# pyramid_handlers Documentation

***Release 0.5***

**Repoze Developers**

**August 16, 2017**

# Contents

CHAPTER 1

# Overview

`pyramid_handlers` is a package which allows Pyramid to largely emulate the functionality of *Pylons* "controllers". Handlers are a synthesis of Pyramid *url dispatch* and method introspection of a view class that makes it easier to create bundles of view logic which reacts to particular route patterns.

`pyramid_handlers` works under Python 2.6 and 2.7. It also works under Python 3.2, but *ZCML* support is not available under Python 3.2.

CHAPTER 2

# Installation

Install using setuptools, e.g. (within a virtualenv):

```
$ easy_install pyramid_handlers
```

# CHAPTER 3

## Setup

Once `pyramid_handlers` is installed, you must use the `config.include` mechanism to include it into your Pyramid project's configuration. In your Pyramid project's `__init__.py`:

```
config = Configurator(.....)
config.include('pyramid_handlers')
```

At this point, it will be possible to use the *`pyramid_handlers.add_handler()`* function as a method of the configurator, ala:

```
config.add_handler(....)
```

# CHAPTER 4

## Handler Registration Using `add_handler()`

`pyramid_handlers` provides the special concept of a *view handler*. View handlers are view classes that implement a number of methods, each of which is a *view callable* as a convenience for *URL dispatch* users.

**Note:** View handlers are *not* useful when using *traversal*, only when using *url dispatch*.

Using a view handler instead of a plain function or class *view callable* makes it unnecessary to call `pyramid.config.Configurator.add_route()` (and/or `pyramid.config.Configurator.add_view()`) "by hand" multiple times, making it more pleasant to register a collection of views as a single class when using *url dispatch*. The view handler machinery also introduces the concept of an `action`, which is used as a *view predicate* to control which method of the handler is called. The method name is the default *action name* of a handler view callable.

The concept of a view handler is analogous to a "controller" in Pylons 1.0.

The view handler class is initialized by Pyramid in the same manner as a "plain" view class. Its `__init__` is called with a request object (see class_as_view). It implements methods, each of which is a *view callable*. When a request enters the system which corresponds with an *action* related to one of its view callable methods, this method is called, and it is expected to return a response.

Here's an example view handler class:

```
from pyramid_handlers import action

from pyramid.response import Response

class Hello(object):
    def __init__(self, request):
        self.request = request

    def index(self):
        return Response('Hello world!')

    @action(renderer="mytemplate.mak")
    def bye(self):
        return {}
```

The *pyramid_handlers.action* decorator is used to fine-tune the view parameters for each potential view callable which is a method of the handler.

Handlers are added to application configuration via the *pyramid_handlers.add_handler()* API, which is accessible after configuration as the method `pyramid.config.Configurator.add_handler`. This function will scan a *view handler* class and automatically set up view configurations for its methods that represent "auto-exposed" view callable, or those that were decorated explicitly with the *action* decorator. This decorator is used to setup additional view configuration information for individual methods of the class, and can be used repeatedly for a single view method to register multiple view configurations for it.

```
from myapp.handlers import Hello
config.add_handler('hello', '/hello/{action}', handler=Hello)
```

This example will result in a route being added for the pattern `/hello/{action}`, and each method of the `Hello` class will then be examined to see if it should be registered as a potential view callable when the `/hello/{action}` pattern matches. The value of `{action}` in the route pattern will be used to determine which view should be called, and each view in the class will be setup with a view predicate that requires a specific `action` name. By default, the action name for a method of a handler is the method name.

If the URL was `/hello/index`, the above example pattern would match, and, by default, the `index` method of the `Hello` class would be called.

Alternatively, the action can be declared specifically for a URL to be registered for a *specific* `action` name:

```
from myapp.handlers import Hello
config.add_handler('hello_index', '/hello/index',
                   handler=Hello, action='index')
```

This will result one of the methods that are configured for the `action` of 'index' in the `Hello` handler class to be called. In this case the name of the method is the same as the action name: `index`. However, this need not be the case, as we will see below.

When calling *pyramid_handlers.add_handler()*, an `action` is required in either the route pattern or as a keyword argument, but **cannot appear in both places**. A `handler` argument must also be supplied, which can be either a *asset specification* or a Python reference to the handler class. Additional keyword arguments are passed directly through to `pyramid.config.Configurator.add_route()`.

For example:

```
config.add_handler('hello', '/hello/{action}',
                   handler='mypackage.handlers.MyHandler')
```

Multiple `add_handler()` calls can specify the same handler, to register specific route names for different handler/action combinations. For example:

```
config.add_handler('hello_index', '/hello/index',
                   handler=Hello, action='index')
config.add_handler('bye_index', '/hello/bye',
                   handler=Hello, action='bye')
```

**Note:** Handler configuration may also be added to the system via *ZCML* (see *Configuring a Handler via ZCML*).

CHAPTER 5

# View Setup in the Handler Class

A handler class can have a single class level attribute called `__autoexpose__` which should be a regular expression or the value `None`. It's used to determine which method names will result in additional view configurations being registered.

When *`pyramid_handlers.add_handler()`* runs, every method in the handler class will be searched and a view registered if the method name matches the `__autoexpose__` regular expression, or if the method was decorated with *`action`*.

Every method in the handler class that has a name meeting the `__autoexpose__` regular expression will have a view registered for an `action` name corresponding to the method name. This functionality can be disabled by setting the `__autoexpose__` attribute to `None`:

```
from pyramid_handlers import action

class Hello(object):
    __autoexpose__ = None

    def __init__(self, request):
        self.request = request

    @action()
    def index(self):
        return Response('Hello world!')

    @action(renderer="mytemplate.mak")
    def bye(self):
        return {}
```

With auto-expose effectively disabled, no views will be registered for a method unless it is specifically decorated with *`action`*.

## Action Decorators in a Handler

The *action* decorator registers view configuration information on the handler method, which is used by *add_handler()* to setup the view configuration.

All keyword arguments are recorded, and passed to `add_view()`. Any valid keyword arguments for `add_view()` can thus be used with the *action* decorator to further restrict when the view will be called.

One important difference is that a handler method can respond to an `action` name that is different from the method name by passing in a `name` argument.

Example:

```
from pyramid_handlers import action

class Hello(object):
    def __init__(self, request):
        self.request = request

    @action(name='index', renderer='created.mak', request_method='POST')
    def create(self):
        return {}

    @action(renderer="view_all.mak", request_method='GET')
    def index(self):
        return {}
```

This will register two views that require the `action` to be `index`, with the additional view predicate requiring a specific request method.

It can be useful to decorate a single method multiple times with *action*. Each action decorator will register a new view for the method. By specifying different names and renderers for each action, the same view logic can be exposed and rendered differently on multiple URLs.

Example:

```
from pyramid_handlers import action

class Hello(object):
    def __init__(self, request):
        self.request = request

    @action(name='home', renderer='home.mak')
    @action(name='about', renderer='about.mak')
    def show_template(self):
        # prep some template vars
        return {}

# in the config
config.add_handler('hello', '/hello/{action}', handler=Hello)
```

With this configuration, the url `/hello/home` will find a view configuration that results in calling the `show_template` method, then rendering the template with `home.mak`, and the url `/hello/about` will call the same method and render the `about.mak` template.

CHAPTER 6

# Handler `__action_decorator__` Attribute

**Note:** In a Pylons 1.0 controller, it was possible to override the `__call__()` method, which allowed a developer to "wrap" the entire action invocation, with a try/except or any other arbitrary code. In Pyramid, this can be emulated with the use of an `__action_decorator__` classmethod on your handler class.

If a handler class has an `__action_decorator__` attribute, then the value of the class attribute will be passed in as the `decorator` argument every time a handler action is registered as a view callable. This means that, like anything passed to `add_view()` as the `decorator` argument, `__action_decorator__` must be a callable accepting a single argument. This argument will itself be a callable accepting `(context, request)` arguments, and `__action_decorator__` must return a replacement callable with the same call signature.

Note that, since handler actions are registered as views against the handler class and not a handler instance, any `__action_decorator__` attribute must *not* be a regular instance method. Defining an `__action_decorator__` instance method on a handler class will result in a `ConfigurationError`. Instead, `__action_decorator__` can be any other type of callable: a staticmethod, classmethod, function, or some sort of callable instance.

The below example uses an `__action_decorator__` which is a staticmethod of the handler class. It wraps every view callable implied by the handler in a decorator function which calls the original view callable, but catches a special exception and returns a response.

```
from pyramid_handlers import action
from pyramid.response import Response

class MySpecialException(Exception):
    pass

class MyHandler(object):
    def __init__(self, request):
        self.request = request

    @staticmethod
    def __action_decorator__(view):
        def decorated_view(context, request):
```

```
            try:
                return view(context, request)
            except MySpecialException:
                return Response('Something bad happened', status=500)
        return decorated_view

    @action(renderer='index.html')
    def index(self):
        raise MySpecialException
```

When the `index` method of the above example handler is invoked, it will raise `MySpecialException`. As a result, the action decorator will cath this exception and turn it into a response.

# CHAPTER 7

# Configuration Knobs

If your handler action methods that have characters in them (such as underscores) that you don't find appropriate in a URL, such as `a_method_with_underscores`:

```
# in a module named mypackage.handlers

from pyramid_handlers import action

class AHandler(object):
    def __init__(self, request):
        self.request = request

    @action(renderer='some/renderer.pt')
    def a_method_with_underscores(self):
        return {}
```

And there is some regular transform you can perform against all action method registrations (such as converting the underscores to dashes), you can define a "method name transformer":

```
# in the same module named mypackage.handlers

def transformer(method_name):
    return method_name.replace('_', '-')
```

You can then use the method name transformer in your Pyramid `settings` via the *.ini*' file:

```
[app:myapp]
...
pyramid_handlers.method_name_xformer = mypackage.handlers.transformer
```

Or directly in your `main()` function:

```
# in a module named mypackage.handlers

from mypackage.handlers import transformer

```

```
def main(global_conf, *settings):
    settings['pyramid_handlers.method_name_xformer'] = transformer
    config = Configurator(settings=settings)
    # .. rest of configuration ...
```

Once you've set up a method name transformer, any `{action}` substitution in the pattern associated with a handler will be matched against the transformed method name value instead of the untransformed method name value:

```
# in a module named mypackage.handlers

from mypackage.handlers import transformer
from mypackage.handlers import AHandler

def main(global_conf, *settings):
    settings['pyramid_handlers.method_name_xformer'] = transformer
    config = Configurator(settings=settings)
    config.add_handler('ahandler', '/ahandler/{action}', handler=AHandler)
    # .. rest of configuration ...
```

Now, when `/ahandler/a-method-with-underscores` is visited, it will invoke the `AHandler.a_method_with_underscores` method. Note that `/ahandler/a_method_with_underscores` will however no longer work to invoke the method.

CHAPTER 8

# More Information

## pyramid_handlers API

pyramid_handlers.**add_handler**(*self*, *route_name*, *pattern*, *handler*, *action=None*, ***kw*)

Add a Pylons-style view handler. This function adds a route and some number of views based on a handler object (usually a class).

This function should never be called directly; instead the `pyramid_handlers.includeme` function should be used to include this function into an application; the function will thereafter be available as a method of the resulting configurator.

`route_name` is the name of the route (to be used later in URL generation).

`pattern` is the matching pattern, e.g. `'/blog/{action}'`.

`handler` is a dotted name of (or direct reference to) a Python handler class, e.g. `'my.package.handlers.MyHandler'`.

If `{action}` or `:action` is in the pattern, the exposed methods of the handler will be used as views.

If `action` is passed, it will be considered the method name of the handler to use as a view.

Passing both `action` and having an `{action}` in the route pattern is disallowed.

Any extra keyword arguments are passed along to `add_route`.

See views_chapter for more explanatory documentation.

**class** pyramid_handlers.**action**(***kw*)

Decorate a method for registration by *add_handler()*.

Keyword arguments are identical to `view_config`, with the exception to how the `name` argument is used.

**name** Designate an alternate action name, rather than the default behavior of registering a view with the action name being set to the methods name.

# Configuring a Handler via ZCML

Instead of using the imperative `pyramid.config.Configurator.add_handler()` method to add a new route, you can alternately use *ZCML*.

**Warning:** ZCML works under Python 2.6 and 2.7; it, however, does not work under Python 3.2 or any other version of Python 3.

*Using The handler ZCML Directive* statements in a *ZCML* file used by your application is a sign that you're using *URL dispatch*. For example, the following *ZCML declaration* causes a route to be added to the application.

```
<handler
  route_name="myroute"
  pattern="/prefix/{action}"
  handler=".handlers.MyHandler"
 />
```

**Note:** Values prefixed with a period (`.`) within the values of ZCML attributes such as the `handler` attribute of a `handler` directive mean "relative to the Python package directory in which this *ZCML* file is stored". So if the above `handler` declaration was made inside a `configure.zcml` file that lived in the `hello` package, you could replace the relative `.views.MyHandler` with the absolute `hello.views.MyHandler` Either the relative or absolute form is functionally equivalent. It's often useful to use the relative form, in case your package's name changes. It's also shorter to type.

The order that the routes attached to handlers are evaluated when declarative configuration is used is the order that they appear relative to each other in the ZCML file.

See *Using The handler ZCML Directive* for full `handler` ZCML directive documentation.

## Using The `handler` ZCML Directive

The `handler` directive adds the configuration of a *view handler* to the *application registry*. It is a declarative analogue of the *`pyramid_handlers.add_handler()`* directive.

### Example

Do the following from within a Pyramid application to use the `handler` ZCML directive.

```
<include package="pyramid_handlers" file="meta.zcml"/>

<handler
   route_name="foo"
   pattern="/foo/{action}"
   handler="some.module.SomeClass"/>
```

### Attributes

**`route_name`** The name of the route, e.g. `myroute`. This attribute is required. It must be unique among all defined handler and route names in a given configuration.

**pattern** The pattern of the route e.g. `ideas/{idea}`. This attribute is required. See route_pattern_syntax for information about the syntax of route patterns. The name `{action}` is treated specially in handler patterns. See *Handler Registration Using add_handler()* for a discussion of how `{action}` in handler patterns is treated.

**handler** A *dotted Python name* to the handler class.

**action** If the action name is not specified in the `pattern`, use this name as the handler action (method name).

**factory** The *dotted Python name* to a function that will generate a Pyramid context object when the associated route matches. e.g. `mypackage.resources.MyResource`. If this argument is not specified, a default root factory will be used.

**xhr** This value should be either `True` or `False`. If this value is specified and is `True`, the *request* must possess an `HTTP_X_REQUESTED_WITH` (aka `X-Requested-With`) header for this route to match. This is useful for detecting AJAX requests issued from jQuery, Prototype and other Javascript libraries. If this predicate returns false, route matching continues.

**traverse** If you would like to cause the context to be something other than the root object when this route matches, you can spell a traversal pattern as the `traverse` argument. This traversal pattern will be used as the traversal path: traversal will begin at the root object implied by this route (either the global root, or the object returned by the `factory` associated with this route).

The syntax of the `traverse` argument is the same as it is for `pattern`. For example, if the `pattern` provided to the `route` directive is `articles/{article}/edit`, and the `traverse` argument provided to the `route` directive is `/{article}`, when a request comes in that causes the route to match in such a way that the `article` match value is '1' (when the request URI is `/articles/1/edit`), the traversal path will be generated as `/1`. This means that the root object's `__getitem__` will be called with the name `1` during the traversal phase. If the `1` object exists, it will become the context of the request. traversal_chapter has more information about traversal.

If the traversal path contains segment marker names which are not present in the `pattern` argument, a runtime error will occur. The `traverse` pattern should not contain segment markers that do not exist in the `pattern`.

A similar combining of routing and traversal is available when a route is matched which contains a `*traverse` remainder marker in its `pattern` (see using_traverse_in_a_route_pattern). The `traverse` argument to the `route` directive allows you to associate route patterns with an arbitrary traversal path without using a a `*traverse` remainder marker; instead you can use other match information.

Note that the `traverse` argument to the `handler` directive is ignored when attached to a route that has a `*traverse` remainder marker in its pattern.

**request_method** A string representing an HTTP method name, e.g. `GET`, `POST`, `HEAD`, `DELETE`, `PUT`. If this argument is not specified, this route will match if the request has *any* request method. If this predicate returns false, route matching continues.

**path_info** The value of this attribute represents a regular expression pattern that will be tested against the `PATH_INFO` WSGI environment variable. If the regex matches, this predicate will be true. If this predicate returns false, route matching continues.

**request_param** This value can be any string. A view declaration with this attribute ensures that the associated route will only match when the request has a key in the `request.params` dictionary (an HTTP `GET` or `POST` variable) that has a name which matches the supplied value. If the value supplied to the attribute has a = sign in it, e.g. `request_params="foo=123"`, then the key (`foo`) must both exist in the `request.params` dictionary, and the value must match the right hand side of the expression (`123`) for the route to "match" the current request. If this predicate returns false, route matching continues.

**header** The value of this attribute represents an HTTP header name or a header name/value pair. If the value contains a `:` (colon), it will be considered a name/value pair (e.g. `User-Agent:Mozilla/.*` or `Host:localhost`). The *value* of an attribute that represent a name/value pair should be a regular expression. If the value does not contain a colon, the entire value will be considered to be the header name (e.g.

If-Modified-Since). If the value evaluates to a header name only without a value, the header specified by the name must be present in the request for this predicate to be true. If the value evaluates to a header name/value pair, the header specified by the name must be present in the request *and* the regular expression specified as the value must match the header value. Whether or not the value represents a header name or a header name/value pair, the case of the header name is not significant. If this predicate returns false, route matching continues.

**accept** The value of this attribute represents a match query for one or more mimetypes in the Accept HTTP request header. If this value is specified, it must be in one of the following forms: a mimetype match token in the form text/plain, a wildcard mimetype match token in the form text/* or a match-all wildcard mimetype match token in the form */*. If any of the forms matches the Accept header of the request, this predicate will be true. If this predicate returns false, route matching continues.

**custom_predicates** This value should be a sequence of references to custom predicate callables. Use custom predicates when no set of predefined predicates does what you need. Custom predicates can be combined with predefined predicates as necessary. Each custom predicate callable should accept two arguments: info and request and should return either True or False after doing arbitrary evaluation of the info and/or the request. If all custom and non-custom predicate callables return True the associated route will be considered viable for a given request. If any predicate callable returns False, route matching continues. Note that the value info passed to a custom route predicate is a dictionary containing matching information; see custom_route_predicates for more information about info.

### Alternatives

You can also add a *route configuration* via:

- Using the pyramid.config.Configurator.add_handler() method.

### See Also

See also views_chapter.

## Glossary

**application registry** A registry of configuration information consulted by Pyramid while servicing an application. An application registry maps resource types to views, as well as housing other application-specific component registrations. Every Pyramid application has one (and only one) application registry.

**asset** Any file contained within a Python *package* which is *not* a Python source code file.

**asset specification** A colon-delimited identifier for an *asset*. The colon separates a Python *package* name from a package subpath. For example, the asset specification my.package:static/baz.css identifies the file named baz.css in the static subdirectory of the my.package Python *package*. See asset_specifications for more info.

**asset specification** A colon-delimited identifier for an *asset*. The colon separates a Python *package* name from a package subpath. For example, the asset specification my.package:static/baz.css identifies the file named baz.css in the static subdirectory of the my.package Python *package*. See asset_specifications for more info.

**configuration declaration** An individual method call made to an instance of a Pyramid *Configurator* object which performs an arbitrary action, such as registering a *view configuration* (via the add_view method of the configurator) or *route configuration* (via the add_route method of the configurator).

**configuration decoration** Metadata implying one or more *configuration declaration* invocations. Often set by configuration Python *decorator* attributes, such as pyramid.view.view_config, aka @view_config.

**configurator** An object used to do *configuration declaration* within an application. The most common configurator is an instance of the `pyramid.config.Configurator` class.

**decorator** A wrapper around a Python function or class which accepts the function or class as its first argument and which returns an arbitrary object. Pyramid provides several decorators, used for configuration and return value modification purposes. See also PEP 318.

**dotted Python name** A reference to a Python object by name using a string, in the form `path.to.modulename:attributename`. Often used in Paste and setuptools configurations. A variant is used in dotted names within configurator method arguments that name objects (such as the "add_view" method's "view" and "context" attributes): the colon (`:`) is not used; in its place is a dot.

**imperative configuration** The configuration mode in which you use Python to call methods on a *Configurator* in order to add each *configuration declaration* required by your application.

**module** A Python source file; a file on the filesystem that typically ends with the extension `.py` or `.pyc`. Modules often live in a *package*.

**package** A directory on disk which contains an `__init__.py` file, making it recognizable to Python as a location which can be `import` -ed. A package exists to contain *module* files.

**Pylons** A lightweight Python web framework.

**Pyramid** A web framework.

**request** A `WebOb` request object. See webob_chapter (narrative) and request_module (API documentation) for information about request objects.

**root factory** The "root factory" of an Pyramid application is called on every request sent to the application. The root factory returns the traversal root of an application. It is conventionally named `get_root`. An application may supply a root factory to Pyramid during the construction of a *Configurator*. If a root factory is not supplied, the application uses a default root object. Use of the default root object is useful in application which use *URL dispatch* for all URL-to-view code mappings.

**route** A single pattern matched by the *url dispatch* subsystem, which generally resolves to one or more *view callable* objects. See also *url dispatch*.

**route configuration** Route configuration is the act of using *imperative configuration* or a *ZCML* `<route>` statement to associate request parameters with a particular *route* using pattern matching and *route predicate* statements. See urldispatch_chapter for more information about route configuration.

**route predicate** An argument to a *route configuration* which implies a value that evaluates to `True` or `False` for a given *request*. All predicates attached to a *route configuration* must evaluate to `True` for the associated route to "match" the current request. If a route does not match the current request, the next route (in definition order) is attempted.

**router** The *WSGI* application created when you start a Pyramid application. The router intercepts requests, invokes traversal and/or URL dispatch, calls view functions, and returns responses to the WSGI server on behalf of your Pyramid application.

**scan** The term used by Pyramid to define the process of importing and examining all code in a Python package or module for *configuration decoration*.

**traversal** The act of descending "up" a tree of resource objects from a root resource in order to find a context resource. The Pyramid *router* performs traversal of resource objects when a *root factory* is specified. See the traversal_chapter chapter for more information. Traversal can be performed *instead* of *URL dispatch* or can be combined *with* URL dispatch. See hybrid_chapter for more information about combining traversal and URL dispatch (advanced).

**URL dispatch** An alternative to *traversal* as a mechanism for locating a a *view callable*. When you use a *route* in your Pyramid application via a *route configuration*, you are using URL dispatch. See the urldispatch_chapter for more information.

**view** Common vernacular for a *view callable*.

**view callable** A "view callable" is a callable Python object which is associated with a *view configuration*; it returns a response object . A view callable accepts a single argument: `request`, which will be an instance of a *request* object. A view callable is the primary mechanism by which a developer writes user interface code within Pyramid. See views_chapter for more information about Pyramid view callables.

**view configuration** View configuration is the act of associating a *view callable* with configuration information. This configuration information helps map a given *request* to a particular view callable and it can influence the response of a view callable. Pyramid views can be configured via *imperative configuration*, *ZCML* or by a special `@view_config` decorator coupled with a *scan*. See view_config_chapter for more information about view configuration.

**View handler** A view handler ties together `pyramid.config.Configurator.add_route()` and `pyramid.config.Configurator.add_view()` to make it more convenient to register a collection of views as a single class when using *url dispatch*. See also views_chapter.

**view predicate** An argument to a *view configuration* which evaluates to `True` or `False` for a given *request*. All predicates attached to a view configuration must evaluate to true for the associated view to be considered as a possible callable for a given request.

**WSGI** Web Server Gateway Interface. This is a Python standard for connecting web applications to web servers, similar to the concept of Java Servlets. Pyramid requires that your application be served as a WSGI application.

**ZCML** Zope Configuration Markup Language, an XML dialect used by Zope and Pyramid for configuration tasks. ZCML is capable of performing different types of *configuration declaration*, but its primary purpose in Pyramid is to perform *view configuration* and *route configuration* within the `configure.zcml` file in a Pyramid application. You can use ZCML as an alternative to *imperative configuration*.

**ZCML declaration** The concrete use of a *ZCML directive* within a ZCML file.

**ZCML directive** A ZCML "tag" such as `<view>`, `<route>`, or `<handler>`.

CHAPTER 9

# Reporting Bugs / Development Versions

Visit http://github.com/Pylons/pyramid_handlers to download development or tagged versions.

Visit http://github.com/Pylons/pyramid_handlers/issues to report bugs.

CHAPTER 10

# Indices and tables

- *Glossary*
- genindex
- modindex
- search

# Python Module Index

## p

# Index