IndexListConverter`**`(`*args*, ***kw*`)`
Converts a index (which may be a string like '2') to the value in the given list.

Examples:

```
>>> index = IndexListConverter(['zero', 'one', 'two'])
>>> index.to_python(0)
'zero'
>>> index.from_python('zero')
0
>>> index.to_python('1')
```

```
    'one'
    >>> index.to_python(5)
    Traceback (most recent call last):
    Invalid: Index out of range
    >>> index(not_empty=True).to_python(None)
    Traceback (most recent call last):
    Invalid: Please enter a value
    >>> index.from_python('five')
    Traceback (most recent call last):
    Invalid: Item 'five' was not found in the list
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

integer: Must be an integer index

noneType: The input must be a string (not None)

notFound: Item `%(value)s` was not found in the list

outOfRange: Index out of range

class formencode.validators.**Int**(**args, **kw*)
Convert a value to an integer.

Example:

```
    >>> Int.to_python('10')
    10
    >>> Int.to_python('ten')
    Traceback (most recent call last):
        ...
    Invalid: Please enter an integer value
    >>> Int(min=5).to_python('6')
    6
    >>> Int(max=10).to_python('11')
    Traceback (most recent call last):
        ...
    Invalid: Please enter a number that is 10 or smaller
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

integer: Please enter an integer value

noneType: The input must be a string (not None)

tooHigh: Please enter a number that is `%(max)s` or smaller

tooLow: Please enter a number that is `%(min)s` or greater

class formencode.validators.**IPhoneNumberValidator**

class formencode.validators.**MACAddress**(**args, **kw*)
Formencode validator to check whether a string is a correct hardware (MAC) address.

Examples:

```
>>> mac = MACAddress()
>>> mac.to_python('aa:bb:cc:dd:ee:ff')
'aabbccddeeff'
>>> mac.to_python('aa:bb:cc:dd:ee:ff:e')
Traceback (most recent call last):
    ...
Invalid: A MAC address must contain 12 digits and A-F; the value you gave has 13 characters
>>> mac.to_python('aa:bb:cc:dd:ee:fx')
Traceback (most recent call last):
    ...
Invalid: MAC addresses may only contain 0-9 and A-F (and optionally :), not 'x'
>>> MACAddress(add_colons=True).to_python('aabbccddeeff')
'aa:bb:cc:dd:ee:ff'
```

Messages

badCharacter: MAC addresses may only contain 0-9 and A-F (and optionally :), not `%(char)r`

badLength: A MAC address must contain 12 digits and A-F; the value you gave has `%(length)s` characters

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**MaxLength**(**args*, ***kw*)

Invalid if the value is longer than *maxLength*. Uses len(), so it can work for strings, lists, or anything with length.

Examples:

```
>>> max5 = MaxLength(5)
>>> max5.to_python('12345')
'12345'
>>> max5.from_python('12345')
'12345'
>>> max5.to_python('123456')
Traceback (most recent call last):
  ...
Invalid: Enter a value less than 5 characters long
>>> max5(accept_python=False).from_python('123456')
Traceback (most recent call last):
  ...
Invalid: Enter a value less than 5 characters long
>>> max5.to_python([1, 2, 3])
[1, 2, 3]
>>> max5.to_python([1, 2, 3, 4, 5, 6])
Traceback (most recent call last):
  ...
Invalid: Enter a value less than 5 characters long
>>> max5.to_python(5)
Traceback (most recent call last):
  ...
Invalid: Invalid value (value with length expected)
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Invalid value (value with length expected)

noneType: The input must be a string (not None)

tooLong: Enter a value less than `%(maxLength)i` characters long

class formencode.validators.**MinLength**(**args*, ***kw*)

Invalid if the value is shorter than *minlength*. Uses len(), so it can work for strings, lists, or anything with length. Note that you **must** use `not_empty=True` if you don't want to accept empty values – empty values are not tested for length.

Examples:

```
>>> min5 = MinLength(5)
>>> min5.to_python('12345')
'12345'
>>> min5.from_python('12345')
'12345'
>>> min5.to_python('1234')
Traceback (most recent call last):
  ...
Invalid: Enter a value at least 5 characters long
>>> min5(accept_python=False).from_python('1234')
Traceback (most recent call last):
  ...
Invalid: Enter a value at least 5 characters long
>>> min5.to_python([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
>>> min5.to_python([1, 2, 3])
Traceback (most recent call last):
  ...
Invalid: Enter a value at least 5 characters long
>>> min5.to_python(5)
Traceback (most recent call last):
  ...
Invalid: Invalid value (value with length expected)
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Invalid value (value with length expected)

noneType: The input must be a string (not None)

tooShort: Enter a value at least `%(minLength)i` characters long

class formencode.validators.**Number**(**args*, ***kw*)

Convert a value to a float or integer.

Tries to convert it to an integer if no information is lost.

Example:

```
>>> Number.to_python('10')
10
>>> Number.to_python('10.5')
10.5
>>> Number.to_python('ten')
Traceback (most recent call last):
    ...
```

```
        Invalid: Please enter a number
        >>> Number(min=5).to_python('6.5')
        6.5
        >>> Number(max=10.5).to_python('11.5')
        Traceback (most recent call last):
            ...
        Invalid: Please enter a number that is 10.5 or smaller
        >>> Number().to_python('infinity')
        inf
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

number: Please enter a number

tooHigh: Please enter a number that is `%(max)s` or smaller

tooLow: Please enter a number that is `%(min)s` or greater

class formencode.validators.**NotEmpty**(**args*, ***kw*)
Invalid if value is empty (empty string, empty list, etc).

Generally for objects that Python considers false, except zero which is not considered invalid.

Examples:

```
>>> ne = NotEmpty(messages=dict(empty='enter something'))
>>> ne.to_python('')
Traceback (most recent call last):
  ...
Invalid: enter something
>>> ne.to_python(0)
0
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**OneOf**(**args*, ***kw*)
Tests that the value is one of the members of a given list.

If `testValueList=True`, then if the input value is a list or tuple, all the members of the sequence will be checked (i.e., the input must be a subset of the allowed values).

Use `hideList=True` to keep the list of valid values out of the error message in exceptions.

Examples:

```
>>> oneof = OneOf([1, 2, 3])
>>> oneof.to_python(1)
1
>>> oneof.to_python(4)
Traceback (most recent call last):
  ...
Invalid: Value must be one of: 1; 2; 3 (not 4)
>>> oneof(testValueList=True).to_python([2, 3, [1, 2, 3]])
```

```
    [2, 3, [1, 2, 3]]
    >>> oneof.to_python([2, 3, [1, 2, 3]])
    Traceback (most recent call last):
      ...
    Invalid: Value must be one of: 1; 2; 3 (not [2, 3, [1, 2, 3]])
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Invalid value

noneType: The input must be a string (not None)

notIn: Value must be one of: `%(items)s` (not `%(value)r`)

class `formencode.validators.`**`PhoneNumber`**

class `formencode.validators.`**`PlainText`**(**args*, ***kw*)
Test that the field contains only letters, numbers, underscore, and the hyphen. Subclasses Regex.

Examples:

```
    >>> PlainText.to_python('_this9_')
    '_this9_'
    >>> PlainText.from_python('  this  ')
    '  this  '
    >>> PlainText(accept_python=False).from_python('  this  ')
    Traceback (most recent call last):
      ...
    Invalid: Enter only letters, numbers, or _ (underscore)
    >>> PlainText(strip=True).to_python('  this  ')
    'this'
    >>> PlainText(strip=True).from_python('  this  ')
    'this'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: Enter only letters, numbers, or _ (underscore)

noneType: The input must be a string (not None)

class `formencode.validators.`**`PostalCode`**

class `formencode.validators.`**`Regex`**(**args*, ***kw*)
Invalid if the value doesn't match the regular expression *regex*.

The regular expression can be a compiled re object, or a string which will be compiled for you.

Use strip=True if you want to strip the value before validation, and as a form of conversion (often useful).

Examples:

```
    >>> cap = Regex(r'^[A-Z]+$')
    >>> cap.to_python('ABC')
    'ABC'
```

Note that `.from_python()` calls (in general) do not validate the input:

```
>>> cap.from_python('abc')
'abc'
>>> cap(accept_python=False).from_python('abc')
Traceback (most recent call last):
  ...
Invalid: The input is not valid
>>> cap.to_python(1)
Traceback (most recent call last):
  ...
Invalid: The input must be a string (not a <type 'int'>: 1)
>>> Regex(r'^[A-Z]+$', strip=True).to_python('  ABC  ')
'ABC'
>>> Regex(r'this', regexOps=('I',)).to_python('THIS')
'THIS'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

invalid: The input is not valid

noneType: The input must be a string (not None)

class formencode.validators.**RequireIfMissing**(**args*, ***kw*)

Require one field based on another field being present or missing.

This validator is applied to a form, not an individual field (usually using a Schema's `pre_validators` or `chained_validators`) and is available under both names `RequireIfMissing` and `RequireIfPresent`.

If you provide a `missing` value (a string key name) then if that field is missing the field must be entered. This gives you an either/or situation.

If you provide a `present` value (another string key name) then if that field is present, the required field must also be present.

```
>>> from formencode import validators
>>> v = validators.RequireIfPresent('phone_type', present='phone')
>>> v.to_python(dict(phone_type='', phone='510 420  4577'))
Traceback (most recent call last):
    ...
Invalid: You must give a value for phone_type
>>> v.to_python(dict(phone=''))
{'phone': ''}
```

Note that if you have a validator on the optionally-required field, you should probably use `if_missing=None`. This way you won't get an error from the Schema about a missing value. For example:

```
class PhoneInput(Schema):
    phone = PhoneNumber()
    phone_type = String(if_missing=None)
    chained_validators = [RequireifPresent('phone_type', present='phone')]
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**Set**(**args*, ***kw*)
This is for when you think you may return multiple values for a certain field.

This way the result will always be a list, even if there's only one result. It's equivalent to ForEach(convert_to_list=True).

If you give `use_set=True`, then it will return an actual `set` object.

```
>>> Set.to_python(None)
[]
>>> Set.to_python('this')
['this']
>>> Set.to_python(('this', 'that'))
['this', 'that']
>>> s = Set(use_set=True)
>>> s.to_python(None)
set([])
>>> s.to_python('this')
set(['this'])
>>> s.to_python(('this',))
set(['this'])
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**SignedString**(**args*, ***kw*)
Encodes a string into a signed string, and base64 encodes both the signature string and a random nonce.

It is up to you to provide a secret, and to keep the secret handy and consistent.

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

badsig: Signature is not correct

empty: Please enter a value

malformed: Value does not contain a signature

noneType: The input must be a string (not None)

class formencode.validators.**StateProvince**

class formencode.validators.**String**(**args*, ***kw*)
Converts things to string, but treats empty things as the empty string.

Also takes a *max* and *min* argument, and the string length must fall in that range.

Also you may give an *encoding* argument, which will encode any unicode that is found. Lists and tuples are joined with *list_joiner* (default `', '`) in `from_python`.

```
>>> String(min=2).to_python('a')
Traceback (most recent call last):
    ...
Invalid: Enter a value 2 characters long or more
>>> String(max=10).to_python('xxxxxxxxxxx')
```

```
    Traceback (most recent call last):
        ...
    Invalid: Enter a value not more than 10 characters long
    >>> String().from_python(None)
    ''
    >>> String().from_python([])
    ''
    >>> String().to_python(None)
    ''
    >>> String(min=3).to_python(None)
    Traceback (most recent call last):
        ...
    Invalid: Please enter a value
    >>> String(min=1).to_python('')
    Traceback (most recent call last):
        ...
    Invalid: Please enter a value
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

tooLong: Enter a value not more than `%(max)i` characters long

tooShort: Enter a value `%(min)i` characters long or more

class `formencode.validators.`**`StringBool`**(**args*, ***kw*)
Converts a string to a boolean.

Values like 'true' and 'false' are considered True and False, respectively; anything in `true_values` is true, anything in `false_values` is false, case-insensitive). The first item of those lists is considered the preferred form.

```
    >>> s = StringBool()
    >>> s.to_python('yes'), s.to_python('no')
    (True, False)
    >>> s.to_python(1), s.to_python('N')
    (True, False)
    >>> s.to_python('ye')
    Traceback (most recent call last):
        ...
    Invalid: Value should be 'true' or 'false'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

string: Value should be `%(true)r` or `%(false)r`

class `formencode.validators.`**`StripField`**(**args*, ***kw*)
Take a field from a dictionary, removing the key from the dictionary.

`name` is the key. The field value and a new copy of the dictionary with that field removed are returned.

```
>>> StripField('test').to_python({'a': 1, 'test': 2})
(2, {'a': 1})
>>> StripField('test').to_python({})
Traceback (most recent call last):
    ...
Invalid: The name 'test' is missing
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

missing: The name `%(name)s` is missing

noneType: The input must be a string (not None)

class `formencode.validators.`**`TimeConverter`**(**args*, ***kw*)

Converts times in the format HH:MM:SSampm to (h, m, s). Seconds are optional.

For ampm, set use_ampm = True. For seconds, use_seconds = True. Use 'optional' for either of these to make them optional.

Examples:

```
>>> tim = TimeConverter()
>>> tim.to_python('8:30')
(8, 30)
>>> tim.to_python('20:30')
(20, 30)
>>> tim.to_python('30:00')
Traceback (most recent call last):
    ...
Invalid: You must enter an hour in the range 0-23
>>> tim.to_python('13:00pm')
Traceback (most recent call last):
    ...
Invalid: You must enter an hour in the range 1-12
>>> tim.to_python('12:-1')
Traceback (most recent call last):
    ...
Invalid: You must enter a minute in the range 0-59
>>> tim.to_python('12:02pm')
(12, 2)
>>> tim.to_python('12:02am')
(0, 2)
>>> tim.to_python('1:00PM')
(13, 0)
>>> tim.from_python((13, 0))
'13:00:00'
>>> tim2 = tim(use_ampm=True, use_seconds=False)
>>> tim2.from_python((13, 0))
'1:00pm'
>>> tim2.from_python((0, 0))
'12:00am'
>>> tim2.from_python((12, 0))
'12:00pm'
```

Examples with `datetime.time`:

```
>>> v = TimeConverter(use_datetime=True)
>>> a = v.to_python('18:00')
>>> a
datetime.time(18, 0)
>>> b = v.to_python('30:00')
Traceback (most recent call last):
    ...
Invalid: You must enter an hour in the range 0-23
>>> v2 = TimeConverter(prefer_ampm=True, use_datetime=True)
>>> v2.from_python(a)
'6:00:00pm'
>>> v3 = TimeConverter(prefer_ampm=True,
...                    use_seconds=False, use_datetime=True)
>>> a = v3.to_python('18:00')
>>> a
datetime.time(18, 0)
>>> v3.from_python(a)
'6:00pm'
>>> a = v3.to_python('18:00:00')
Traceback (most recent call last):
    ...
Invalid: You may not enter seconds
```

Messages

badHour: You must enter an hour in the range `%(range)s`

badMinute: You must enter a minute in the range 0-59

badNumber: The `%(part)s` value you gave is not a number: `%(number)r`

badSecond: You must enter a second in the range 0-59

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

minutesRequired: You must enter minutes (after a :)

noAMPM: You must indicate AM or PM

noSeconds: You may not enter seconds

noneType: The input must be a string (not None)

secondsRequired: You must enter seconds

tooManyColon: There are too many :'s

class formencode.validators.**UnicodeString**(***kw*)

Converts things to unicode string, this is a specialization of the String class.

In addition to the String arguments, an encoding argument is also accepted. By default the encoding will be utf-8. You can overwrite this using the encoding parameter. You can also set inputEncoding and outputEncoding differently. An inputEncoding of None means "do not decode", an outputEncoding of None means "do not encode".

All converted strings are returned as Unicode strings.

```
>>> UnicodeString().to_python(None)
u''
>>> UnicodeString().to_python([])
u''
```

```
>>> UnicodeString(encoding='utf-7').to_python('Ni Ni Ni')
u'Ni Ni Ni'
```

Messages

badEncoding: Invalid data or incorrect encoding

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

tooLong: Enter a value not more than `%(max)i` characters long

tooShort: Enter a value `%(min)i` characters long or more

class formencode.validators.**URL**(**args*, ***kw*)

Validate a URL, either http://... or https://. If check_exists is true, then we'll actually make a request for the page.

If add_http is true, then if no scheme is present we'll add http://

```
>>> u = URL(add_http=True)
>>> u.to_python('foo.com')
'http://foo.com'
>>> u.to_python('http://hahaha.ha/bar.html')
'http://hahaha.ha/bar.html'
>>> u.to_python('http://xn--m7r7ml7t24h.com')
'http://xn--m7r7ml7t24h.com'
>>> u.to_python('http://foo.com/test?bar=baz&fleem=morx')
'http://foo.com/test?bar=baz&fleem=morx'
>>> u.to_python('http://foo.com/login?came_from=http%3A%2F%2Ffoo.com%2Ftest')
'http://foo.com/login?came_from=http%3A%2F%2Ffoo.com%2Ftest'
>>> u.to_python('http://foo.com:8000/test.html')
'http://foo.com:8000/test.html'
>>> u.to_python('http://foo.com/something\nelse')
Traceback (most recent call last):
    ...
Invalid: That is not a valid URL
>>> u.to_python('https://test.com')
'https://test.com'
>>> u.to_python('http://test')
Traceback (most recent call last):
    ...
Invalid: You must provide a full domain name (like test.com)
>>> u.to_python('http://test..com')
Traceback (most recent call last):
    ...
Invalid: That is not a valid URL
>>> u = URL(add_http=False, check_exists=True)
>>> u.to_python('http://google.com')
'http://google.com'
>>> u.to_python('google.com')
Traceback (most recent call last):
    ...
Invalid: You must start your URL with http://, https://, etc
>>> u.to_python('http://formencode.org/doesnotexist.html')
Traceback (most recent call last):
    ...
Invalid: The server responded that the page could not be found
```

```
>>> u.to_python('http://this.domain.does.not.exist.example.org/test.html')
...
Traceback (most recent call last):
    ...
Invalid: An error occured when trying to connect to the server: ...
```

If you want to allow addresses without a TLD (e.g., `localhost`) you can do:

```
>>> URL(require_tld=False).to_python('http://localhost')
'http://localhost'
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

badURL: That is not a valid URL

empty: Please enter a value

httpError: An error occurred when trying to access the URL: `%(error)s`

noScheme: You must start your URL with http://, https://, etc

noTLD: You must provide a full domain name (like `%(domain)s`.com)

noneType: The input must be a string (not None)

notFound: The server responded that the page could not be found

socketError: An error occured when trying to connect to the server: `%(error)s`

status: The server responded with a bad status code (`%(status)s`)

Wrapper Validators

class formencode.validators.**ConfirmType**(**args*, ***kw*)

Confirms that the input/output is of the proper type.

Uses the parameters:

subclass: The class or a tuple of classes; the item must be an instance of the class or a subclass.

type: A type or tuple of types (or classes); the item must be of the exact class or type. Subclasses are not allowed.

Examples:

```
>>> cint = ConfirmType(subclass=int)
>>> cint.to_python(True)
True
>>> cint.to_python('1')
Traceback (most recent call last):
    ...
Invalid: '1' is not a subclass of <type 'int'>
>>> cintfloat = ConfirmType(subclass=(float, int))
>>> cintfloat.to_python(1.0), cintfloat.from_python(1.0)
(1.0, 1.0)
>>> cintfloat.to_python(1), cintfloat.from_python(1)
(1, 1)
>>> cintfloat.to_python(None)
Traceback (most recent call last):
    ...
Invalid: None is not a subclass of one of the types <type 'float'>, <type 'int'>
```

```
>>> cint2 = ConfirmType(type=int)
>>> cint2(accept_python=False).from_python(True)
Traceback (most recent call last):
    ...
Invalid: True must be of the type <type 'int'>
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

inSubclass: `%(object)r` is not a subclass of one of the types `%(subclassList)s`

inType: `%(object)r` must be one of the types `%(typeList)s`

noneType: The input must be a string (not None)

subclass: `%(object)r` is not a subclass of `%(subclass)s`

type: `%(object)r` must be of the type `%(type)s`

class `formencode.validators.`**`Wrapper`**(**args*, ***kw*)
Used to convert functions to validator/converters.

You can give a simple function for *to_python*, *from_python*, *validate_python* or *validate_other*. If that function raises an exception, the value is considered invalid. Whatever value the function returns is considered the converted value.

Unlike validators, the *state* argument is not used. Functions like *int* can be used here, that take a single argument.

Examples:

```
>>> def downcase(v):
...     return v.lower()
>>> wrap = Wrapper(to_python=downcase)
>>> wrap.to_python('This')
'this'
>>> wrap.from_python('This')
'This'
>>> wrap2 = Wrapper(from_python=downcase)
>>> wrap2.from_python('This')
'this'
>>> wrap2.from_python(1)
Traceback (most recent call last):
  ...
Invalid: 'int' object has no attribute 'lower'
>>> wrap3 = Wrapper(validate_python=int)
>>> wrap3.to_python('1')
'1'
>>> wrap3.to_python('a')
Traceback (most recent call last):
  ...
Invalid: invalid literal for int()...
```

Messages

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

noneType: The input must be a string (not None)

class formencode.validators.**Constant**(**args*, ***kw*)
This converter converts everything to the same thing.

I.e., you pass in the constant value when initializing, then all values get converted to that constant value.

This is only really useful for funny situations, like:

```
fromEmailValidator = ValidateAny(
                           ValidEmailAddress(),
                           Constant('unknown@localhost'))
```

In this case, the if the email is not valid 'unknown@localhost' will be used instead. Of course, you could use if_invalid instead.

Examples:

```
>>> Constant('X').to_python('y')
'X'
```

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

27.2.3 Validator Modifiers

formencode.compound

Validators for applying validations in sequence.

class formencode.compound.**Any**(**args*, ***kw*)
This class is like an 'or' operator for validators. The first validator/converter that validates the value will be used. (You can pass in lists of validators, which will be ANDed)

class formencode.compound.**All**(**args*, ***kw*)
This class is like an 'and' operator for validators. All validators must work, and the results are passed in turn through all validators for conversion.

formencode.foreach

Validator for repeating items.

class formencode.foreach.**ForEach**(**args*, ***kw*)
Use this to apply a validator/converter to each item in a list.

For instance:

```
ForEach(AsInt(), InList([1, 2, 3]))
```

Will take a list of values and try to convert each of them to an integer, and then check if each integer is 1, 2, or 3. Using multiple arguments is equivalent to:

```
ForEach(All(AsInt(), InList([1, 2, 3])))
```

Use convert_to_list=True if you want to force the input to be a list. This will turn non-lists into one-element lists, and None into the empty list. This tries to detect sequences by iterating over them (except strings, which aren't considered sequences).

ForEach will try to convert the entire list, even if errors are encountered. If errors are encountered, they will be collected and a single Invalid exception will be raised at the end (with error_list set).

If the incoming value is a set, then we return a set.

27.2.4 HTML Parsing and Form Filling

formencode.htmlfill

Parser for HTML forms, that fills in defaults and errors. See `render`.

formencode.htmlfill.**render**(*form*, *defaults=None*, *errors=None*, *use_all_keys=False*, *error_formatters=None*, *add_attributes=None*, *auto_insert_errors=True*, *auto_error_formatter=None*, *text_as_default=False*, *listener=None*, *encoding=None*, *error_class='error'*, *prefix_error=True*, *force_defaults=True*)

Render the `form` (which should be a string) given the defaults and errors. Defaults are the values that go in the input fields (overwriting any values that are there) and errors are displayed inline in the form (and also effect input classes). Returns the rendered string.

If `auto_insert_errors` is true (the default) then any errors for which `<form:error>` tags can't be found will be put just above the associated input field, or at the top of the form if no field can be found.

If `use_all_keys` is true, if there are any extra fields from defaults or errors that couldn't be used in the form it will be an error.

`error_formatters` is a dictionary of formatter names to one-argument functions that format an error into HTML. Some default formatters are provided if you don't provide this.

`error_class` is the class added to input fields when there is an error for that field.

`add_attributes` is a dictionary of field names to a dictionary of attribute name/values. If the name starts with + then the value will be appended to any existing attribute (e.g., `{'+class': ' important'}`).

`auto_error_formatter` is used to create the HTML that goes above the fields. By default it wraps the error message in a span and adds a `<br>`.

If `text_as_default` is true (default false) then `<input type=unknown>` will be treated as text inputs.

`listener` can be an object that watches fields pass; the only one currently is in `htmlfill_schemabuilder.SchemaBuilder`

`encoding` specifies an encoding to assume when mixing str and unicode text in the template.

`prefix_error` specifies if the HTML created by auto_error_formatter is put before the input control (default) or after the control.

`force_defaults` specifies if a field default is not given in the `defaults` dictionary then the control associated with the field should be set as an unsuccessful control. So checkboxes will be cleared, radio and select controls will have no value selected, and textareas will be emptied. This defaults to `True`, which is appropriate the defaults are the result of a form submission.

formencode.htmlfill.**default_formatter**(*error*)
Formatter that escapes the error, wraps the error in a span with class `error-message`, and adds a `<br>`

formencode.htmlfill.**none_formatter**(*error*)
Formatter that does nothing, no escaping HTML, nothin'

formencode.htmlfill.**escape_formatter**(*error*)
Formatter that escapes HTML, no more.

formencode.htmlfill.**escapenl_formatter**(*error*)
Formatter that escapes HTML, and translates newlines to `<br>`

class formencode.htmlfill.**FillingParser**(*defaults*, *errors=None*, *use_all_keys=False*, *error_formatters=None*, *error_class='error'*, *add_attributes=None*, *listener=None*, *auto_error_formatter=None*, *text_as_default=False*, *encoding=None*, *prefix_error=True*, *force_defaults=True*)
Fills HTML with default values, as in a form.

Examples:

```
>>> defaults = dict(name='Bob Jones',
...                 occupation='Crazy Cultist',
...                 address='14 W. Canal\nNew Guinea',
...                 living='no',
...                 nice_guy=0)
>>> parser = FillingParser(defaults)
>>> parser.feed('''<input type="text" name="name" value="fill">
... <select name="occupation"> <option value="">Default</option>
... <option value="Crazy Cultist">Crazy cultist</option> </select>
... <textarea cols="20" style="width: 100%" name="address">
... An address</textarea>
... <input type="radio" name="living" value="yes">
... <input type="radio" name="living" value="no">
... <input type="checkbox" name="nice_guy" checked="checked">''')
>>> parser.close()
>>> print parser.text()
<input type="text" name="name" value="Bob Jones">
<select name="occupation">
<option value="">Default</option>
<option value="Crazy Cultist" selected="selected">Crazy cultist</option>
</select>
<textarea cols="20" style="width: 100%" name="address">14 W. Canal
New Guinea</textarea>
<input type="radio" name="living" value="yes">
<input type="radio" name="living" value="no" checked="checked">
<input type="checkbox" name="nice_guy">
```

27.3 routes – Route and Mapper core classes

27.3.1 routes.base

Route and Mapper core classes

class routes.base.**Route**(*name*, *routepath*, ***kargs*)
The Route object holds a route recognition and generation routine.

See Route.__init__ docs for usage.

buildfullreg(*clist*, *include_names=True*)
Build the regexp by iterating through the routelist and replacing dicts with the appropriate regexp match

buildnextreg(*path*, *clist*, *include_names=True*)
Recursively build our regexp given a path, and a controller list.

Returns the regular expression string, and two booleans that can be ignored as they're only used internally by buildnextreg.

generate(*_ignore_req_list=False*, *_append_slash=False*, ***kargs*)
Generate a URL from ourself given a set of keyword arguments

Toss an exception if this set of keywords would cause a gap in the url.

generate_minimized(*kargs*)
Generate a minimized version of the URL

generate_non_minimized(*kargs*)
Generate a non-minimal version of the URL

make_full_route()
Make a full routelist string for use with non-minimized generation

make_unicode(*s*)
Transform the given argument into a unicode string.

makeregexp(*clist*, *include_names=True*)
Create a regular expression for matching purposes

Note: This MUST be called before match can function properly.

clist should be a list of valid controller strings that can be matched, for this reason makeregexp should be called by the web framework after it knows all available controllers that can be utilized.

include_names indicates whether this should be a match regexp assigned to itself using regexp grouping names, or if names should be excluded for use in a single larger regexp to determine if any routes match

match(*url*, *environ=None*, *sub_domains=False*, *sub_domains_ignore=None*, *domain_match=''*)
Match a url to our regexp.

While the regexp might match, this operation isn't guaranteed as there's other factors that can cause a match to fail even though the regexp succeeds (Default that was relied on wasn't given, requirement regexp doesn't pass, etc.).

Therefore the calling function shouldn't assume this will return a valid dict, the other possible return is False if a match doesn't work out.

class routes.base.**Mapper**(*controller_scan=<function controller_scan at 0x43ac0c8>*, *directory=None*, *always_scan=False*, *register=True*, *explicit=True*)
Mapper handles URL generation and URL recognition in a web application.

Mapper is built handling dictionary's. It is assumed that the web application will handle the dictionary returned by URL recognition to dispatch appropriately.

URL generation is done by passing keyword parameters into the generate function, a URL is then returned.

connect (**args*, ***kargs*)
Create and connect a new Route to the Mapper.

Usage:

```
m = Mapper()
m.connect(':controller/:action/:id')
m.connect('date/:year/:month/:day', controller="blog", action="view")
m.connect('archives/:page', controller="blog", action="by_page",
requirements = { 'page':'\d{1,2}' })
m.connect('category_list', 'archives/category/:section', controller='blog', action='category
section='home', type='list')
m.connect('home', '', controller='blog', action='view', section='home')
```

create_regs (**args*, ***kwargs*)
Atomically creates regular expressions for all connected routes

extend (*routes*, *path_prefix=''*)
Extends the mapper routes with a list of Route objects

If a path_prefix is provided, all the routes will have their path prepended with the path_prefix.

Example:

```
>>> map = Mapper(controller_scan=None)
>>> map.connect('home', '/', controller='home', action='splash')
>>> map.matchlist[0].name == 'home'
True
>>> routes = [Route('index', '/index.htm', controller='home',
...                 action='index')]
>>> map.extend(routes)
>>> len(map.matchlist) == 2
True
>>> map.extend(routes, path_prefix='/subapp')
>>> len(map.matchlist) == 3
True
>>> map.matchlist[2].routepath == '/subapp/index.htm'
True
```

Note: This function does not merely extend the mapper with the given list of routes, it actually creates new routes with identical calling arguments.

---

generate (**args*, ***kargs*)
Generate a route from a set of keywords

Returns the url text, or None if no URL could be generated.

```
m.generate(controller='content',action='view',id=10)
```

match (*url=None*, *environ=None*)
Match a URL against against one of the routes contained.

Will return None if no valid match is found.

```
resultdict = m.match('/joe/sixpack')
```

redirect (*match_path*, *destination_path*, **args*, ***kwargs*)
Add a redirect route to the mapper

Redirect routes bypass the wrapped WSGI application and instead result in a redirect being issued by the RoutesMiddleware. As such, this method is only meaningful when using RoutesMiddleware.

By default, a 302 Found status code is used, this can be changed by providing a `_redirect_code` keyword argument which will then be used instead. Note that the entire status code string needs to be present.

When using keyword arguments, all arguments that apply to matching will be used for the match, while generation specific options will be used during generation. Thus all options normally available to connected Routes may be used with redirect routes as well.

Example:

```
map = Mapper()
map.redirect('/legacyapp/archives/{url:.*}, '/archives/{url})
map.redirect('/home/index', '/', _redirect_code='301 Moved Permanently')
```

resource(*member_name*, *collection_name*, ***kwargs*)
Generate routes for a controller resource

The member_name name should be the appropriate singular version of the resource given your locale and used with members of the collection. The collection_name name will be used to refer to the resource collection methods and should be a plural version of the member_name argument. By default, the member_name name will also be assumed to map to a controller you create.

The concept of a web resource maps somewhat directly to 'CRUD' operations. The overlying things to keep in mind is that mapping a resource is about handling creating, viewing, and editing that resource.

All keyword arguments are optional.

controller If specified in the keyword args, the controller will be the actual controller used, but the rest of the naming conventions used for the route names and URL paths are unchanged.

collection Additional action mappings used to manipulate/view the entire set of resources provided by the controller.

Example:

```
map.resource('message', 'messages', collection={'rss':'GET'})
# GET /message/rss (maps to the rss action)
# also adds named route "rss_message"
```

member Additional action mappings used to access an individual 'member' of this controllers resources.

Example:

```
map.resource('message', 'messages', member={'mark':'POST'})
# POST /message/1/mark (maps to the mark action)
# also adds named route "mark_message"
```

new Action mappings that involve dealing with a new member in the controller resources.

Example:

```
map.resource('message', 'messages', new={'preview':'POST'})
# POST /message/new/preview (maps to the preview action)
# also adds a url named "preview_new_message"
```

path_prefix Prepends the URL path for the Route with the path_prefix given. This is most useful for cases where you want to mix resources or relations between resources.

name_prefix Perpends the route names that are generated with the name_prefix given. Combined with the path_prefix option, it's easy to generate route names and paths that represent resources that are in relations.

Example:

```
map.resource('message', 'messages', controller='categories',
    path_prefix='/category/:category_id',
    name_prefix="category_")
# GET /category/7/message/1
# has named route "category_message"
```

parent_resource A `dict` containing information about the parent resource, for creating a nested resource. It should contain the `member_name` and `collection_name` of the parent resource. This `dict` will be available via the associated `Route` object which can be accessed during a request via `request.environ['routes.route']`

If `parent_resource` is supplied and `path_prefix` isn't, `path_prefix` will be generated from `parent_resource` as "<parent collection name>/:<parent member name>_id".

If `parent_resource` is supplied and `name_prefix` isn't, `name_prefix` will be generated from `parent_resource` as "<parent member name>_".

Example:

```
>>> from routes.util import url_for
>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'))
>>> # path_prefix is "regions/:region_id"
>>> # name prefix is "region_"
>>> url_for('region_locations', region_id=13)
'/regions/13/locations'
>>> url_for('region_new_location', region_id=13)
'/regions/13/locations/new'
>>> url_for('region_location', region_id=13, id=60)
'/regions/13/locations/60'
>>> url_for('region_edit_location', region_id=13, id=60)
'/regions/13/locations/60/edit'
```

Overriding generated `path_prefix`:

```
>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'),
...            path_prefix='areas/:area_id')
>>> # name prefix is "region_"
>>> url_for('region_locations', area_id=51)
'/areas/51/locations'
```

Overriding generated `name_prefix`:

```
>>> m = Mapper()
>>> m.resource('location', 'locations',
...            parent_resource=dict(member_name='region',
...                                 collection_name='regions'),
...            name_prefix='')
>>> # path_prefix is "regions/:region_id"
>>> url_for('locations', region_id=51)
'/regions/51/locations'
```

routematch(*url=None*, *environ=None*)

Match a URL against against one of the routes contained.

Will return None if no valid match is found, otherwise a result dict and a route object is returned.

```
resultdict, route_obj = m.match('/joe/sixpack')
```

27.3.2 routes.util

Utility functions for use in templates / controllers

PLEASE NOTE: Many of these functions expect an initialized RequestConfig object. This is expected to have been initialized for EACH REQUEST by the web framework.

routes.util.**url_for**(**args*, ***kargs*)

Generates a URL

All keys given to url_for are sent to the Routes Mapper instance for generation except for:

```
anchor          specified the anchor name to be appened to the path
host            overrides the default (current) host if provided
protocol        overrides the default (current) protocol if provided
qualified       creates the URL with the host/port information as
                needed
```

The URL is generated based on the rest of the keys. When generating a new URL, values will be used from the current request's parameters (if present). The following rules are used to determine when and how to keep the current requests parameters:

- If the controller is present and begins with '/', no defaults are used
- If the controller is changed, action is set to 'index' unless otherwise specified

For example, if the current request yielded a dict of {'controller': 'blog', 'action': 'view', 'id': 2}, with the standard ':controller/:action/:id' route, you'd get the following results:

```
url_for(id=4)                    =>  '/blog/view/4',
url_for(controller='/admin')     =>  '/admin',
url_for(controller='admin')      =>  '/admin/view/2'
url_for(action='edit')           =>  '/blog/edit/2',
url_for(action='list', id=None)  =>  '/blog/list'
```

Static and Named Routes

If there is a string present as the first argument, a lookup is done against the named routes table to see if there's any matching routes. The keyword defaults used with static routes will be sent in as GET query arg's if a route matches.

If no route by that name is found, the string is assumed to be a raw URL. Should the raw URL begin with / then appropriate SCRIPT_NAME data will be added if present, otherwise the string will be used as the url with keyword args becoming GET query args.

class routes.util.**URLGenerator**(*mapper*, *environ*)
The URL Generator generates URL's

It is automatically instantiated by the RoutesMiddleware and put into the wsgiorg.routing_args tuple accessible as:

```
url = environ['wsgiorg.routing_args'][0][0]
```

Or via the routes.url key:

```
url = environ['routes.url']
```

The url object may be instantiated outside of a web context for use in testing, however sub_domain support and fully qualified URL's cannot be generated without supplying a dict that must contain the key HTTP_HOST.

__call__(**args*, ***kargs*)
Generates a URL

All keys given to url_for are sent to the Routes Mapper instance for generation except for:

```
anchor          specified the anchor name to be appened to the path
host            overrides the default (current) host if provided
protocol        overrides the default (current) protocol if provided
qualified       creates the URL with the host/port information as
                needed
```

current(**args*, ***kwargs*)
Generate a route that includes params used on the current request

The arguments for this method are identical to __call__ except that arguments set to None will remove existing route matches of the same name from the set of arguments used to construct a URL.

routes.util.**redirect_to**(**args*, ***kargs*)
Issues a redirect based on the arguments.

Redirect's *should* occur as a "302 Moved" header, however the web framework may utilize a different method.

All arguments are passed to url_for to retrieve the appropriate URL, then the resulting URL it sent to the redirect function as the URL.

27.4 weberror – Weberror

27.4.1 weberror.errormiddleware

Error handler middleware

class weberror.errormiddleware.**ErrorMiddleware**(*application*, *global_conf=None*, *debug=<NoDefault>*, *error_email=None*, *error_log=None*, *show_exceptions_in_wsgi_errors=<NoDefault>*, *from_address=None*, *smtp_server=None*, *smtp_username=None*, *smtp_password=None*, *smtp_use_tls=False*, *error_subject_prefix=None*, *error_message=None*, *xmlhttp_key=None*, *reporters=None*)

Error handling middleware

Usage:

```
error_catching_wsgi_app = ErrorMiddleware(wsgi_app)
```

Settings:

debug: If true, then tracebacks will be shown in the browser.

error_email: an email address (or list of addresses) to send exception reports to

error_log: a filename to append tracebacks to

show_exceptions_in_wsgi_errors: If true, then errors will be printed to `wsgi.errors` (frequently a server error log, or stderr).

from_address, smtp_server, error_subject_prefix, smtp_username, smtp_password, smtp_use_t variables to control the emailed exception reports

error_message: When debug mode is off, the error message to show to users.

xmlhttp_key: When this key (default _) is in the request GET variables (not POST!), expect that this is an XMLHttpRequest, and the response should be more minimal; it should not be a complete HTML page.

Environment Configuration:

paste.throw_errors: If this setting in the request environment is true, then this middleware is disabled. This can be useful in a testing situation where you don't want errors to be caught and transformed.

paste.expected_exceptions: When this middleware encounters an exception listed in this environment variable and when the `start_response` has not yet occurred, the exception will be re-raised instead of being caught. This should generally be set by middleware that may (but probably shouldn't be) installed above this middleware, and wants to get certain exceptions. Exceptions raised after `start_response` have been called are always caught since by definition they are no longer expected.

27.4.2 weberror.evalcontext

class weberror.evalcontext.**EvalContext**(*namespace*, *globs*)

Class that represents a interactive interface. It has its own namespace. Use eval_context.exec_expr(expr) to run commands; the output of those commands is returned, as are print statements.

This is essentially what doctest does, and is taken directly from doctest.

27.4.3 weberror.evalexception

Exception-catching middleware that allows interactive debugging.

This middleware catches all unexpected exceptions. A normal traceback, like produced by `weberror.exceptions.errormiddleware.ErrorMiddleware` is given, plus controls to see local variables and evaluate expressions in a local context.

This can only be used in single-process environments, because subsequent requests must go back to the same process that the exception originally occurred in. Threaded or non-concurrent environments both work.

This shouldn't be used in production in any way. That would just be silly.

If calling from an XMLHttpRequest call, if the GET variable _ is given then it will make the response more compact (and less Javascripty), since if you use innerHTML it'll kill your browser. You can look for the header X-Debug-URL in your 500 responses if you want to see the full debuggable traceback. Also, this URL is printed to `wsgi.errors`, so you can open it up in another browser window.

class `weberror.evalexception.`**EvalException**(*application*, *global_conf=None*, *error_template_filename=None*, *xmlhttp_key=None*, *media_paths=None*, *templating_formatters=None*, *head_html=''*, *footer_html=''*, *reporters=None*, *libraries=None*, ***params*)

Handles capturing an exception and turning it into an interactive exception explorer

media(*req*)
Static path where images and other files live

relay(*req*)
Relay a request to a remote machine for JS proxying

summary(*req*)
Returns a JSON-format summary of all the cached exception reports

view(*req*)
View old exception reports

27.4.4 weberror.formatter

Formatters for the exception data that comes from ExceptionCollector.

class `weberror.formatter.`**AbstractFormatter**(*show_hidden_frames=False*, *include_reusable=True*, *show_extra_data=True*, *trim_source_paths=()*, ***kwargs*)

filter_frames(*frames*)
Removes any frames that should be hidden, according to the values of traceback_hide, self.show_hidden_frames, and the hidden status of the final frame.

format_frame_end(*frame*)
Called after each frame ends; may return None to output no text.

format_frame_start(*frame*)
Called before each frame starts; may return None to output no text.

long_item_list(*lst*)
Returns true if the list contains items that are long, and should be more nicely formatted.

pretty_string_repr(*s*)
Formats the string as a triple-quoted string when it contains newlines.

class weberror.formatter.**TextFormatter**(*show_hidden_frames=False, include_reusable=True, show_extra_data=True, trim_source_paths=(), **kwargs*)

class weberror.formatter.**HTMLFormatter**(*show_hidden_frames=False, include_reusable=True, show_extra_data=True, trim_source_paths=(), **kwargs*)

class weberror.formatter.**XMLFormatter**(*show_hidden_frames=False, include_reusable=True, show_extra_data=True, trim_source_paths=(), **kwargs*)

weberror.formatter.**create_text_node**(*doc, elem, text*)

weberror.formatter.**html_quote**(*s*)

weberror.formatter.**format_html**(*exc_data, include_hidden_frames=False, **ops*)

weberror.formatter.**format_text**(*exc_data, **ops*)

weberror.formatter.**format_xml**(*exc_data, **ops*)

weberror.formatter.**str2html**(*src, strip=False, indent_subsequent=0, highlight_inner=False, frame=None, filename=None*)
Convert a string to HTML. Try to be really safe about it, returning a quoted version of the string if nothing else works.

weberror.formatter.**_str2html**(*src, strip=False, indent_subsequent=0, highlight_inner=False, frame=None, filename=None*)

weberror.formatter.**truncate**(*string, limit=1000*)
Truncate the string to the limit number of characters

weberror.formatter.**make_wrappable**(*html, wrap_limit=60, split_on=';?&@!$#-/\\"\''*)

weberror.formatter.**make_pre_wrappable**(*html, wrap_limit=60, split_on=';?&@!$#-/\\"\''*)
Like `make_wrappable()` but intended for text that will go in a `<pre>` block, so wrap on a line-by-line basis.

27.4.5 weberror.reporter

class weberror.reporter.**Reporter**(***conf*)

class weberror.reporter.**EmailReporter**(***conf*)

class weberror.reporter.**LogReporter**(***conf*)

class weberror.reporter.**FileReporter**(***conf*)

class weberror.reporter.**WSGIAppReporter**(*exc_data*)

27.4.6 weberror.collector

An exception collector that finds traceback information plus supplements

class weberror.collector.**ExceptionCollector**(*limit=None*)
Produces a data structure that can be used by formatters to display exception reports.

Magic variables:

If you define one of these variables in your local scope, you can add information to tracebacks that happen in that context. This allows applications to add all sorts of extra information about the context of the error, including URLs, environmental variables, users, hostnames, etc. These are the variables we look for:

__traceback_supplement__: You can define this locally or globally (unlike all the other variables, which must be defined locally).

`__traceback_supplement__` is a tuple of `(factory, arg1, arg2...)`. When there is an exception, `factory(arg1, arg2, ...)` is called, and the resulting object is inspected for supplemental information.

__traceback_info__: This information is added to the traceback, usually fairly literally.

__traceback_hide__: If set and true, this indicates that the frame should be hidden from abbreviated tracebacks. This way you can hide some of the complexity of the larger framework and let the user focus on their own errors.

By setting it to `'before'`, all frames before this one will be thrown away. By setting it to `'after'` then all frames after this will be thrown away until `'reset'` is found. In each case the frame where it is set is included, unless you append `'_and_this'` to the value (e.g., `'before_and_this'`).

Note that formatters will ignore this entirely if the frame that contains the error wouldn't normally be shown according to these rules.

__traceback_reporter__: This should be a reporter object (see the reporter module), or a list/tuple of reporter objects. All reporters found this way will be given the exception, innermost first.

__traceback_decorator__: This object (defined in a local or global scope) will get the result of this function (the CollectedException defined below). It may modify this object in place, or return an entirely new object. This gives the object the ability to manipulate the traceback arbitrarily.

The actually interpretation of these values is largely up to the reporters and formatters.

`collect_exception(*sys.exc_info())` will return an object with several attributes:

frames: A list of frames

exception_formatted: The formatted exception, generally a full traceback

exception_type: The type of the exception, like `ValueError`

exception_value: The string value of the exception, like `'x not in list'`

identification_code: A hash of the exception data meant to identify the general exception, so that it shares this code with other exceptions that derive from the same problem. The code is a hash of all the module names and function names in the traceback, plus exception_type. This should be shown to users so they can refer to the exception later. (@@: should it include a portion that allows identification of the specific instance of the exception as well?)

The list of frames goes innermost first. Each frame has these attributes; some values may be None if they could not be determined.

modname: the name of the module

filename: the filename of the module

lineno: the line of the error

revision: the contents of __version__ or __revision__

name: the function name

supplement: an object created from `__traceback_supplement__`

supplement_exception: a simple traceback of any exception `__traceback_supplement__` created

traceback_info: the str() of any `__traceback_info__` variable found in the local scope (@@: should it str()-ify it or not?)

traceback_hide: the value of any `__traceback_hide__` variable

traceback_log: the value of any `__traceback_log__` variable

`__traceback_supplement__` is thrown away, but a fixed set of attributes are captured; each of these attributes is optional.

object: the name of the object being visited

source_url: the original URL requested

line: the line of source being executed (for interpreters, like ZPT)

column: the column of source being executed

expression: the expression being evaluated (also for interpreters)

warnings: a list of (string) warnings to be displayed

getInfo: a function/method that takes no arguments, and returns a string describing any extra information

extraData: a function/method that takes no arguments, and returns a dictionary. The contents of this dictionary will not be displayed in the context of the traceback, but globally for the exception. Results will be grouped by the keys in the dictionaries (which also serve as titles). The keys can also be tuples of (importance, title); in this case the importance should be `important` (shows up at top), `normal` (shows up somewhere; unspecified), `supplemental` (shows up at bottom), or `extra` (shows up hidden or not at all).

These are used to create an object with attributes of the same names (`getInfo` becomes a string attribute, not a method). `__traceback_supplement__` implementations should be careful to produce values that are relatively static and unlikely to cause further errors in the reporting system – any complex introspection should go in `getInfo()` and should ultimately return a string.

Note that all attributes are optional, and under certain circumstances may be None or may not exist at all – the collector can only do a best effort, but must avoid creating any exceptions itself.

Formatters may want to use `__traceback_hide__` as a hint to hide frames that are part of the 'framework' or underlying system. There are a variety of rules about special values for this variables that formatters should be aware of.

TODO:

More attributes in __traceback_supplement__? Maybe an attribute that gives a list of local variables that should also be collected? Also, attributes that would be explicitly meant for the entire request, not just a single frame. Right now some of the fixed set of attributes (e.g., source_url) are meant for this use, but there's no explicit way for the supplement to indicate new values, e.g., logged-in user, HTTP referrer, environment, etc. Also, the attributes that do exist are Zope/Web oriented.

More information on frames? cgitb, for instance, produces extensive information on local variables. There exists the possibility that getting this information may cause side effects, which can make debugging more difficult; but it also provides fodder for post-mortem debugging. However, the collector is not meant to be configurable, but to capture everything it can and let the formatters be configurable. Maybe this would have to be a configuration value, or maybe it could be indicated by another magical variable (which would probably mean 'show all local variables below this frame')

class weberror.collector.**ExceptionFrame**(***attrs*)
This represents one frame of the exception. Each frame is a context in the call stack, typically represented by a line number and module name in the traceback.

get_source_line(*context=0*)
Return the source of the current line of this frame. You probably want to .strip() it as well, as it is likely to have leading whitespace.

If context is given, then that many lines on either side will also be returned. E.g., context=1 will give 3 lines.

weberror.collector.**collect_exception**(*t*, *v*, *tb*, *limit=None*)
Collection an exception from `sys.exc_info()`.

Use like:

```
try:
    blah blah
except:
    exc_data = collect_exception(*sys.exc_info())
```

27.5 webhelpers – Web Helpers package

Warning: Pertinent to *WebHelpers 0.6*. **NB:** Significant changes from WebHelpers 0.3

WebHelpers is wide variety of functions for web applications and other applications.

27.5.1 Current

constants – Useful constants (Geo-lists)

webhelpers.constants

Place names and other constants often used in web forms.

webhelpers.constants.**uk_counties**()
Return a list of UK county names.

webhelpers.constants.**country_codes**()
Return a list of all country names as tuples. The tuple value is the country's 2-letter ISO code and its name; e.g., `("GB", "United Kingdom")`. The countries are in name order.

Can be used like this:

```
import webhelpers.constants as constants
from webhelpers.html.tags import select
select("country", country_codes(),
    prompt="Please choose a country ...")
```

See here for more information: http://www.iso.org/iso/english_country_names_and_code_elements

webhelpers.constants.**us_states**()
List of USA states.

Return a list of `(abbreviation, name)` for all US states, sorted by name. Includes the District of Columbia.

webhelpers.constants.**us_territories**()
USA postal abbreviations for territories, protectorates, and military.

The return value is a list of `(abbreviation, name)` tuples. The locations are sorted by name.

webhelpers.constants.**canada_provinces**()
List of Canadian provinces.

Return a list of `(abbreviation, name)` tuples for all Canadian provinces and territories, sorted by name.

containers Handy Containers

webhelpers.containers

class webhelpers.containers.**NotGiven**
A default value for function args.

Use this when you need to distinguish between `None` and no value.

Example:

```
>>> def foo(arg=NotGiven):
...     print arg is NotGiven
...
>>> foo()
True
>>> foo(None)
False
```

class webhelpers.containers.**DumbObject**(***kw*)
A container for arbitrary attributes.

Usage:

```
>>> do = DumbObject(a=1, b=2)
>>> do.b
2
```

Alternatives to this class include `collections.namedtuple` in Python 2.6, and `formencode.declarative.Declarative` in Ian Bicking's FormEncode package. Both alternatives offer more features, but `DumbObject` shines in its simplicity and lack of dependencies.

class webhelpers.containers.**Counter**
I count the number of occurrences of each value registered with me.

Call the instance to register a value. The result is available as the `.result` attribute. Example:

```
>>> counter = Counter()
>>> counter("foo")
>>> counter("bar")
>>> counter("foo")
>>> sorted(counter.result.items())
[('bar', 1), ('foo', 2)]

>> counter.result
{'foo': 2, 'bar': 1}
```

To see the most frequently-occurring items in order:

```
>>> counter.get_popular(1)
[(2, 'foo')]
>>> counter.get_popular()
[(2, 'foo'), (1, 'bar')]
```

Or if you prefer the list in item order:

```
>>> counter.get_sorted_items()
[('bar', 1), ('foo', 2)]
```

classmethod **correlate**(*class_*, *iterable*)
Build a Counter from an iterable in one step.

This is the same as adding each item individually.

```
>>> counter = Counter.correlate(["A", "B", "A"])
>>> counter.result["A"]
2
>>> counter.result["B"]
1
```

get_popular(*max_items=None*)
Return the results as as a list of `(count, item)` pairs, with the most frequently occurring items first.

If `max_items` is provided, return no more than that many items.

get_sorted_items()
Return the result as a list of `(item, count)` pairs sorted by item.

class webhelpers.containers.**Accumulator**
Accumulate a dict of all values for each key.

Call the instance to register a value. The result is available as the `.result` attribute. Example:

```
>>> bowling_scores = Accumulator()
>>> bowling_scores("Fred", 0)
>>> bowling_scores("Barney", 10)
>>> bowling_scores("Fred", 1)
>>> bowling_scores("Barney", 9)
>>> sorted(bowling_scores.result.items())
[('Barney', [10, 9]), ('Fred', [0, 1])]

>> bowling_scores.result
{'Fred': [0, 1], 'Barney': [10, 9]}
```

The values are stored in the order they're registered.

Alternatives to this class include `paste.util.multidict.MultiDict` in Ian Bicking's Paste package.

classmethod **correlate**(*class_*, *iterable*, *key*)
Create an Accumulator based on several related values.

`key` is a function to calculate the key for each item, akin to `list.sort(key=)`.

This is the same as adding each item individually.

class webhelpers.containers.**UniqueAccumulator**
Accumulate a dict of unique values for each key.

The values are stored in an unordered set.

Call the instance to register a value. The result is available as the `.result` attribute.

`webhelpers.containers.`**`unique`**(*it*)
Return a list of unique elements in the iterable, preserving the order.

Usage:

```
>>> unique([None, "spam", 2, "spam", "A", "spam", "spam", "eggs", "spam"])
[None, 'spam', 2, 'A', 'eggs']
```

`webhelpers.containers.`**`only_some_keys`**(*dic*, *keys*)
Return a copy of the dict with only the specified keys present.

`dic` may be any mapping. The return value is always a Python dict.

```
>> only_some_keys({"A": 1, "B": 2, "C": 3}, ["A", "C"])
>>> sorted(only_some_keys({"A": 1, "B": 2, "C": 3}, ["A", "C"]).items())
[('A', 1), ('C', 3)]
```

`webhelpers.containers.`**`except_keys`**(*dic*, *keys*)
Return a copy of the dict without the specified keys.

```
>>> except_keys({"A": 1, "B": 2, "C": 3}, ["A", "C"])
{'B': 2}
```

`webhelpers.containers.`**`extract_keys`**(*dic*, *keys*)
Return two copies of the dict. The first has only the keys specified. The second has all the *other* keys from the original dict.

```
>> extract_keys({"From": "F", "To": "T", "Received", R"}, ["To", "From"])
({"From": "F", "To": "T"}, {"Received": "R"})
>>> regular, extra = extract_keys({"From": "F", "To": "T", "Received": "R"}, ["To", "From"])
>>> sorted(regular.keys())
['From', 'To']
>>> sorted(extra.keys())
['Received']
```

`webhelpers.containers.`**`ordered_items`**(*dic*, *key_order*, *other_keys=True*, *default=<class 'webhelpers.misc.NotGiven'>*)
Like `dict.iteritems()` but with a specified key order.

Arguments:

- `dic` is any mapping.
- `key_order` is a list of keys. Items will be yielded in this order.
- `other_keys` is a boolean.
- `default` is a value returned if the key is not in the dict.

This yields the items listed in `key_order`. If a key does not exist in the dict, yield the default value if specified, otherwise skip the missing key. Afterwards, if `other_keys` is true, yield the remaining items in an arbitrary order.

Usage:

```
>>> dic = {"To": "you", "From": "me", "Date": "2008/1/4", "Subject": "X"}
>>> dic["received"] = "..."
>>> order = ["From", "To", "Subject"]
>>> list(ordered_items(dic, order, False))
[('From', 'me'), ('To', 'you'), ('Subject', 'X')]
```

webhelpers.containers.**del_quiet**(*dic*, *keys*)
Delete several keys from a dict, ignoring those that don't exist.

This modifies the dict in place.

```
>>> d ={"A": 1, "B": 2, "C": 3}
>>> del_quiet(d, ["A", "C"])
>>> d
{'B': 2}
```

webhelpers.containers.**correlate_dicts**(*dicts*, *key*)
Correlate several dicts under one superdict.

If you have several dicts each with a 'name' key, this puts them in a container dict keyed by name.

```
>>> d1 = {"name": "Fred", "age": 41}
>>> d2 = {"name": "Barney", "age": 31}
>>> flintstones = correlate_dicts([d1, d2], "name")
>>> sorted(flintstones.keys())
['Barney', 'Fred']
>>> flintstones["Fred"]["age"]
41
```

If you're having trouble spelling this method correctly, remember: "relate" has one 'l'. The 'r' is doubled because it occurs after a prefix. Thus "correlate".

webhelpers.containers.**correlate_objects**(*objects*, *attr*)
Correlate several objects under one dict.

If you have several objects each with a 'name' attribute, this puts them in a dict keyed by name.

```
>>> class Flintstone(DumbObject):
...     pass
...
>>> fred = Flintstone(name="Fred", age=41)
>>> barney = Flintstone(name="Barney", age=31)
>>> flintstones = correlate_objects([fred, barney], "name")
>>> sorted(flintstones.keys())
['Barney', 'Fred']
>>> flintstones["Barney"].age
31
```

If you're having trouble spelling this method correctly, remember: "relate" has one 'l'. The 'r' is doubled because it occurs after a prefix. Thus "correlate".

webhelpers.containers.**distribute**(*lis*, *columns*, *direction*, *fill=None*)
Distribute a list into a N-column table (list of lists).

`lis` is a list of values to distribute.

`columns` is an int greater than 1, specifying the number of columns in the table.

`direction` is a string beginning with "H" (horizontal) or "V" (vertical), case insensitive. This affects how values are distributed in the table, as described below.

`fill` is a value that will be placed in any remaining cells if the data runs out before the last row or column is completed. This must be an immutable value such as `None`, `""`, 0, " ", etc. If you use a mutable value like `[]` and later change any cell containing the fill value, all other cells containing the fill value will also be changed.

The return value is a list of lists, where each sublist represents a row in the table. `table[0]` is the first row. `table[0][0]` is the first column in the first row. `table[0][1]` is the second column in the first row.

This can be displayed in an HTML table via the following Mako template:

```
<table>
% for row in table:
  <tr>
% for cell in row:
    <td>${cell}</td>
% endfor   cell
  </tr>
% endfor   row
</table>
```

In a horizontal table, each row is filled before going on to the next row. This is the same as dividing the list into chunks:

```
>>> distribute([1, 2, 3, 4, 5, 6, 7, 8], 3, "H")
[[1, 2, 3], [4, 5, 6], [7, 8, None]]
```

In a vertical table, the first element of each sublist is filled before going on to the second element. This is useful for displaying an alphabetical list in columns, or when the entire column will be placed in a single <td> with a
 between each element:

```
>>> food = ["apple", "banana", "carrot", "daikon", "egg", "fish", "gelato", "honey"]
>>> table = distribute(food, 3, "V", "")
>>> table
[['apple', 'daikon', 'gelato'], ['banana', 'egg', 'honey'], ['carrot', 'fish', '']]
>>> for row in table:
...     for item in row:
...         print "%-9s" % item,
...     print "."    # To show where the line ends.
...
apple     daikon    gelato    .
banana    egg       honey     .
carrot    fish                .
```

Alternatives to this function include a NumPy matrix of objects.

`date` – Date helpers

`webhelpers.date`

webhelpers.date.**distance_of_time_in_words**(*from_time*, *to_time=0*, *granularity='second'*, *round=False*)

Return the absolute time-distance string for two datetime objects, ints or any combination you can dream of.

If times are integers, they are interpreted as seconds from now.

`granularity` dictates where the string calculation is stopped. If set to seconds (default) you will receive the full string. If another accuracy is supplied you will receive an approximation. Available granularities are: 'century', 'decade', 'year', 'month', 'day', 'hour', 'minute', 'second'

Setting `round` to true will increase the result by 1 if the fractional value is greater than 50% of the granularity unit.

Examples:

```
>>> distance_of_time_in_words(86399, round=True, granularity='day')
'1 day'
```

```
    >>> distance_of_time_in_words(86399, granularity='day')
    'less than 1 day'
    >>> distance_of_time_in_words(86399)
    '23 hours, 59 minutes and 59 seconds'
    >>> distance_of_time_in_words(datetime(2008,3,21, 16,34),
    ... datetime(2008,2,6,9,45))
    '1 month, 15 days, 6 hours and 49 minutes'
    >>> distance_of_time_in_words(datetime(2008,3,21, 16,34),
    ... datetime(2008,2,6,9,45), granularity='decade')
    'less than 1 decade'
    >>> distance_of_time_in_words(datetime(2008,3,21, 16,34),
    ... datetime(2008,2,6,9,45), granularity='second')
    '1 month, 15 days, 6 hours and 49 minutes'
```

webhelpers.date.**time_ago_in_words**(*from_time, granularity='second', round=False*)

Return approximate-time-distance string for `from_time` till now.

Same as `distance_of_time_in_words` but the endpoint is now.

`feedgenerator` – Feed generator

The feed generator is intended for use in controllers, and generates an output stream. Currently the following feeds can be created by imported the appropriate class:

- RssFeed
- RssUserland091Feed
- Rss201rev2Feed
- Atom1Feed

All of these format specific Feed generators inherit from the `SyndicationFeed()` class.

Example controller method:

```
import logging

from pylons import request, response, session
from pylons import tmpl_context as c
from pylons.controllers.util import abort, redirect_to, url_for
from webhelpers.feedgenerator import Atom1Feed

from helloworld.lib.base import BaseController, render

log = logging.getLogger(__name__)

class CommentsController(BaseController):

    def index(self):
        feed = Atom1Feed(
            title=u"An excellent Sample Feed",
            link=url_for(),
            description=u"A sample feed, showing how to make and add entries",
            language=u"en",
        )
        feed.add_item(title="Sample post",
                      link=u"http://hellosite.com/posts/sample",
                      description="Testing.")
```

```
        response.content_type = 'application/atom+xml'
        return feed.writeString('utf-8')
```

Module Contents

class webhelpers.feedgenerator.**SyndicationFeed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

Base class for all syndication feeds. Subclasses should provide write()

__init__(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

add_item(*title*, *link*, *description*, *author_email=None*, *author_name=None*, *author_link=None*, *pubdate=None*, *comments=None*, *unique_id=None*, *enclosure=None*, *categories=()*, *item_copyright=None*, *ttl=None*, ***kwargs*)

Adds an item to the feed. All args are expected to be Python Unicode objects except pubdate, which is a datetime.datetime object, and enclosure, which is an instance of the Enclosure class.

add_item_elements(*handler*, *item*)

Add elements on each item (i.e. item/entry) element.

add_root_elements(*handler*)

Add elements in the root (i.e. feed/channel) element. Called from write().

item_attributes(*item*)

Return extra attributes to place on each item (i.e. item/entry) element.

latest_post_date()

Returns the latest item's pubdate. If none of them have a pubdate, this returns the current date/time.

root_attributes()

Return extra attributes to place on the root (i.e. feed/channel) element. Called from write().

write(*outfile*, *encoding*)

Outputs the feed in the given encoding to outfile, which is a file-like object. Subclasses should override this.

writeString(*encoding*)

Returns the feed in the given encoding as a string.

class webhelpers.feedgenerator.**Enclosure**(*url*, *length*, *mime_type*)

Represents an RSS enclosure

class webhelpers.feedgenerator.**RssFeed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

class webhelpers.feedgenerator.**RssUserland091Feed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

class webhelpers.feedgenerator.**Rss201rev2Feed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

class webhelpers.feedgenerator.**Atom1Feed**(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)

webhelpers.feedgenerator.**rfc2822_date**(*date*)

webhelpers.feedgenerator.**rfc3339_date**(*date*)

webhelpers.feedgenerator.**get_tag_uri**(*url*, *date*)

Creates a TagURI. See http://diveintomark.org/archives/2004/05/28/howto-atom-id

webhelpers.html – HTML handling

webhelpers.html.builder

HTML/XHTML tag builder

HTML Builder provides an `HTML` object that creates (X)HTML tags in a Pythonic way, a `literal` class used to mark strings containing intentional HTML markup, and a smart `escape()` function that preserves literals but escapes other strings that may accidentally contain markup characters ("<", ">", "&") or malicious Javascript tags. Escaped strings are returned as literals to prevent them from being double-escaped later.

`literal` is a subclass of `unicode`, so it works with all string methods and expressions. The only thing special about it is the `.__html__` method, which returns the string itself. `escape()` follows a simple protocol: if the object has an `.__html__` method, it calls that rather than `.__str__` to get the HTML representation. Third-party libraries that do not want to import `literal` (and this create a dependency on WebHelpers) can put an `.__html__` method in their own classes returning the desired HTML representation.

When used in a mixed expression containing both literals and ordinary strings, `literal` tries hard to escape the strings and return a literal. However, this depends on which value has "control" of the expression. `literal` seems to be able to take control with all combinations of the `+` operator, but with `%` and `join` it must be on the left side of the expression. So these all work:

```
"A" + literal("B")
literal(", ").join(["A", literal("B")])
literal("%s %s") % (16, literal("kg"))
```

But these return an ordinary string which is prone to double-escaping later:

```
"\\n".join([literal('<span class="foo">Foo!</span>'), literal('Bar!')])
"%s %s" % (literal("16"), literal("&lt;em&gt;kg&lt;/em&gt;"))
```

Third-party libraries that don't want to import `literal` and thus avoid a dependency on WebHelpers can add an `.__html__` method to any class, which can return the same as `.__str__` or something else. `escape()` trusts the HTML method and does not escape the return value. So only strings that lack an `.__html__` method will be escaped.

The `HTML` object has the following methods for tag building:

HTML(*strings) Escape the string args, concatenate them, and return a literal. This is the same as `escape(s)` but accepts multiple strings. Multiple args are useful when mixing child tags with text, such as:

```
html = HTML("The king is a >>", HTML.strong("fink"), "<<!")
```

HTML.literal(*strings) Same as `literal` but concatenates multiple arguments.

HTML.comment(*strings) Escape and concatenate the strings, and wrap the result in an HTML comment.

HTML.tag(tag, *content, **attrs) Create an HTML tag `tag` with the keyword args converted to attributes. The other positional args become the content for the tag, and are escaped and concatenated. If an attribute name conflicts with a Python keyword (notably "class"), append an underscore. If an attribute value is `None`, the attribute is not inserted. Two special keyword args are recognized:

c Specifies the content. This cannot be combined with content in positional args. The purpose of this argument is to position the content at the end of the argument list to match the native HTML syntax more closely. Its use is entirely optional. The value can be a string, a tuple, or a tag.

_close If present and false, do not close the tag. Otherwise the tag will be closed with a closing tag or an XHTML-style trailing slash as described below.

Example:

```
>>> HTML.tag("a", href="http://www.yahoo.com", name=None,
... c="Click Here")
literal(u'<a href="http://www.yahoo.com">Click Here</a>')
```

HTML.__getattr__ Same as `HTML.tag` but using attribute access. Example:

```
>>> HTML.a("Foo", href="http://example.com/", class_="important")
literal(u'<a class="important" href="http://example.com/">Foo</a>')
```

The protocol is simple: if an object has an `.__html__` method, `escape()` calls it rather than `.__str__()` to obtain a string representation.

About XHTML and HTML

This builder always produces tags that are valid as *both* HTML and XHTML. "Empty" tags (like `<br>`, `<input>` etc) are written like `<br />`, with a space and a trailing `/`.

Only empty tags get this treatment. The library will never, for example, product `<script src="..." />`, which is invalid HTML.

The W3C HTML validator validates these constructs as valid HTML Strict. It does produce warnings, but those warnings warn about the ambiguity if this same XML-style self-closing tags are used for HTML elements that can take content (`<script>`, `<textarea>`, etc). This library never produces markup like that.

Rather than add options to generate different kinds of behavior, we felt it was better to create markup that could be used in different contexts without any real problems and without the overhead of passing

options around or maintaining different contexts, where you'd have to keep track of whether markup is being rendered in an HTML or XHTML context.

If you _really_ want tags without training slashes (e.g., `<br>`')`', you can "abuse" ``'_close=False` to produce them.

class webhelpers.html.builder.**UnfinishedTag**(*tag*)
Represents an unfinished or empty tag.

class webhelpers.html.builder.**UnfinishedComment**
Represents an unfinished or empty comment.

class webhelpers.html.builder.**UnfinishedLiteral**
Represent an unfinished literal value.

class webhelpers.html.builder.**HTMLBuilder**
Base HTML object.

webhelpers.html.builder.**make_tag**(*tag*, **args*, ***kw*)

webhelpers.html.builder.**literal**()
Represents an HTML literal.

This subclass of unicode has a `.__html__()` method that is detected by the `escape()` function.

Also, if you add another string to this string, the other string will be quoted and you will get back another literal object. Also `literal(...) % obj` will quote any value(s) from `obj`. If you do something like `literal(...) + literal(...)`, neither string will be changed because `escape(literal(...))` doesn't change the original literal.

webhelpers.html.builder.**lit_sub**(**args*, ***kw*)
Literal-safe version of re.sub. If the string to be operated on is a literal, return a literal result. All arguments are passed directly to `re.sub`.

webhelpers.html.builder.**escape**()
escape_silent(s) -> markup

Like escape but converts None to an empty string.

webhelpers.html.converters

webhelpers.html.converters.**markdown**(*text*, *markdown=None*, ***kwargs*)
Format the text to HTML with Markdown formatting.

Markdown is a wiki-like text markup language, originally written by John Gruber for Perl. The helper converts Markdown text to HTML.

There are at least two Python implementations of Markdown. Markdown <http://www.freewisdom.org/projects/python-markdown/>'_is the original port, and version 2.x contains extensions for footnotes, RSS, etc. Markdown2 is another port which claims to be faster and to handle edge cases better.

You can pass the desired Markdown module as the `markdown` argument, or the helper will try to import `markdown`. If neither is available, it will fall back to `webhelpers.markdown`, which is Freewisdom's Markdown 1.7 without extensions.

IMPORTANT: If your source text is untrusted and may contain malicious HTML markup, pass `safe_mode="escape"` to escape it, `safe_mode="replace"` to replace it with a scolding message, or `safe_mode="remove"` to strip it.

webhelpers.html.converters.**textilize**(*text*, *sanitize=False*)
Format the text to HTML with Textile formatting.

This function uses the PyTextile library which is included with WebHelpers.

Additionally, the output can be sanitized which will fix tags like <img />,
 and <hr /> for proper XHTML output.

webhelpers.html.secure_form

webhelpers.html.tags

webhelpers.html.tags.**form**(*url*, *method='post'*, *multipart=False*, *hidden_fields=None*, ***attrs*)
An open tag for a form that will submit to `url`.

You must close the form yourself by calling `end_form()` or outputting </form>.

Options:

method The method to use when submitting the form, usually either "GET" or "POST". If "PUT", "DELETE", or another verb is used, a hidden input with name _method is added to simulate the verb over POST.

multipart If set to True, the enctype is set to "multipart/form-data". You must set it to true when uploading files, or the browser will submit the filename rather than the file.

hidden_fields Additional hidden fields to add to the beginning of the form. It may be a dict or an iterable of key-value tuples. This is implemented by calling the object's `.items()` method if it has one, or just iterating the object. (This will successfuly get multiple values for the same key in WebOb MultiDict objects.)

Because input tags must be placed in a block tag rather than directly inside the form, all hidden fields will be put in a ' '. The style prevents the from being displayed or affecting the layout.

Examples:

```
>>> form("/submit")
literal(u'<form action="/submit" method="post">')
>>> form("/submit", method="get")
literal(u'<form action="/submit" method="get">')
>>> form("/submit", method="put")
literal(u'<form action="/submit" method="post"><div style="display:none">\n<input name="_method"
>>> form("/submit", "post", multipart=True)
literal(u'<form action="/submit" enctype="multipart/form-data" method="post">')
```

Changed in WebHelpers 1.0b2: add and `hidden_fields` arg.

Changed in WebHelpers 1.2: don't add an "id" attribute to hidden tags generated by this helper; they clash if there are multiple forms on the page.

webhelpers.html.tags.**end_form**()
Output "</form>".

Example:

```
>>> end_form()
literal(u'</form>')
```

webhelpers.html.tags.**text**(*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, *type='text'*, ***attrs*)

Create a standard text field.

`value` is a string, the content of the text field.

`id` is the HTML ID attribute, and should be passed as a keyword argument. By default the ID is the same as the name filtered through `_make_safe_id_component()`. Pass None to suppress the ID attribute entirely.

`type` is the input field type, normally "text". You can override it for HTML 5 input fields that don't have their own helper; e.g., "search", "email", "date".

Options:

- **`disabled` - If set to True, the user will not be able to use** this input.
- **`size` - The number of visible characters that will fit in the** input.
- **`maxlength` - The maximum number of characters that the browser** will allow the user to enter.

The remaining keyword args will be standard HTML attributes for the tag.

Example, a text input field:

```
>>> text("address")
literal(u'<input id="address" name="address" type="text" />')
```

HTML 5 example, a color picker:

```
>>> text("color", type="color")
literal(u'<input id="color" name="color" type="color" />')
```

webhelpers.html.tags.**hidden**(*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a hidden field.

webhelpers.html.tags.**file**(*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a file upload field.

If you are using file uploads then you will also need to set the multipart option for the form.

Example:

```
>>> file('myfile')
literal(u'<input id="myfile" name="myfile" type="file" />')
```

webhelpers.html.tags.**password**(*name*, *value=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a password field.

Takes the same options as `text()`.

webhelpers.html.tags.**textarea**(*name*, *content=''*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a text input area.

Example:

```
>>> textarea("body", "", cols=25, rows=10)
literal(u'<textarea cols="25" id="body" name="body" rows="10"></textarea>')
```

webhelpers.html.tags.**checkbox**(*name*, *value='1'*, *checked=False*, *label=None*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a check box.

Arguments: `name` – the widget's name.

`value` – the value to return to the application if the box is checked.

`checked` – true if the box should be initially checked.

`label` – a text label to display to the right of the box.

`id` is the HTML ID attribute, and should be passed as a keyword argument. By default the ID is the same as the name filtered through `_make_safe_id_component()`. Pass None to suppress the ID attribute entirely.

The following HTML attributes may be set by keyword argument:

- `disabled` - If true, checkbox will be grayed out.
- `readonly` - If true, the user will not be able to modify the checkbox.

To arrange multiple checkboxes in a group, see webhelpers.containers.distribute().

Example:

```
>>> checkbox("hi")
literal(u'<input id="hi" name="hi" type="checkbox" value="1" />')
```

webhelpers.html.tags.**_make_safe_id_component**(*idstring*)

Make a string safe for including in an id attribute.

The HTML spec says that id attributes 'must begin with a letter ([A-Za-z]) and may be followed by any number of letters, digits ([0-9]), hyphens ("-"), underscores ("_"), colons (":"), and periods (".")'. These regexps are slightly over-zealous, in that they remove colons and periods unnecessarily.

Whitespace is transformed into underscores, and then anything which is not a hyphen or a character that matches w (alphanumerics and underscore) is removed.

webhelpers.html.tags.**radio**(*name*, *value*, *checked=False*, *label=None*, ***attrs*)

Create a radio button.

Arguments: `name` – the field's name.

`value` – the value returned to the application if the button is pressed.

`checked` – true if the button should be initially pressed.

`label` – a text label to display to the right of the button.

The id of the radio button will be set to the name + '_' + value to ensure its uniqueness. An `id` keyword arg overrides this. (Note that this behavior is unique to the `radio()` helper.)

To arrange multiple radio buttons in a group, see webhelpers.containers.distribute().

webhelpers.html.tags.**submit**(*name*, *value*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a submit button with the text `value` as the caption.

webhelpers.html.tags.**select**(*name*, *selected_values*, *options*, *id=<class 'webhelpers.misc.NotGiven'>*, ***attrs*)

Create a dropdown selection box.

- `name` – the name of this control.
- `selected_values` – a string or list of strings or integers giving the value(s) that should be preselected.

- `options` – an `Options` object or iterable of `(value, label)` pairs. The label will be shown on the form; the option will be returned to the application if that option is chosen. If you pass a string or int instead of a 2-tuple, it will be used for both the value and the label. If the *value* is a tuple or a list, it will be added as an optgroup, with *label* as label.

`id` is the HTML ID attribute, and should be passed as a keyword argument. By default the ID is the same as the name. filtered through `_make_safe_id_component()`. Pass None to suppress the ID attribute entirely.

CAUTION: the old rails helper `options_for_select` had the label first. The order was reversed because most real-life collections have the value first, including dicts of the form `{value: label}`. For those dicts you can simply pass `D.items()` as this argument.

HINT: You can sort options alphabetically by label via: `sorted(my_options, key=lambda x: x[1])`

The following options may only be keyword arguments:

- **`multiple` – if true, this control will allow multiple** selections.
- `prompt` – if specified, an extra option will be prepended to the list: ("", `prompt`). This is intended for those "Please choose ..." pseudo-options. Its value is "", equivalent to not making a selection.

Any other keyword args will become HTML attributes for the <select>.

Examples (call, result):

```
>>> select("currency", "$", [["$", "Dollar"], ["DKK", "Kroner"]])
literal(u'<select id="currency" name="currency">\n<option selected="selected" value="$">Dollar</
>>> select("cc", "MasterCard", [ "VISA", "MasterCard" ], id="cc", class_="blue")
literal(u'<select class="blue" id="cc" name="cc">\n<option value="VISA">VISA</option>\n<option s
>>> select("cc", ["VISA", "Discover"], [ "VISA", "MasterCard", "Discover" ])
literal(u'<select id="cc" name="cc">\n<option selected="selected" value="VISA">VISA</option>\n<c
>>> select("currency", None, [["$", "Dollar"], ["DKK", "Kroner"]], prompt="Please choose ...")
literal(u'<select id="currency" name="currency">\n<option selected="selected" value="">Please ch
>>> select("privacy", 3L, [(1, "Private"), (2, "Semi-public"), (3, "Public")])
literal(u'<select id="privacy" name="privacy">\n<option value="1">Private</option>\n<option valu
>>> select("recipients", None, [(["u1", "User1"), ("u2", "User2")], "Users"), ([("g1", "Group1"
literal(u'<select id="recipients" name="recipients">\n<optgroup label="Users">\n<option value="u
```

class `webhelpers.html.tags.`**`ModelTags`**(*record*, *use_keys=False*, *date_format='%m/%d/%Y'*, *id_format=None*)

A nice way to build a form for a database record.

ModelTags allows you to build a create/update form easily. (This is the C and U in CRUD.) The constructor takes a database record, which can be a SQLAlchemy mapped class, or any object with attributes or keys for the field values. Its methods shadow the the form field helpers, but it automatically fills in the value attribute based on the current value in the record. (It also knows about the 'checked' and 'selected' attributes for certain tags.)

You can also use the same form to input a new record. Pass `None` or `""` instead of a record, and it will set all the current values to a default value, which is either the *default* keyword arg to the method, or "" if not specified.

(Hint: in Pylons you can put `mt = ModelTags(c.record)` in your template, and then if the record doesn't exist you can either set `c.record = None` or not set it at all. That's because nonexistent `c` attributes resolve to "" unless you've set `config["pylons.strict_c"] = True`. However, having a `c` attribute that's sometimes set and sometimes not is arguably bad programming style.)

`checkbox`(*name*, *value='1'*, *label=None*, ***kw*)

Build a checkbox field.

The box will be initially checked if the value of the corresponding database field is true.

The submitted form value will be "1" if the box was checked. If the box is unchecked, no value will be submitted. (This is a downside of the standard checkbox tag.)

To display multiple checkboxes in a group, see webhelper.containers.distribute().

date(*name*, ***kw*)
Same as text but format a date value into a date string.

The value can be a *datetime.date*, *datetime.datetime*, *None*, or "". The former two are converted to a string using the date format passed to the constructor. The latter two are converted to "".

If there's no database record, consult keyword arg *default*. It it's the string "today", use todays's date. Otherwise it can be any of the values allowed above. If no default is specified, the text field is initialized to "".

Hint: you may wish to attach a Javascript calendar to the field.

file(*name*, ***kw*)
Build a file upload field.

User agents may or may not respect the contents of the 'value' attribute.

hidden(*name*, ***kw*)
Build a hidden HTML field.

password(*name*, ***kw*)
Build a password field.

This is the same as a text box but the value will not be shown on the screen as the user types.

radio(*name*, *checked_value*, *label=None*, ***kw*)
Build a radio button.

The radio button will initially be selected if the database value equals `checked_value`. On form submission the value will be `checked_value` if the button was selected, or `""` otherwise.

In case of a ModelTags object that is created from scratch (e.g. `new_employee=ModelTags(None)`) the option that should be checked can be set by the 'default' parameter. As in: `new_employee.radio('status', checked_value=7, default=7)`

The control's 'id' attribute will be modified as follows:

1. If not specified but an 'id_format' was given to the constructor, generate an ID based on the format.
2. If an ID was passed in or was generated by step (1), append an underscore and the checked value. Before appending the checked value, lowercase it, change any spaces to `"_"`, and remove any non-alphanumeric characters except underscores and hyphens.
3. If no ID was passed or generated by step (1), the radio button will not have an 'id' attribute.

To display multiple radio buttons in a group, see webhelper.containers.distribute().

select(*name*, *options*, ***kw*)
Build a dropdown select box or list box.

See the `select()` function for the meaning of the arguments.

If the corresponding database value is not a list or tuple, it's wrapped in a one-element list. But if it's "" or `None`, an empty list is substituted. This is to accommodate multiselect lists, which may have multiple values selected.

text(*name*, ***kw*)
Build a text box.

textarea(*name*, ***kw*)
Build a rectangular text area.

webhelpers.html.tags.**link_to**(*label*, *url=''*, ***attrs*)
Create a hyperlink with the given text pointing to the URL.

If the label is `None` or empty, the URL will be used as the label.

This function does not modify the URL in any way. The label will be escaped if it contains HTML markup. To prevent escaping, wrap the label in a `webhelpers.html.literal()`.

webhelpers.html.tags.**link_to_if**(*condition*, *label*, *url=''*, ***attrs*)
Same as `link_to` but return just the label if the condition is false.

This is useful in a menu when you don't want the current option to be a link. The condition will be something like: `actual_value != value_of_this_menu_item`.

webhelpers.html.tags.**link_to_unless**(*condition*, *label*, *url=''*, ***attrs*)
The opposite of `link_to`. Return just the label if the condition is true.

webhelpers.html.tags.**th_sortable**(*current_order*, *column_order*, *label*, *url*, *class_if_sort_column='sort'*, *class_if_not_sort_column=None*, *link_attrs=None*, *name='th'*, ***attrs*)
<th> for a "click-to-sort-by" column.

Convenience function for a sortable column. If this is the current sort column, just display the label and set the cell's class to `class_if_sort_column`.

`current_order` is the table's current sort order. `column_order` is the value pertaining to this column. In other words, if the two are equal, the table is currently sorted by this column.

If this is the sort column, display the label and set the <th>'s class to `class_if_sort_column`.

If this is not the sort column, display an <a> hyperlink based on `label`, `url`, and `link_attrs` (a dict), and set the <th>'s class to `class_if_not_sort_column`.

`url` is the literal href= value for the link. Pylons users would typically pass something like `url=h.url_for("mypage", sort="date")`.

`**attrs` are additional attributes for the <th> tag.

If you prefer a <td> tag instead of <th>, pass `name="td"`.

To change the sort order via client-side Javascript, pass `url=None` and the appropriate Javascript attributes in `link_attrs`.

Examples:

```
>>> sort = "name"
>>> th_sortable(sort, "name", "Name", "?sort=name")
literal(u'<th class="sort">Name</th>')
>>> th_sortable(sort, "date", "Date", "?sort=date")
literal(u'<th><a href="?sort=date">Date</a></th>')
>>> th_sortable(sort, "date", "Date", None, link_attrs={"onclick": "myfunc()"})
literal(u'<th><a onclick="myfunc()">Date</a></th>')
```

webhelpers.html.tags.**image**(*url*, *alt*, *width=None*, *height=None*, *path=None*, *use_pil=False*, ***attrs*)
Return an image tag for the specified `source`.

url The URL of the image. (This must be the exact URL desired. A previous version of this helper added magic prefixes; this is no longer the case.)

alt The img's alt tag. Non-graphical browsers and screen readers will output this instead of the image. If the image is pure decoration and uninteresting to non-graphical users, pass "". To omit the alt tag completely, pass None.

width The width of the image, default is not included.

height The height of the image, default is not included.

path Calculate the width and height based on the image file at `path` if possible. May not be specified if `width` or `height` is specified. The results are also written to the debug log for troubleshooting.

use_pil If true, calcuate the image dimensions using the Python Imaging Library, which must be installed. Otherwise use a pure Python algorithm which understands fewer image formats and may be less accurate. This flag controls whether `webhelpers.media.get_dimensions_pil` or `webhelpers.media.get_dimensions` is called. It has no effect if `path` is not specified.

Examples:

```
>>> image('/images/rss.png', 'rss syndication')
literal(u'<img alt="rss syndication" src="/images/rss.png" />')

>>> image('/images/xml.png', "")
literal(u'<img alt="" src="/images/xml.png" />')

>>> image("/images/icon.png", height=16, width=10, alt="Edit Entry")
literal(u'<img alt="Edit Entry" height="16" src="/images/icon.png" width="10" />')

>>> image("/icons/icon.gif", alt="Icon", width=16, height=16)
literal(u'<img alt="Icon" height="16" src="/icons/icon.gif" width="16" />')

>>> image("/icons/icon.gif", None, width=16)
literal(u'<img alt="" src="/icons/icon.gif" width="16" />')
```

webhelpers.html.tags.**javascript_link**(**urls*, ***attrs*)

Return script include tags for the specified javascript URLs.

`urls` should be the exact URLs desired. A previous version of this helper added magic prefixes; this is no longer the case.

Specify the keyword argument `defer=True` to enable the script defer attribute.

Examples:

```
>>> print javascript_link('/javascripts/prototype.js', '/other-javascripts/util.js')
<script src="/javascripts/prototype.js" type="text/javascript"></script>
<script src="/other-javascripts/util.js" type="text/javascript"></script>

>>> print javascript_link('/app.js', '/test/test.1.js')
<script src="/app.js" type="text/javascript"></script>
<script src="/test/test.1.js" type="text/javascript"></script>
```

webhelpers.html.tags.**stylesheet_link**(**urls*, ***attrs*)

Return CSS link tags for the specified stylesheet URLs.

`urls` should be the exact URLs desired. A previous version of this helper added magic prefixes; this is no longer the case.

Examples:

```
>>> stylesheet_link('/stylesheets/style.css')
literal(u'<link href="/stylesheets/style.css" media="screen" rel="stylesheet" type="text/css" />

>>> stylesheet_link('/stylesheets/dir/file.css', media='all')
literal(u'<link href="/stylesheets/dir/file.css" media="all" rel="stylesheet" type="text/css" />
```

webhelpers.html.tags.**auto_discovery_link**(*url*, *feed_type='rss'*, ***attrs*)

Return a link tag allowing auto-detecting of RSS or ATOM feed.

The auto-detection of feed for the current page is only for browsers and news readers that support it.

url The URL of the feed. (This should be the exact URLs desired. A previous version of this helper added magic prefixes; this is no longer the case.)

feed_type The type of feed. Specifying 'rss' or 'atom' automatically translates to a type of 'application/rss+xml' or 'application/atom+xml', respectively. Otherwise the type is used as specified. Defaults to 'rss'.

Examples:

```
>>> auto_discovery_link('http://feed.com/feed.xml')
literal(u'<link href="http://feed.com/feed.xml" rel="alternate" title="RSS" type="application/rs

>>> auto_discovery_link('http://feed.com/feed.xml', feed_type='atom')
literal(u'<link href="http://feed.com/feed.xml" rel="alternate" title="ATOM" type="application/a

>>> auto_discovery_link('app.rss', feed_type='atom', title='atom feed')
literal(u'<link href="app.rss" rel="alternate" title="atom feed" type="application/atom+xml" />'

>>> auto_discovery_link('/app.html', feed_type='text/html')
literal(u'<link href="/app.html" rel="alternate" title="" type="text/html" />')
```

webhelpers.html.tools

webhelpers.html.tools.**button_to**(*name*, *url=''*, ***html_attrs*)

Generate a form containing a sole button that submits to `url`.

Use this method instead of `link_to` for actions that do not have the safe HTTP GET semantics implied by using a hypertext link.

The parameters are the same as for `link_to`. Any `html_attrs` that you pass will be applied to the inner `input` element. In particular, pass

disabled = True/False

as part of `html_attrs` to control whether the button is disabled. The generated form element is given the class 'button-to', to which you can attach CSS styles for display purposes.

The submit button itself will be displayed as an image if you provide both `type` and `src` as followed:

type='image', src='icon_delete.gif'

The `src` path should be the exact URL desired. A previous version of this helper added magical prefixes but this is no longer the case.

Example 1:

```
# inside of controller for "feeds"
>> button_to("Edit", url(action='edit', id=3))
<form method="post" action="/feeds/edit/3" class="button-to">
```

```
<div><input value="Edit" type="submit" /></div>
</form>
```

Example 2:

```
>> button_to("Destroy", url(action='destroy', id=3),
.. method='DELETE')
<form method="POST" action="/feeds/destroy/3"
 class="button-to">
<div>
    <input type="hidden" name="_method" value="DELETE" />
    <input value="Destroy" type="submit" />
</div>
</form>
```

Example 3:

```
# Button as an image.
>> button_to("Edit", url(action='edit', id=3), type='image',
.. src='icon_delete.gif')
<form method="POST" action="/feeds/edit/3" class="button-to">
<div><input alt="Edit" src="/images/icon_delete.gif"
 type="image" value="Edit" /></div>
</form>
```

Note: This method generates HTML code that represents a form. Forms are "block" content, which means that you should not try to insert them into your HTML where only inline content is expected. For example, you can legally insert a form inside of a `div` or `td` element or in between `p` elements, but not in the middle of a run of text, nor can you place a form within another form. (Bottom line: Always validate your HTML before going public.)

Changed in WebHelpers 1.2: Preserve case of "method" arg for XHTML compatibility. E.g., "POST" or "PUT" causes *method="POST"*; "post" or "put" causes *method="post"*.

`webhelpers.html.tools.`**`mail_to`**(*email_address*, *name=None*, *cc=None*, *bcc=None*, *subject=None*, *body=None*, *replace_at=None*, *replace_dot=None*, *encode=None*, ***html_attrs*)

Create a link tag for starting an email to the specified `email_address`.

This `email_address` is also used as the name of the link unless `name` is specified. Additional HTML options, such as class or id, can be passed in the `html_attrs` hash.

You can also make it difficult for spiders to harvest email address by obfuscating them.

Examples:

```
>>> mail_to("me@domain.com", "My email", encode = "javascript")
literal(u'<script type="text/javascript">\n//<![CDATA[\neval(unescape(\'%64%6f%63%75%6d%65%6e%74

>>> mail_to("me@domain.com", "My email", encode = "hex")
literal(u'<a href="&#109;&#97;&#105;&#108;&#116;&#111;&#58;%6d%65@%64%6f%6d%61%69%6e.%63%6f%6d">
```

You can also specify the cc address, bcc address, subject, and body parts of the message header to create a complex e-mail using the corresponding `cc`, `bcc`, `subject`, and `body` keyword arguments. Each of these options are URI escaped and then appended to the `email_address` before being output. **Be aware that javascript keywords will not be escaped and may break this feature when encoding with javascript.**

Examples:

```
>>> mail_to("me@domain.com", "My email", cc="ccaddress@domain.com", bcc="bccaddress@domain.com",
literal(u'<a href="mailto:me@domain.com?cc=ccaddress%40domain.com&amp;bcc=bccaddress%40domain.cc
```

webhelpers.html.tools.**highlight**(*text*, *phrase*, *highlighter=None*, *case_sensitive=False*, *class_='highlight'*, ***attrs*)

Highlight all occurrences of `phrase` in `text`.

This inserts "..." around every occurrence.

Arguments:

text: The full text.

phrase: A phrase to find in the text. This may be a string, a list of strings, or a compiled regular expression. If a string, it's regex-escaped and compiled. If a list, all of the strings will be highlighted. This is done by regex-escaping all elements and then joining them using the regex "|" token.

highlighter: Deprecated. A replacement expression for the regex substitution. This was deprecated because it bypasses the HTML builder and creates tags via string mangling. The previous default was '1', which mimics the normal behavior of this function. `phrase` must be a string if `highlighter` is specified. Overrides `class_` and `attrs_` arguments.

case_sensitive: If false (default), the phrases are searched in a case-insensitive manner. No effect if `phrase` is a regex object.

class_: CSS class for the tag.

****attrs:** Additional HTML attributes for the tag.

Changed in WebHelpers 1.0b2: new implementation using HTML builder. Allow `phrase` to be list or regex. Deprecate `highlighter` and change its default value to None. Add `case_sensitive`, `class_`, and `**attrs` arguments.

webhelpers.html.tools.**auto_link**(*text*, *link='all'*, ***href_attrs*)

Turn all urls and email addresses into clickable links.

link Used to determine what to link. Options are "all", "email_addresses", or "urls"

href_attrs Additional attributes for generated <a> tags.

Example:

```
>>> auto_link("Go to http://www.planetpython.com and say hello to guido@python.org")
literal(u'Go to <a href="http://www.planetpython.com">http://www.planetpython.com</a> and say he
```

webhelpers.html.tools.**strip_links**(*text*)

Strip link tags from `text` leaving just the link label.

Example:

```
>>> strip_links('<a href="something">else</a>')
'else'
```

webhelpers.markdown – Markdown

webhelpers.markdown

webhelpers.markdown.**markdown**(*text*, *extensions=[]*, *safe_mode=False*)

webhelpers.markdown.**markdownFromFile**(*input=None, output=None, extensions=[], encoding=None, message_threshold=50, safe=False*)

class webhelpers.markdown.**Markdown**(*source=None, extensions=[], extension_configs=None, safe_mode=False*)

Markdown formatter class for creating an html document from Markdown text

convert(*source=None*)

Return the document in XHTML format.

@returns: A serialized XHTML body.

registerExtension(*extension*)

This gets called by the extension

reset()

Resets all state variables so that we can start with a new text.

class webhelpers.markdown.**Document**

class webhelpers.markdown.**Element**(*tag*)

find(*test, depth=0*)

Returns a list of descendants that pass the test function

class webhelpers.markdown.**TextNode**(*text*)

class webhelpers.markdown.**EntityReference**(*entity*)

class webhelpers.markdown.**HeaderPreprocessor**

Replaces underlined headers with hashed headers to avoid the need for lookahead later.

class webhelpers.markdown.**LinePreprocessor**

Deals with HR lines (needs to be done before processing lists)

class webhelpers.markdown.**HtmlBlockPreprocessor**

Removes html blocks from the source text and stores it.

class webhelpers.markdown.**ReferencePreprocessor**

Removes reference definitions from the text and stores them for later use.

class webhelpers.markdown.**Pattern**(*pattern*)

class webhelpers.markdown.**SimpleTextPattern**(*pattern*)

class webhelpers.markdown.**SimpleTagPattern**(*pattern, tag*)

class webhelpers.markdown.**BacktickPattern**(*pattern*)

class webhelpers.markdown.**DoubleTagPattern**(*pattern, tag*)

class webhelpers.markdown.**HtmlPattern**(*pattern*)

class webhelpers.markdown.**LinkPattern**(*pattern*)

class webhelpers.markdown.**ImagePattern**(*pattern*)

class webhelpers.markdown.**ReferencePattern**(*pattern*)

class webhelpers.markdown.**ImageReferencePattern**(*pattern*)

class webhelpers.markdown.**AutolinkPattern**(*pattern*)

class webhelpers.markdown.**AutomailPattern**(*pattern*)

class webhelpers.markdown.**Postprocessor**
Postprocessors are run before the dom it converted back into text.

Each Postprocessor implements a "run" method that takes a pointer to a NanoDom document, modifies it as necessary and returns a NanoDom document.

Postprocessors must extend markdown.Postprocessor.

There are currently no standard post-processors, but the footnote extension uses one.

class webhelpers.markdown.**HtmlStash**
This class is used for stashing HTML objects that we extract in the beginning and replace with placeholders.

store(*html*, *safe=False*)
Saves an HTML segment for later reinsertion. Returns a placeholder string that needs to be inserted into the document.

@param html: an html segment @param safe: label an html segment as safe for safemode @param inline: label a segmant as inline html @returns : a placeholder string

class webhelpers.markdown.**BlockGuru**

detabbed_fn(*line*)
An auxiliary method to be passed to _findHead

webhelpers.markdown.**print_error**(*string*)
Print an error string to stderr

webhelpers.markdown.**dequote**(*string*)
Removes quotes from around a string

class webhelpers.markdown.**CorePatterns**
This class is scheduled for removal as part of a refactoring effort.

class webhelpers.markdown.**Extension**(*configs={}*)

webhelpers.markdown.**parse_options**()

mimehelper – MIMEtypes helper

webhelpers.mimehelper

class webhelpers.mimehelper.**MIMETypes**(*environ*)
MIMETypes registration mapping

The MIMETypes object class provides a single point to hold onto all the registered mimetypes, and their association extensions. It's used by the mimetypes method to determine the appropriate content type to return to a client.

classmethod add_alias(*alias*, *mimetype*)
Create a MIMEType alias to a full mimetype.

Examples:

- `add_alias('html', 'text/html')`
- `add_alias('xml', 'application/xml')`

An `alias` may not contain the `/` character.

classmethod init()
Loads a default mapping of extensions and mimetypes

These are suitable for most web applications by default. Additional types can be added by using the mimetypes module.

mimetype(*content_type*)
Check the PATH_INFO of the current request and client's HTTP Accept to attempt to use the appropriate mime-type.

If a content-type is matched, return the appropriate response content type, and if running under Pylons, set the response content type directly. If a content-type is not matched, return `False`.

This works best with URLs that end in extensions that differentiate content-type. Examples: `http://example.com/example`, `http://example.com/example.xml`, `http://example.com/example.csv`

Since browsers generally allow for any content-type, but should be sent HTML when possible, the html mimetype check should always come first, as shown in the example below.

Example:

```
# some code likely in environment.py
MIMETypes.init()
MIMETypes.add_alias('html', 'text/html')
MIMETypes.add_alias('xml', 'application/xml')
MIMETypes.add_alias('csv', 'text/csv')

# code in a Pylons controller
def someaction(self):
    # prepare a bunch of data
    # ......

    # prepare MIMETypes object
    m = MIMETypes(request.environ)

    if m.mimetype('html'):
        return render('/some/template.html')
    elif m.mimetype('atom'):
        return render('/some/xml_template.xml')
    elif m.mimetype('csv'):
        # write the data to a csv file
        return csvfile
    else:
        abort(404)

# Code in a non-Pylons controller.
m = MIMETypes(environ)
response_type = m.mimetype('html')
# ``response_type`` is a MIME type or ``False``.
```

misc – Miscellaneous helpers

webhelpers.misc

webhelpers.misc.**all**(*seq*, *pred=None*)
Is `pred(elm)` true for all elements?

With the default predicate, this is the same as Python 2.5's `all()` function; i.e., it returns true if all elements are true.

```
>>> all(["A", "B"])
True
>>> all(["A", ""])
False
>>> all(["", ""])
False
>>> all(["A", "B", "C"], lambda x: x <= "C")
True
>>> all(["A", "B", "C"], lambda x: x < "C")
False
```

From recipe in itertools docs.

webhelpers.misc.**any**(*seq*, *pred=None*)

Is `pred(elm)` is true for any element?

With the default predicate, this is the same as Python 2.5's `any()` function; i.e., it returns true if any element is true.

```
>>> any(["A", "B"])
True
>>> any(["A", ""])
True
>>> any(["", ""])
False
>>> any(["A", "B", "C"], lambda x: x <= "C")
True
>>> any(["A", "B", "C"], lambda x: x < "C")
True
```

From recipe in itertools docs.

webhelpers.misc.**no**(*seq*, *pred=None*)

Is `pred(elm)` false for all elements?

With the default predicate, this returns true if all elements are false.

```
>>> no(["A", "B"])
False
>>> no(["A", ""])
False
>>> no(["", ""])
True
>>> no(["A", "B", "C"], lambda x: x <= "C")
False
>>> no(["X", "Y", "Z"], lambda x: x <="C")
True
```

From recipe in itertools docs.

webhelpers.misc.**count_true**(*seq*, *pred=<function <lambda> at 0x52391b8>*)

How many elements is `pred(elm)` true for?

With the default predicate, this counts the number of true elements.

```
>>> count_true([1, 2, 0, "A", ""])
3
>>> count_true([1, "A", 2], lambda x: isinstance(x, int))
2
```

This is equivalent to the `itertools.quantify` recipe, which I couldn't get to work.

`webhelpers.misc.`**`convert_or_none`**(*value*, *type_*)
Return the value converted to the type, or None if error.

`type_` may be a Python type or any function taking one argument.

```
>>> print convert_or_none("5", int)
5
>>> print convert_or_none("A", int)
None
```

class `webhelpers.misc.`**`DeclarativeException`**(*message=None*)
A simpler way to define an exception with a fixed message.

Subclasses have a class attribute `.message`, which is used if no message is passed to the constructor. The default message is the empty string.

Example:

```
>>> class MyException(DeclarativeException):
...     message="can't frob the bar when foo is enabled"
...
>>> try:
...     raise MyException()
... except Exception, e:
...      print e
...
can't frob the bar when foo is enabled
```

`number` – Numbers and statistics helpers

`webhelpers.number`

`webhelpers.number.`**`percent_of`**(*part*, *whole*)
What percent of `whole` is `part`?

```
>>> percent_of(5, 100)
5.0
>>> percent_of(13, 26)
50.0
```

`webhelpers.number.`**`mean`**(*r*)
Return the mean (i.e., average) of a sequence of numbers.

```
>>> mean([5, 10])
7.5
```

`webhelpers.number.`**`median`**(*r*)
Return the median of an iterable of numbers.

The median is the point at which half the numbers are lower than it and half the numbers are higher. This gives a better sense of the majority level than the mean (average) does, because the mean can be skewed by a few extreme numbers at either end. For instance, say you want to calculate the typical household income in a community and you've sampled four households:

```
>>> incomes = [18000]        # Fast food crew
>>> incomes.append(24000)    # Janitor
>>> incomes.append(32000)    # Journeyman
```

```
>>> incomes.append(44000)   # Experienced journeyman
>>> incomes.append(67000)   # Manager
>>> incomes.append(9999999) # Bill Gates
>>> median(incomes)
49500.0
>>> mean(incomes)
1697499.8333333333
```

The median here is somewhat close to the majority of incomes, while the mean is far from anybody's income.

This implementation makes a temporary list of all numbers in memory.

webhelpers.number.**standard_deviation**(*r*, *sample=True*)

Standard deviation.

From the Python Cookbook. Population mode contributed by Lorenzo Catucci.

Standard deviation shows the variability within a sequence of numbers. A small standard deviation means the numbers are close to each other. A large standard deviation shows they are widely different. In fact it shows how far the numbers tend to deviate from the average. This can be used to detect whether the average has been skewed by a few extremely high or extremely low values.

Most natural and random phenomena follow the normal distribution (aka the bell curve), which says that most values are close to average but a few are extreme. E.g., most people are close to 5'9" tall but a few are very tall or very short. If the data does follow the bell curve, 68% of the values will be within 1 standard deviation (stdev) of the average, and 95% will be within 2 standard deviations. So a university professor grading exams on a curve might give a "C" (mediocre) grade to students within 1 stdev of the average score, "B" (better than average) to those within 2 stdevs above, and "A" (perfect) to the 0.25% higher than 2 stdevs. Those between 1 and 2 stdevs below get a "D" (poor), and those below 2 stdevs... we won't talk about them.

By default the helper computes the unbiased estimate for the population standard deviation, by applying an unbiasing factor of sqrt(N/(N-1)).

If you'd rather have the function compute the population standard deviation, pass `sample=False`.

The following examples are taken from Wikipedia. http://en.wikipedia.org/wiki/Standard_deviation

```
>>> standard_deviation([0, 0, 14, 14])
8.082903768654761...
>>> standard_deviation([0, 6, 8, 14])
5.773502691896258...
>>> standard_deviation([6, 6, 8, 8])
1.1547005383792515
>>> standard_deviation([0, 0, 14, 14], sample=False)
7.0
>>> standard_deviation([0, 6, 8, 14], sample=False)
5.0
>>> standard_deviation([6, 6, 8, 8], sample=False)
1.0
```

(The results reported in Wikipedia are those expected for whole population statistics and therefore are equal to the ones we get by setting `sample=False` in the later tests.)

```
# Fictitious average monthly temperatures in Southern California.
#                       Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
>>> standard_deviation([70, 70, 70, 75, 80, 85, 90, 95, 90, 80, 75, 70])
9.003366373785...
>>> standard_deviation([70, 70, 70, 75, 80, 85, 90, 95, 90, 80, 75, 70], sample=False)
8.620067027323...
```

```
    # Fictitious average monthly temperatures in Montana.
    #                       Jan  Feb  Mar Apr May Jun Jul  Aug Sep Oct Nov Dec
    >>> standard_deviation([-32, -10, 20, 30, 60, 90, 100, 80, 60, 30, 10, -32])
    45.1378360405574...
    >>> standard_deviation([-32, -10, 20, 30, 60, 90, 100, 80, 60, 30, 10, -32], sample=False)
    43.2161878106906...
```

class webhelpers.number.**SimpleStats**(*numeric=False*)
Calculate a few simple statistics on data.

This class calculates the minimum, maximum, and count of all the values given to it. The values are not saved in the object. Usage:

```
    >>> stats = SimpleStats()
    >>> stats(2)               # Add one data value.
    >>> stats.extend([6, 4])   # Add several data values at once.
```

The statistics are available as instance attributes:

```
    >>> stats.count
    3
    >>> stats.min
    2
    >>> stats.max
    6
```

Non-numeric data is also allowed:

```
    >>> stats2 = SimpleStats()
    >>> stats2("foo")
    >>> stats2("bar")
    >>> stats2.count
    2
    >>> stats2.min
    'bar'
    >>> stats2.max
    'foo'
```

`.min` and `.max` are `None` until the first data value is registered.

Subclasses can override `._init_stats` and `._update_stats` to add additional statistics.

The constructor accepts one optional argument, `numeric`. If true, the instance accepts only values that are `int`, `long`, or `float`. The default is false, which accepts any value. This is meant for instances or subclasses that don't want non-numeric values.

extend(*values*)
Add several data values at once, akin to `list.extend`.

class webhelpers.number.**Stats**
A container for data and statistics.

This class extends `SimpleStats` by calculating additional statistics, and by storing all data seen. All values must be numeric (`int`, `long`, and/or `float`), and you must call `.finish()` to generate the additional statistics. That's because the statistics here cannot be calculated incrementally, but only after all data is known.

```
    >>> stats = Stats()
    >>> stats.extend([5, 10, 10])
    >>> stats.count
    3
```

```
>>> stats.finish()
>>> stats.mean
8.33333333333333...
>>> stats.median
10
>>> stats.standard_deviation
2.8867513459481287
```

All data is stored in a list and a set for later use:

```
>>> stats.list
[5, 10, 10]

>>  stats.set
set([5, 10])
```

(The double prompt ">>" is used to hide the example from doctest.)

The stat attributes are `None` until you call `.finish()`. It's permissible – though not recommended – to add data after calling `.finish()` and then call `.finish()` again. This recalculates the stats over the entire data set.

In addition to the hook methods provided by `SimpleStats`, subclasses can override `._finish-stats` to provide additional statistics.

finish()
Finish calculations. (Call after adding all data values.)

Call this after adding all data values, or the results will be incomplete.

webhelpers.number.**format_number**(*n*, *thousands=', '*, *decimal='.'*)
Format a number with a thousands separator and decimal delimiter.

`n` may be an int, long, float, or numeric string. `thousands` is a separator to put after each thousand. `decimal` is the delimiter to put before the fractional portion if any.

The default style has a thousands comma and decimal point per American usage:

```
>>> format_number(1234567.89)
'1,234,567.89'
>>> format_number(123456)
'123,456'
>>> format_number(-123)
'-123'
```

Various European and international styles are also possible:

```
>>> format_number(1234567.89, " ")
'1 234 567.89'
>>> format_number(1234567.89, " ", ",")
'1 234 567,89'
>>> format_number(1234567.89, ".", ",")
'1.234.567,89'
```

paginate – Paging and pagination

webhelpers.paginate

webhelpers.paginate.**get_wrapper**(*obj*, *sqlalchemy_session=None*)
Auto-detect the kind of object and return a list/tuple to access items from the collection.

class webhelpers.paginate.**Page**(*collection*, *page=1*, *items_per_page=20*, *item_count=None*, *sqlalchemy_session=None*, *presliced_list=False*, *url=None*, ***kwargs*)

A list/iterator of items representing one page in a larger collection.

An instance of the "Page" class is created from a collection of things. The instance works as an iterator running from the first item to the last item on the given page. The collection can be:

- a sequence
- an SQLAlchemy query - e.g.: Session.query(MyModel)
- an SQLAlchemy select - e.g.: sqlalchemy.select([my_table])

A "Page" instance maintains pagination logic associated with each page, where it begins, what the first/last item on the page is, etc. The pager() method creates a link list allowing the user to go to other pages.

WARNING: Unless you pass in an item_count, a count will be performed on the collection every time a Page instance is created. If using an ORM, it's advised to pass in the number of items in the collection if that number is known.

Instance attributes:

original_collection Points to the collection object being paged through

item_count Number of items in the collection

page Number of the current page

items_per_page Maximal number of items displayed on a page

first_page Number of the first page - starts with 1

last_page Number of the last page

page_count Number of pages

items Sequence/iterator of items on the current page

first_item Index of first item on the current page - starts with 1

last_item Index of last item on the current page

pager(*format='~2~'*, *page_param='page'*, *partial_param='partial'*, *show_if_single_page=False*, *separator=' '*, *onclick=None*, *symbol_first='<<'*, *symbol_last='>>'*, *symbol_previous='<'*, *symbol_next='>'*, *link_attr={'class': 'pager_link'}*, *curpage_attr={'class': 'pager_curpage'}*, *dotdot_attr={'class': 'pager_dotdot'}*, ***kwargs*)

Return string with links to other pages (e.g. "1 2 [3] 4 5 6 7").

format: Format string that defines how the pager is rendered. The string can contain the following $-tokens that are substituted by the string.Template module:

- $first_page: number of first reachable page
- $last_page: number of last reachable page
- $page: number of currently selected page
- $page_count: number of reachable pages
- $items_per_page: maximal number of items per page
- $first_item: index of first item on the current page
- $last_item: index of last item on the current page
- $item_count: total number of items

- $link_first: link to first page (unless this is first page)
- $link_last: link to last page (unless this is last page)
- $link_previous: link to previous page (unless this is first page)
- $link_next: link to next page (unless this is last page)

To render a range of pages the token '~3~' can be used. The number sets the radius of pages around the current page. Example for a range with radius 3:

'1 .. 5 6 7 [8] 9 10 11 .. 500'

Default: '~2~'

symbol_first String to be displayed as the text for the %(link_first)s link above.

Default: '<<'

symbol_last String to be displayed as the text for the %(link_last)s link above.

Default: '>>'

symbol_previous String to be displayed as the text for the %(link_previous)s link above.

Default: '<'

symbol_next String to be displayed as the text for the %(link_next)s link above.

Default: '>'

separator: String that is used to separate page links/numbers in the above range of pages.

Default: ' '

page_param: The name of the parameter that will carry the number of the page the user just clicked on. The parameter will be passed to a url_for() call so if you stay with the default ':controller/:action/:id' routing and set page_param='id' then the :id part of the URL will be changed. If you set page_param='page' then url_for() will make it an extra parameters like ':controller/:action/:id?page=1'. You need the page_param in your action to determine the page number the user wants to see. If you do not specify anything else the default will be a parameter called 'page'.

Note: If you set this argument and are using a URL generator callback, the callback must accept this name as an argument instead of 'page'. callback, becaust the callback requires its argument to be 'page'. Instead the callback itself can return any URL necessary.

partial_param: When using AJAX/AJAH to do partial updates of the page area the application has to know whether a partial update (only the area to be replaced) or a full update (reloading the whole page) is required. So this parameter is the name of the URL parameter that gets set to 1 if the 'onclick' parameter is used. So if the user requests a new page through a Javascript action (onclick) then this parameter gets set and the application is supposed to return a partial content. And without Javascript this parameter is not set. The application thus has to check for the existence of this parameter to determine whether only a partial or a full page needs to be returned. See also the examples in this modules docstring.

Default: 'partial'

Note: If you set this argument and are using a URL generator callback, the callback must accept this name as an argument instead of 'partial'.

show_if_single_page: if True the navigator will be shown even if there is only one page

Default: False

link_attr (optional) A dictionary of attributes that get added to A-HREF links pointing to other pages. Can be used to define a CSS style or class to customize the look of links.

Example: { 'style':'border: 1px solid green' }

Default: { 'class':'pager_link' }

curpage_attr (optional) A dictionary of attributes that get added to the current page number in the pager (which is obviously not a link). If this dictionary is not empty then the elements will be wrapped in a SPAN tag with the given attributes.

Example: { 'style':'border: 3px solid blue' }

Default: { 'class':'pager_curpage' }

dotdot_attr (optional) A dictionary of attributes that get added to the '..' string in the pager (which is obviously not a link). If this dictionary is not empty then the elements will be wrapped in a SPAN tag with the given attributes.

Example: { 'style':'color: #808080' }

Default: { 'class':'pager_dotdot' }

onclick (optional) This paramter is a string containing optional Javascript code that will be used as the 'onclick' action of each pager link. It can be used to enhance your pager with AJAX actions loading another page into a DOM object.

In this string the variable '$partial_url' will be replaced by the URL linking to the desired page with an added 'partial=1' parameter (or whatever you set 'partial_param' to). In addition the '$page' variable gets replaced by the respective page number.

Note that the URL to the destination page contains a 'partial_param' parameter so that you can distinguish between AJAX requests (just refreshing the paginated area of your page) and full requests (loading the whole new page).

[Backward compatibility: you can use '%s' instead of '$partial_url']

jQuery example: "$('#my-page-area').load('$partial_url'); return false;"

Yahoo UI example:

"YAHOO.util.Connect.asyncRequest('GET','$partial_url',{
success:function(o){YAHOO.util.Dom.get('#my-page-area').innerHTML=o.responseText;}
},null); return false;"

scriptaculous example:

"new Ajax.Updater('#my-page-area', '$partial_url', {asynchronous:true,
evalScripts:true}); return false;"

ExtJS example: "Ext.get('#my-page-area').load({url:'$partial_url'}); return false;"

Custom example: "my_load_page($page)"

Additional keyword arguments are used as arguments in the links. Otherwise the link will be created with url_for() which points to the page you are currently displaying.

Temporary understudy for examples:

The unit tests are often educational ...

```
"""Test webhelpers.paginate package."""

from routes import Mapper

from webhelpers.paginate import Page


def test_empty_list():
    """Test whether an empty list is handled correctly."""
    items = []
    page = Page(items, page=0)
    assert page.page == 0
    assert page.first_item is None
    assert page.last_item is None
    assert page.first_page is None
    assert page.last_page is None
    assert page.previous_page is None
    assert page.next_page is None
    assert page.items_per_page == 20
    assert page.item_count == 0
    assert page.page_count == 0
    assert page.pager() == ''
    assert page.pager(show_if_single_page=True) == ''

def test_one_page():
    """Test that we fit 10 items on a single 10-item page."""
    items = range(10)
    page = Page(items, page=0, items_per_page=10)
    assert page.page == 1
    assert page.first_item == 1
    assert page.last_item == 10
    assert page.first_page == 1
    assert page.last_page == 1
    assert page.previous_page is None
    assert page.next_page is None
    assert page.items_per_page == 10
    assert page.item_count == 10
    assert page.page_count == 1
    assert page.pager() == ''
    assert page.pager(show_if_single_page=True) == \
        '<span class="pager_curpage">1</span>'

def test_many_pages():
    """Test that 100 items fit on seven 15-item pages."""
    # Create routes mapper so that webhelper can create URLs
    # using webhelpers.url_for()
    mapper = Mapper()
    mapper.connect(':controller')

    items = range(100)
    page = Page(items, page=0, items_per_page=15)
    assert page.page == 1
    assert page.first_item == 1
    assert page.last_item == 15
    assert page.first_page == 1
    assert page.last_page == 7
    assert page.previous_page is None
    assert page.next_page == 2
```

```
    assert page.items_per_page == 15
    assert page.item_count == 100
    assert page.page_count == 7
    print page.pager()
    assert page.pager() == \
        '<span class="pager_curpage">1</span> ' + \
        '<a class="pager_link" href="/content?page=2">2</a> ' + \
        '<a class="pager_link" href="/content?page=3">3</a> ' + \
        '<span class="pager_dotdot">..</span> ' + \
        '<a class="pager_link" href="/content?page=7">7</a>'
    assert page.pager(separator='_') == \
        '<span class="pager_curpage">1</span>_' + \
        '<a class="pager_link" href="/content?page=2">2</a>_' + \
        '<a class="pager_link" href="/content?page=3">3</a>_' + \
        '<span class="pager_dotdot">..</span>_' + \
        '<a class="pager_link" href="/content?page=7">7</a>'
    assert page.pager(page_param='xy') == \
        '<span class="pager_curpage">1</span> ' + \
        '<a class="pager_link" href="/content?xy=2">2</a> ' + \
        '<a class="pager_link" href="/content?xy=3">3</a> ' + \
        '<span class="pager_dotdot">..</span> ' + \
        '<a class="pager_link" href="/content?xy=7">7</a>'
    assert page.pager(
        link_attr={'style':'s1'},
        curpage_attr={'style':'s2'},
        dotdot_attr={'style':'s3'}) == \
            '<span style="s2">1</span> ' + \
            '<a href="/content?page=2" style="s1">2</a> ' + \
            '<a href="/content?page=3" style="s1">3</a> ' + \
            '<span style="s3">..</span> ' + \
            '<a href="/content?page=7" style="s1">7</a>'
```

webhelpers.pylonslib – flash alert div helpers

webhelpers.pylonslib

class webhelpers.pylonslib.**Flash**(*session_key='flash'*, *categories=None*, *default_category=None*)

Accumulate a list of messages to show at the next page request.

pop_messages()

Return all accumulated messages and delete them from the session.

The return value is a list of `Message` objects.

text – Text helpers

webhelpers.text

webhelpers.text.**truncate**(*text*, *length=30*, *indicator='...'*, *whole_word=False*)

Truncate `text` with replacement characters.

length The maximum length of `text` before replacement

indicator If `text` exceeds the `length`, this string will replace the end of the string

whole_word If true, shorten the string further to avoid breaking a word in the middle. A word is defined as any string not containing whitespace. If the entire text before the break is a single word, it will have to be broken.

Example:

```
>>> truncate('Once upon a time in a world far far away', 14)
'Once upon a...'
```

webhelpers.text.**excerpt**(*text*, *phrase*, *radius=100*, *excerpt_string='...'*)

Extract an excerpt from the `text`, or '' if the phrase isn't found.

phrase Phrase to excerpt from `text`

radius How many surrounding characters to include

excerpt_string Characters surrounding entire excerpt

Example:

```
>>> excerpt("hello my world", "my", 3)
'...lo my wo...'
```

webhelpers.text.**plural**(*n*, *singular*, *plural*, *with_number=True*)

Return the singular or plural form of a word, according to the number.

If `with_number` is true (default), the return value will be the number followed by the word. Otherwise the word alone will be returned.

Usage:

```
>>> plural(2, "ox", "oxen")
'2 oxen'
>>> plural(2, "ox", "oxen", False)
'oxen'
```

webhelpers.text.**chop_at**(*s*, *sub*, *inclusive=False*)

Truncate string `s` at the first occurrence of `sub`.

If `inclusive` is true, truncate just after `sub` rather than at it.

```
>>> chop_at("plutocratic brats", "rat")
'plutoc'
>>> chop_at("plutocratic brats", "rat", True)
'plutocrat'
```

webhelpers.text.**lchop**(*s*, *sub*)

Chop `sub` off the front of `s` if present.

```
>>> lchop("##This is a comment.##", "##")
'This is a comment.##'
```

The difference between `lchop` and `s.lstrip` is that `lchop` strips only the exact prefix, while `s.lstrip` treats the argument as a set of leading characters to delete regardless of order.

webhelpers.text.**rchop**(*s*, *sub*)

Chop `sub` off the end of `s` if present.

```
>>> rchop("##This is a comment.##", "##")
'##This is a comment.'
```

The difference between `rchop` and `s.rstrip` is that `rchop` strips only the exact suffix, while `s.rstrip` treats the argument as a set of trailing characters to delete regardless of order.

webhelpers.text.**strip_leading_whitespace**(*s*)
Strip the leading whitespace in all lines in `s`.

This deletes *all* leading whitespace. `textwrap.dedent` deletes only the whitespace common to all lines.

webhelpers.text.**wrap_paragraphs**(*text*, *width=72*)
Wrap all paragraphs in a text string to the specified width.

`width` may be an int or a `textwrap.TextWrapper` instance. The latter allows you to set other options besides the width, and is more efficient when wrapping many texts.

`textile` – Textile

`webhelpers.textile`

webhelpers.textile.**textile**(*text*, ***args*)
This is Textile.

Generates XHTML from a simple markup developed by Dean Allen.

This function should be called like this:

textile(text, head_offset=0, validate=0, sanitize=0, encoding='latin-1', output='ASCII')

`util` – Utilities

`webhelpers.util`

webhelpers.util.**html_escape**(*s*)
HTML-escape a string or object.

This converts any non-string objects passed into it to strings (actually, using `unicode()`). All values returned are non-unicode strings (using `&#num;` entities for all non-ASCII characters).

None is treated specially, and returns the empty string.

This function returns a plain string. Programs using the HTML builder should wrap the result in `literal()` to prevent double-escaping.

webhelpers.util.**iri_to_uri**(*iri*)
Convert an IRI portion to a URI portion suitable for inclusion in a URL.

(An IRI is an Internationalized Resource Identifier.)

This is the algorithm from section 3.1 of RFC 3987. However, since we are assuming input is either UTF-8 or unicode already, we can simplify things a little from the full method.

Returns an ASCII string containing the encoded result.

class webhelpers.util.**Partial**(**args*, ***kw*)
A partial function object.

Equivalent to functools.partial, which was introduced in Python 2.5.

class webhelpers.util.**SimplerXMLGenerator**(*out=None*, *encoding='iso-8859-1'*)
A subclass of Python's SAX XMLGenerator.

addQuickElement(*name*, *contents=None*, *attrs=None*)
Add an element with no children.

class webhelpers.util.**UnicodeMultiDict**(*multi=None*, *encoding=None*, *errors='strict'*, *decode_keys=False*)

A MultiDict wrapper that decodes returned values to unicode on the fly.

Decoding is not applied to assigned values.

The key/value contents are assumed to be `str/strs` or `str/FieldStorages` (as is returned by the `paste.request.parse()` functions).

Can optionally also decode keys when the `decode_keys` argument is True.

`FieldStorage` instances are cloned, and the clone's `filename` variable is decoded. Its `name` variable is decoded when `decode_keys` is enabled.

add(*key*, *value*)
Add the key and value, not overwriting any previous value.

dict_of_lists()
Return dict where each key is associated with a list of values.

getall(*key*)
Return list of all values matching the key (may be an empty list).

getone(*key*)
Return one value matching key. Raise KeyError if multiple matches.

mixed()
Return dict where values are single values or a list of values.

The value is a single value if key appears just once. It is 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.

27.5.2 Deprecated

webhelpers.commands.compress_resources – *(deprecated)*

Warning: DEPRECATED!! BUGGY!! Do not use in new projects.

hinclude *(deprecated)*

webhelpers.hinclude

Warning: `webhelpers/hinclude.py` is deprecated and is too trivial to port, DIY.

htmlgen *(deprecated)*

webhelpers.htmlgen

Warning: `webhelpers/htmlgen.py` is deprecated, use `webhelpers.html` instead.

pagination – WebHelpers Pagination *(part deprecated)*

links

orm

> **Warning:** Deprecated: Use `webhelpers.paginate`

rails WebHelpers Rails *(deprecated)*

> **Warning:** Deprecated in 0.6

asset_tag

date

form_options

form_tag

javascript

number

prototype

> **Warning:** Deprecated, will be removed. No replacement is planned.

scriptaculous

> **Warning:** Deprecated, will be removed. No replacement is planned.

secure_form_tag

tags

text

urls

wrapped

javascripts

Warning: Deprecated, will be removed. No replacement is planned.

27.6 webtest – WebTest

Routines for testing WSGI applications.

Most interesting is app

class webtest.**TestApp**(*app*, *extra_environ=None*, *relative_to=None*, *use_unicode=True*)
Wraps a WSGI application in a more convenient interface for testing.

app may be an application, or a Paste Deploy app URI, like 'config:filename.ini#test'.

extra_environ is a dictionary of values that should go into the environment for each request. These can provide a communication channel with the application.

relative_to is a directory, and filenames used for file uploads are calculated relative to this. Also config: URIs that aren't absolute.

delete(*url*, *params=''*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*, *content_type=None*)
Do a DELETE request. Very like the .get() method.

Returns a webob.Response object.

delete_json(*url*, *params=<class 'webtest.app.NoDefault'>*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a DELETE request. Very like the .get() method. Content-Type is set to application/json.

Returns a webob.Response object.

do_request(*req*, *status*, *expect_errors*)
Executes the given request (req), with the expected status. Generally .get() and .post() are used instead.

To use this:

```
resp = app.do_request(webtest.TestRequest.blank(
    'url', ...args...))
```

Note you can pass any keyword arguments to TestRequest.blank(), which will be set on the request. These can be arguments like content_type, accept, etc.

encode_multipart(*params*, *files*)
Encodes a set of parameters (typically a name/value list) and a set of files (a list of (name, filename, file_body)) into a typical POST body, returning the (content_type, body).

get(*url*, *params=None*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Get the given url (well, actually a path like `'/page.html'`).

params: A query string, or a dictionary that will be encoded into a query string. You may also include a query string on the `url`.

headers: A dictionary of extra headers to send.

extra_environ: A dictionary of environmental variables that should be added to the request.

status: The integer status code you expect (if not 200 or 3xx). If you expect a 404 response, for instance, you must give `status=404` or it will be an error. You can also give a wildcard, like `'3*'` or `'*'`.

expect_errors: If this is not true, then if anything is written to `wsgi.errors` it will be an error. If it is true, then non-200/3xx responses are also okay.

Returns a `webtest.TestResponse` object.

head(*url*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a HEAD request. Very like the `.get()` method.

Returns a `webob.Response` object.

options(*url*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a OPTIONS request. Very like the `.get()` method.

Returns a `webob.Response` object.

post(*url*, *params=''*, *headers=None*, *extra_environ=None*, *status=None*, *upload_files=None*, *expect_errors=False*, *content_type=None*)
Do a POST request. Very like the `.get()` method. `params` are put in the body of the request.

`upload_files` is for file uploads. It should be a list of `[(fieldname, filename, file_content)]`. You can also use just `[(fieldname, filename)]` and the file content will be read from disk.

For post requests params could be a collections.OrderedDict with Upload fields included in order:

app.post('/myurl', collections.OrderedDict([('textfield1', 'value1'), ('uploadfield', webapp.Upload('filename.txt', 'contents'), ('textfield2', 'value2')])))

Returns a `webob.Response` object.

post_json(*url*, *params=<class 'webtest.app.NoDefault'>*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a POST request. Very like the `.get()` method. `params` are dumps to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webob.Response` object.

put(*url*, *params=''*, *headers=None*, *extra_environ=None*, *status=None*, *upload_files=None*, *expect_errors=False*, *content_type=None*)
Do a PUT request. Very like the `.post()` method. `params` are put in the body of the request, if params is a tuple, dictionary, list, or iterator it will be urlencoded and placed in the body as with a POST, if it is string it will not be encoded, but placed in the body directly.

Returns a `webob.Response` object.

put_json(*url*, *params=<class 'webtest.app.NoDefault'>*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a PUT request. Very like the `.post()` method. `params` are dumps to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webob.Response` object.

request(*url_or_req*, *status=None*, *expect_errors=False*, ***req_params*)
Creates and executes a request. You may either pass in an instantiated `TestRequest` object, or you may pass in a URL and keyword arguments to be passed to `TestRequest.blank()`.

You can use this to run a request without the intermediary functioning of `TestApp.get()` etc. For instance, to test a WebDAV method:

```
resp = app.request('/new-col', method='MKCOL')
```

Note that the request won't have a body unless you specify it, like:

```
resp = app.request('/test.txt', method='PUT', body='test')
```

You can use `POST={args}` to set the request body to the serialized arguments, and simultaneously set the request method to `POST`

reset()
Resets the state of the application; currently just clears saved cookies.

class webtest.**TestResponse**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)
Instances of this class are return by `TestApp`

click(*description=None*, *linkid=None*, *href=None*, *anchor=None*, *index=None*, *verbose=False*, *extra_environ=None*)
Click the link as described. Each of `description`, `linkid`, and `url` are *patterns*, meaning that they are either strings (regular expressions), compiled regular expressions (objects with a `search` method), or callables returning true or false.

All the given patterns are ANDed together:

- `description` is a pattern that matches the contents of the anchor (HTML and all – everything between `<a...>` and `</a>`)
- `linkid` is a pattern that matches the `id` attribute of the anchor. It will receive the empty string if no id is given.
- `href` is a pattern that matches the `href` of the anchor; the literal content of that attribute, not the fully qualified attribute.
- `anchor` is a pattern that matches the entire anchor, with its contents.

If more than one link matches, then the `index` link is followed. If `index` is not given and more than one link matches, or if no link matches, then `IndexError` will be raised.

If you give `verbose` then messages will be printed about each link, and why it does or doesn't match. If you use `app.click(verbose=True)` you'll see a list of all the links.

You can use multiple criteria to essentially assert multiple aspects about the link, e.g., where the link's destination is.

clickbutton(*description=None*, *buttonid=None*, *href=None*, *button=None*, *index=None*, *verbose=False*)
Like `.click()`, except looks for link-like buttons. This kind of button should look like `<button onclick="...location.href='url'...">`.

follow(***kw*)
If this request is a redirect, follow that redirect. It is an error if this is not a redirect response. Returns another response object.

form
Returns a single `Form` instance; it is an error if there are multiple forms on the page.

forms
A list of :class:'~webtest.Form's found on the page

forms__get()
Returns a dictionary of `Form` objects. Indexes are both in order (from zero) and by form id (if the form is given an id).

goto(*href*, *method='get'*, ***args*)
Go to the (potentially relative) link `href`, using the given method (`'get'` or `'post'`) and any extra arguments you want to pass to the `app.get()` or `app.post()` methods.

All hostnames and schemes will be ignored.

html
Returns the response as a BeautifulSoup object.

Only works with HTML responses; other content-types raise AttributeError.

json
Return the response as a JSON response. You must have simplejson installed to use this, or be using a Python version with the json module.

The content type must be application/json to use this.

lxml
Returns the response as an lxml object. You must have lxml installed to use this.

If this is an HTML response and you have lxml 2.x installed, then an `lxml.html.HTML` object will be returned; if you have an earlier version of lxml then a `lxml.HTML` object will be returned.

mustcontain(**strings*, ***kw*)
Assert that the response contains all of the strings passed in as arguments.

Equivalent to:

```
assert string in res
```

normal_body
Return the whitespace-normalized body

pyquery
Returns the response as a PyQuery object.

Only works with HTML and XML responses; other content-types raise AttributeError.

showbrowser()
Show this response in a browser window (for debugging purposes, when it's hard to read the HTML).

unicode_normal_body
Return the whitespace-normalized body, as unicode

xml
Returns the response as an ElementTree object.

Only works with XML responses; other content-types raise AttributeError

class webtest.**Form**(*response*, *text*)
This object represents a form that has been found in a page. This has a couple useful attributes:

text: the full HTML of the form.

action: the relative URI of the action.

method: the method (e.g., 'GET').

id: the id, or None if not given.

fields: a dictionary of fields, each value is a list of fields by that name. <input type="radio"> and <select> are both represented as single fields with multiple options.

FieldClass
alias of Field

get(*name*, *index=None*, *default=<class 'webtest.app.NoDefault'>*)
Get the named/indexed field object, or default if no field is found.

lint()
Check that the html is valid:

- each field must have an id
- each field must have a label

select(*name*, *value*, *index=None*)
Like .set(), except also confirms the target is a <select>.

set(*name*, *value*, *index=None*)
Set the given name, using index to disambiguate.

submit(*name=None*, *index=None*, ***args*)
Submits the form. If name is given, then also select that button (using index to disambiguate)''.

Any extra keyword arguments are passed to the .get() or .post() method.

Returns a webtest.TestResponse object.

submit_fields(*name=None*, *index=None*)
Return a list of [(name, value), ...] for the current state of the form.

upload_fields()

Return a list of file field tuples of the form: (field name, file name)

or (field name, file name, file contents).

27.7 webob – WebOb

class webob.**Request**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
The default request implementation

class webob.**Response**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)
Represents a WSGI response

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 `str`. 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 (in the Content-Type)

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 (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.

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)

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(*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.

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.

classmethod 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`

headerlist
The list of response headers

headers
The headers in a dictionary-like object

json
Access the body of the response as JSON

json_body
Access the body of the response as JSON

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(*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.

Arguments are:

`key`

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 or `None`. If this value is an integer, it is used as the `Max-Age` of the generated cookie. If `expires` is not passed and this value is an integer, 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 also sent).

`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 or the value `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 (`Max-Age` will be 'expires_value - datetime.utcnow()'). If this value is `None`, the `Expires` cookie value will be unset (unless `max_age` is also passed).

`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)

ubody
Deprecated alias for .text

unicode_body
Deprecated alias for .text

unset_cookie(*key*, *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`.

`webob.`**html_escape**(*s*)
HTML-escape a string or object

This converts any non-string objects passed into it to strings (actually, using `unicode()`). All values returned are non-unicode strings (using `&#num;` entities for all non-ASCII characters).

None is treated specially, and returns the empty string.

`webob.`**timedelta_to_seconds**(*td*)
Converts a timedelta instance to seconds.

27.7.1 webob.acceptparse

class `webob.acceptparse.`**Accept**(*header_value*)
Represents a generic `Accept-*` style header.

This object should not be modified. To add items you can use `accept_obj + 'accept_thing'` to get a new object

best_match(*offers*, *default_match=None*)
Returns the best match in the sequence of offered types.

The sequence can be a simple sequence, or you can have `(match, server_quality)` items in the sequence. If you have these tuples then the client quality is multiplied by the server_quality to get a total. If two matches have equal weight, then the one that shows up first in the *offers* list will be returned.

But among matches with the same quality the match to a more specific requested type will be chosen. For example a match to text/* trumps /.

default_match (default None) is returned if there is no intersection.

first_match(*offers*)
DEPRECATED Returns the first allowed offered type. Ignores quality. Returns the first offered type if nothing else matches; or if you include None at the end of the match list then that will be returned.

static parse(*value*)
Parse `Accept-*` style header.

Return iterator of `(value, quality)` pairs. `quality` defaults to 1.

quality(*offer*, *modifier=1*)
Return the quality of the given offer. Returns None if there is no match (not 0).

class webob.acceptparse.**NilAccept**

MasterClass
alias of Accept

class webob.acceptparse.**NoAccept**

class webob.acceptparse.**MIMEAccept**(*header_value*)
Represents the `Accept` header, which is a list of mimetypes.

This class knows about mime wildcards, like `image/*`

accept_html()
Returns true if any HTML-like type is accepted

accepts_html
Returns true if any HTML-like type is accepted

class webob.acceptparse.**MIMENilAccept**

MasterClass
alias of MIMEAccept

27.7.2 webob.byterange

class webob.byterange.**Range**(*start*, *end*)
Represents the Range header.

content_range(*length*)
Works like range_for_length; returns None or a ContentRange object

You can use it like:

```
response.content_range = req.range.content_range(response.content_length)
```

Though it's still up to you to actually serve that content range!

classmethod parse(*header*)
Parse the header; may return None if header is invalid

range_for_length(*length*)
If there is only one range, and *if* it is satisfiable by the given length, then return a (start, end) non-inclusive range of bytes to serve. Otherwise return None

class webob.byterange.**ContentRange**(*start*, *stop*, *length*)
Represents the Content-Range header

This header is `start-stop/length`, where start-stop and length can be `*` (represented as None in the attributes).

classmethod parse(*value*)
Parse the header. May return None if it cannot parse.

27.7.3 webob.cachecontrol

Represents the Cache-Control header

class webob.cachecontrol.**exists_property**(*prop*, *type=None*)
Represents a property that either is listed in the Cache-Control header, or is not listed (has no value)

class webob.cachecontrol.**value_property**(*prop*, *default=None*, *none=None*, *type=None*)
Represents a property that has a value in the Cache-Control header.

When no value is actually given, the value of self.none is returned.

class webob.cachecontrol.**CacheControl**(*properties*, *type*)
Represents the Cache-Control header.

By giving a type of `'request'` or `'response'` you can control what attributes are allowed (some Cache-Control values only apply to requests or responses).

copy()
Returns a copy of this object.

classmethod parse(*header*, *updates_to=None*, *type=None*)
Parse the header, returning a CacheControl object.

The object is bound to the request or response object `updates_to`, if that is given.

update_dict
alias of `UpdateDict`

webob.cachecontrol.**serialize_cache_control**(*properties*)

27.7.4 webob.datastruct

27.7.5 webob.etag

Does parsing of ETag-related headers: If-None-Matches, If-Matches

Also If-Range parsing

webob.etag.**AnyETag**

webob.etag.**NoETag**

class webob.etag.**ETagMatcher**(*etags*)

classmethod parse(*value*, *strong=True*)
Parse this from a header value

class webob.etag.**IfRange**(*etag*)

classmethod parse(*value*)
Parse this from a header value.

27.7.6 mod:*webob.exc*

HTTP Exception

This module processes Python exceptions that relate to HTTP exceptions by defining a set of exceptions, all subclasses of HTTPException. Each exception, in addition to being a Python exception that can be raised and caught, is also a WSGI application and `webob.Response` object.

This module defines exceptions according to RFC 2068 [1] : codes with 100-300 are not really errors; 400's are client errors, and 500's are server errors. According to the WSGI specification [2] , the application can call `start_response` more then once only under two conditions: (a) the response has not yet been sent, or (b) if the second and subsequent invocations of `start_response` have a valid `exc_info` argument obtained from `sys.exc_info()`. The WSGI specification then requires the server or gateway to handle the case where content has been sent and then an exception was encountered.

Exception

HTTPException

HTTPOk

- 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
- 306 - Unused (not implemented, obviously)
- 307 - HTTPTemporaryRedirect

HTTPError

HTTPClientError

- 400 - HTTPBadRequest
- 401 - HTTPUnauthorized
- 402 - HTTPPaymentRequired
- 403 - HTTPForbidden

[1] http://www.python.org/peps/pep-0333.html#error-handling
[2] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5

- 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
- 428 - HTTPPreconditionRequired
- 429 - HTTPTooManyRequests
- 431 - HTTPRequestHeaderFieldsTooLarge

HTTPServerError

- 500 - HTTPInternalServerError
- 501 - HTTPNotImplemented
- 502 - HTTPBadGateway
- 503 - HTTPServiceUnavailable
- 504 - HTTPGatewayTimeout
- 505 - HTTPVersionNotSupported
- 511 - HTTPNetworkAuthenticationRequired

Subclass usage notes:

The HTTPException class is complicated by 4 factors:

1. The content given to the exception may either be plain-text or as html-text.
2. The template may want to have string-substitutions taken from the current `environ` or values from incoming headers. This is especially troublesome due to case sensitivity.
3. The final output may either be text/plain or text/html mime-type as requested by the client application.
4. Each exception has a default explanation, but those who raise exceptions may want to provide additional detail.

Subclass attributes and call parameters are designed to provide an easier path through the complications.

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)s

detail a plain-text message customization that is not subject to environment or header substitutions; accessible in the template via %(detail)s

body_template a content fragment (in HTML) used for environment and header substitution; the default template includes both the explanation and further detail provided in the message

Parameters:

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

To override the template (which is HTML content) or the plain-text explanation, one must subclass the given exception; or customize it after it has been created. This particular breakdown of a message into explanation, detail and template allows both the creation of plain-text and html messages for various clients as well as error-free substitution of environment variables and headers.

The subclasses of `_HTTPMove` (`HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPFound`, `HTTPSeeOther`, `HTTPUseProxy` and `HTTPTemporaryRedirect`) are redirections that require a `Location` field. Reflecting this, these subclasses have two additional keyword arguments: `location` and `add_slash`.

Parameters:

location to set the location immediately

add_slash set to True to redirect to the same URL as the request, except with a `/` appended

Relative URLs in the location will be resolved to absolute.

References:

`webob.exc.`**no_escape**(*value*)

`webob.exc.`**strip_tags**(*value*)

class `webob.exc.`**HTTPException**(*message*, *wsgi_response*)

class `webob.exc.`**WSGIHTTPException**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

class `webob.exc.`**HTTPError**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for status codes in the 400's and 500's

This is an exception which indicates that an error has occurred, and that any work in progress should not be committed. These are typically results in the 400's and 500's.

class `webob.exc.`**HTTPRedirection**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for 300's status code (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.

class webob.exc.**HTTPOk**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
Base class for the 200's status code (successful responses)

code: 200, title: OK

class webob.exc.**HTTPCreated**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
subclass of HTTPOk

This indicates that request has been fulfilled and resulted in a new resource being created.

code: 201, title: Created

class webob.exc.**HTTPAccepted**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
subclass of HTTPOk

This indicates that the request has been accepted for processing, but the processing has not been completed.

code: 202, title: Accepted

class webob.exc.**HTTPNonAuthoritativeInformation**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
subclass of HTTPOk

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

class webob.exc.**HTTPNoContent**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
subclass of HTTPOk

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

class webob.exc.**HTTPResetContent**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
subclass of HTTPOk

This indicates that the 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

class webob.exc.**HTTPPartialContent**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
subclass of HTTPOk

This indicates that the server has fulfilled the partial GET request for the resource.

code: 206, title: Partial Content

class webob.exc.**HTTPMultipleChoices**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)
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

class webob.exc.**HTTPMovedPermanently**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

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

class webob.exc.**HTTPFound**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of _HTTPMove

This indicates that the requested resource resides temporarily under a different URI.

code: 302, title: Found

class webob.exc.**HTTPSeeOther**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

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

class webob.exc.**HTTPNotModified**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPUseProxy**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

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

class webob.exc.**HTTPTemporaryRedirect**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of _HTTPMove

This indicates that the requested resource resides temporarily under a different URI.

code: 307, title: Temporary Redirect

class webob.exc.**HTTPClientError**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for the 400's, 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'

class webob.exc.**HTTPBadRequest**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

class webob.exc.**HTTPUnauthorized**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPClientError

This indicates that the request requires user authentication.

code: 401, title: Unauthorized

class webob.exc.**HTTPPaymentRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPClientError

code: 402, title: Payment Required

class webob.exc.**HTTPForbidden**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPClientError

This indicates that the server understood the request, but is refusing to fulfill it.

code: 403, title: Forbidden

class webob.exc.**HTTPNotFound**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPClientError

This indicates that the server did not find anything matching the Request-URI.

code: 404, title: Not Found

class webob.exc.**HTTPMethodNotAllowed**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPNotAcceptable**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPProxyAuthenticationRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPRequestTimeout**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPConflict**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPGone**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPLengthRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)
subclass of HTTPClientError

This indicates that the the server refuses to accept the request without a defined Content-Length.

code: 411, title: Length Required

class webob.exc.**HTTPPreconditionFailed**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPRequestEntityTooLarge**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPRequestURITooLong**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPUnsupportedMediaType**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPRequestRangeNotSatisfiable**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPExpectationFailed**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPClientError

This indidcates that the expectation given in an Expect request-header field could not be met by this server.

code: 417, title: Expectation Failed

class webob.exc.**HTTPUnprocessableEntity**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPClientError

This indicates that the server is unable to process the contained instructions. Only for WebDAV.

code: 422, title: Unprocessable Entity

class webob.exc.**HTTPLocked**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPClientError

This indicates that the resource is locked. Only for WebDAV

code: 423, title: Locked

class webob.exc.**HTTPFailedDependency**(*detail=None*, *headers=None*, *comment=None*, *body_template=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. Only for WebDAV.

code: 424, title: Failed Dependency

class webob.exc.**HTTPServerError**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for the 500's, where the server is in-error

This is an error condition in which the server is presumed to be in-error. This is usually unexpected, and thus requires a traceback; ideally, opening a support ticket for the customer. Unless specialized, this is a '500 Internal Server Error'

class webob.exc.**HTTPInternalServerError**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

class webob.exc.**HTTPNotImplemented**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPBadGateway**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPServiceUnavailable**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPGatewayTimeout**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPVersionNotSupported**(*detail=None*, *headers=None*, *comment=None*, *body_template=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

class webob.exc.**HTTPInsufficientStorage**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of HTTPServerError

This indicates that the server does not have enough space to save the resource.

code: 507, title: Insufficient Storage

class webob.exc.**HTTPExceptionMiddleware**(*application*)

Middleware that catches exceptions in the sub-application. This does not catch exceptions in the app_iter; only during the initial calling of the application.

This should be put *very close* to applications that might raise these exceptions. This should not be applied globally; letting *expected* exceptions raise through the WSGI stack is dangerous.

27.7.7 webob.headerdict

27.7.8 webob.multidict

Gives a multi-value dictionary object (MultiDict) plus several wrappers

class webob.multidict.**MultiDict**(**args*, ***kw*)

An ordered dictionary that can have multiple values for each key. Adds the methods getall, getone, mixed and extend and add to the normal dictionary interface.

add(*key*, *value*)

Add the key and value, not overwriting any previous value.

dict_of_lists()

Returns a dictionary where each key is associated with a list of values.

classmethod **from_fieldstorage**(*fs*)
Create a dict from a cgi.FieldStorage instance

getall(*key*)
Return a list of all values matching the key (may be an empty list)

getone(*key*)
Get one value matching the key, raising a KeyError if multiple values were found.

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.

classmethod **view_list**(*lst*)
Create a dict that is a view on the given list

class webob.multidict.**NestedMultiDict**(**dicts*)
Wraps several MultiDict objects, treating it as one large MultiDict

class webob.multidict.**NoVars**(*reason=None*)
Represents no variables; used when no variables are applicable.

This is read-only

27.7.9 webob.statusreasons

27.7.10 webob.updatedict

CHAPTER

TWENTYEIGHT

GLOSSARY

action The class method in a Pylons applications' controller that handles a request.

API Application Programming Interface. The means of communication between a programmer and a software program or operating system.

app_globals The `app_globals` object is created on application instantiation by the `Globals` class in a projects `lib/app_globals.py` module.

This object is created once when the application is loaded by the projects `config/environment.py` module (See *Environment*). It remains persistent during the lifecycle of the web application, and is *not* thread-safe which means that it is best used for global options that should be *read-only*, or as an object to attach db connections or other objects which ensure their own access is thread-safe.

c Commonly used alias for *tmpl_context* to save on the typing when using lots of controller populated variables in templates.

caching The storage of the results of expensive or length computations for later re-use at a point more quickly accessed by the end user.

CDN Content Delivery Networks (CDN's) are generally globally distributed content delivery networks optimized for low latency for static file distribution. They can significantly increase page-load times by ensuring that the static resources on a page are delivered by servers geographically close to the client in addition to lightening the load placed on the application server.

ColdFusion Components CFCs represent an attempt by Macromedia to bring ColdFusion closer to an Object Oriented Programming (OOP) language. ColdFusion is in no way an OOP language, but thanks in part to CFCs, it does boast some of the attributes that make OOP languages so popular.

controller The 'C' in MVC. The controller is given a request, does the necessary logic to prepare data for display, then renders a template with the data and returns it to the user. See *Controllers*.

easy_install A tool that lets you download, build, install and manage Python packages and their dependencies. easy_install is the end-user facing component of *setuptools*.

Pylons can be installed with `easy_install`, and applications built with Pylons can easily be deployed this way as well.

See Also:

Pylons *Packaging and Deployment Overview*

egg Python egg's are bundled Python packages, generally installed by a package called *setuptools*. Unlike normal Python package installs, egg's allow a few additional features, such as package dependencies, and dynamic discovery.

See Also:

The Quick Guide to Python Eggs

EJBs Enterprise JavaBeans (EJB) technology is the server-side component architecture for Java Platform, Enterprise Edition (Java EE). EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology.

environ environ is a dictionary passed into all *WSGI* application. It generally contains unparsed header information, CGI style variables and other objects inserted by *WSGI Middleware*.

ETag An ETag (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL. See http://wikipedia.org/wiki/HTTP_ETag

g Alias used in prior versions of Pylons for *app_globals*.

Google App Engine A cloud computing platform for hosting web applications implemented in Python. Building Pylons applications for App Engine is facilitated by Ian Bicking's appengine-monkey project.

See Also:

What is Google App Engine? - Official Doc

h The helpers reference, `h`, is made available for use inside templates to assist with common rendering tasks. `h` is just a reference to the `lib/helpers.py` module and can be used in the same manner as any other module import.

Model-View-Controller An architectural pattern used in software engineering. In Pylons, the MVC paradigm is extended slightly with a pipeline that may transform and extend the data available to a controller, as well as the Pylons *WSGI* app itself that determines the appropriate Controller to call.

See Also:

MVC at Wikipedia

MVC See *Model-View-Controller*

ORM (Object-Relational Mapper) Maps relational databases such as MySQL, Postgres, Oracle to objects providing a cleaner API. Most ORM's also make it easier to prevent SQL Injection attacks by binding variables, and can handle generating sometimes extensive SQL.

Pylons A Python-based WSGI oriented web framework.

Rails Abbreviated as RoR, Ruby on Rails (also referred to as just Rails) is an open source Web application framework, written in Ruby

request Refers to the current request being processed. Available to import from `pylons` and is available for use in templates by the same name. See `Request`.

response Refers to the response to the current request. Available to import from `pylons` and is available for use in template by the same name. See `Response`.

route Routes determine how the URL's are mapped to the controllers and which URL is generated. See *URL Configuration*

setuptools An extension to the basic distutils, setuptools allows packages to specify package dependencies and have dynamic discovery of other installed Python packages.

See Also:

Building and Distributing Packages with setuptools

SQLAlchemy One of the most popular Python database object-relational mappers (*ORM*). SQLAlchemy is the default ORM recommended in Pylons. SQLAlchemy at the ORM level can look similar to Rails ActiveRecord, but uses the DataMapper pattern for additional flexibility with the ability to map simple to extremely complex databases.

tmpl_context The `tmpl_context` is available in the `pylons` module, and refers to the template context. Objects attached to it are available in the template namespace as either `tmpl_context` or `c` for convenience.

UI User interface. The means of communication between a person and a software program or operating system.

virtualenv A tool to create isolated Python environments, designed to supersede the `workingenv` package and virtual python configurations. In addition to isolating packages from possible system conflicts, virtualenv makes it easy to install Python libraries using *easy_install* without dumping lots of packages into the system-wide Python.

The other great benefit is that no root access is required since all modules are kept under the desired directory. This makes it easy to setup a working Pylons install on shared hosting providers and other systems where system-wide access is unavailable.

`virtualenv` is employed automatically by the `go-pylons.py` script described in *Getting Started*. The Pylons wiki has more information on working with virtualenv.

web server gateway interface A specification for web servers and application servers to communicate with web applications. Also referred to by its initials, as *WSGI*.

WSGI The WSGI Specification, also commonly referred to as PEP 333 and described by **PEP 333**.

WSGI Middleware *WSGI* Middleware refers to the ability of WSGI applications to modify the environ, and/or the content of other WSGI applications by being placed in between the request and the other WSGI application.

See Also:

WSGI Middleware in Concepts of Pylons WSGI Middleware Configuration