Pyramid Cookbook

Release 0.2

Pylons Project Contributors

July 03, 2015

The Pyramid Cookbook presents topical, practical "recipes" of using `Pyramid`. It supplements the main documentation.

To contribute your recipe to the Pyramid Cookbook, read Contributing.

CHAPTER 1

Table of contents

1.1 Authentication and Authorization

1.1.1 Basic Authentication Policy

Here's an implementation of an HTTP basic auth Pyramid authentication policy:

```
import binascii

from paste.httpheaders import AUTHORIZATION
from paste.httpheaders import WWW_AUTHENTICATE

from pyramid.security import Everyone
from pyramid.security import Authenticated

def _get_basicauth_credentials(request):
    authorization = AUTHORIZATION(request.environ)
    try:
        authmeth, auth = authorization.split(' ', 1)
    except ValueError:  # not enough values to unpack
        return None
    if authmeth.lower() == 'basic':
        try:
            auth = auth.strip().decode('base64')
        except binascii.Error:  # can't decode
            return None
        try:
            login, password = auth.split(':', 1)
        except ValueError:  # not enough values to unpack
            return None
        return {'login': login, 'password': password}

    return None

class BasicAuthenticationPolicy(object):
    """ A :app:`Pyramid` :term:`authentication policy` which
    obtains data from basic authentication headers.

    Constructor Arguments

    ``check``

```

```
        A callback passed the credentials and the request,
        expected to return None if the userid doesn't exist or a sequence
        of group identifiers (possibly empty) if the user does exist.
        Required.

    ``realm``

        Default: ``Realm``.  The Basic Auth realm string.

    """

    def __init__(self, check, realm='Realm'):
        self.check = check
        self.realm = realm

    def authenticated_userid(self, request):
        credentials = _get_basicauth_credentials(request)
        if credentials is None:
            return None
        userid = credentials['login']
        if self.check(credentials, request) is not None:  # is not None!
            return userid

    def effective_principals(self, request):
        effective_principals = [Everyone]
        credentials = _get_basicauth_credentials(request)
        if credentials is None:
            return effective_principals
        userid = credentials['login']
        groups = self.check(credentials, request)
        if groups is None:  # is None!
            return effective_principals
        effective_principals.append(Authenticated)
        effective_principals.append(userid)
        effective_principals.extend(groups)
        return effective_principals

    def unauthenticated_userid(self, request):
        creds = _get_basicauth_credentials(request)
        if creds is not None:
            return creds['login']
        return None

    def remember(self, request, principal, **kw):
        return []

    def forget(self, request):
        head = WWW_AUTHENTICATE.tuples('Basic realm="%s"' % self.realm)
        return head
```

Use it something like:

```
def mycheck(credentials, request):
    pwd_ok = my_password_check(credentials['login'], credentials['password'])
    if not pwd_ok:
        return None
    return ['groups', 'that', 'login', 'is', 'member', 'of']

config = Configurator(
                authentication_policy=BasicAuthenticationPolicy(mycheck))
```

1.1.2 Custom Authentication Policy

Here is an example of a custom AuthenticationPolicy, based off of the native `AuthTktAuthenticationPolicy`, but with added groups support. This example implies you have a `user` attribute on your request (see *Making A "User Object" Available as a Request Attribute*) and that the `user` should have a `groups` relation on it:

```python
from pyramid.authentication import AuthTktCookieHelper
from pyramid.security import Everyone, Authenticated

class MyAuthenticationPolicy(object):

    def __init__(self, settings):
        self.cookie = AuthTktCookieHelper(
            settings.get('auth.secret'),
            cookie_name=settings.get('auth.token') or 'auth_tkt',
            secure=asbool(settings.get('auth.secure')),
            timeout=asint(settings.get('auth.timeout')),
            reissue_time=asint(settings.get('auth.reissue_time')),
            max_age=asint(settings.get('auth.max_age')),
        )

    def remember(self, request, principal, **kw):
        return self.cookie.remember(request, principal, **kw)

    def forget(self, request):
        return self.cookie.forget(request)

    def unauthenticated_userid(self, request):
        result = self.cookie.identify(request)
        if result:
            return result['userid']

    def authenticated_userid(self, request):
        if request.user:
            return request.user.id

    def effective_principals(self, request):
        principals = [Everyone]
        user = request.user
        if user:
            principals += [Authenticated, 'u:%s' % user.id]
            principals.extend(('g:%s' % g.name for g in user.groups))
        return principals
```

Thanks to *raydeo* for this one.

1.1.3 Making A "User Object" Available as a Request Attribute

This is you: your application wants a "user object". Pyramid is only willing to supply you with a user *id* (via `pyramid.security.authenticated_userid()`). You don't want to create a function that accepts a request object and returns a user object from your domain model for efficiency reasons, and you want the user object to be omnipresent as `request.user`.

You've tried using a `NewRequest` subscriber to attach a user object to the request, but the `NewRequest` susbcriber is called on every request, even ones for static resources, and this bothers you (which it should).

A lazy property can be registered to the request via the `pyramid.config.Configurator.add_request_method()` API (introduced in Pyramid 1.4; see below for older releases). This allows you to specify a callable that will be available on the request object, but will not actually execute the function until accessed. The result of this function can also be cached per-request, to eliminate the overhead of running the function multiple times (this is done by setting `reify=True`:

```
from pyramid.security import unauthenticated_userid

def get_user(request):
    # the below line is just an example, use your own method of
    # accessing a database connection here (this could even be another
    # request property such as request.db, implemented using this same
    # pattern).
    dbconn = request.registry.settings['dbconn']
    userid = unauthenticated_userid(request)
    if userid is not None:
        # this should return None if the user doesn't exist
        # in the database
        return dbconn['users'].query({'id':userid})
```

Here's how you should add your new request property in configuration code:

```
config.add_request_method(get_user, 'user', reify=True)
```

Then in your view code, you should be able to happily do `request.user` to obtain the "user object" related to that request. It will return `None` if there aren't any user credentials associated with the request, or if there are user credentials associated with the request but the userid doesn't exist in your database. No inappropriate execution of `authenticated_userid` is done (as would be if you used a `NewRequest` subscriber).

After doing such a thing, if your user object has a `groups` attribute, which returns a list of groups that have `name` attributes, you can use the following as a `callback` (aka `groupfinder`) argument to most builtin authentication policies. For example:

```
from pyramid.authentication import AuthTktAuthenticationPolicy

def groupfinder(userid, request):
    user = request.user
    if user is not None:
        return [ group.name for group in request.user.groups ]
    return None

authn_policy = AuthTktAuthenticationPolicy('seekrITT', callback=groupfinder)
```

Prior to Pyramid 1.4

If you are using version 1.3, you can follow the same procedure as above, except use this instead of `add_request_method`:

```
config.set_request_property(get_user, 'user', reify=True)
```

Deprecated since version 1.4: `set_request_property()`

Prior to `set_request_property` and `add_request_method`, a similar pattern could be used, but it required registering a new request factory via `set_request_factory()`. This works in the same way, but each application can only have one request factory and so it is not very extensible for arbitrary properties.

The code for this method is below:

```python
from pyramid.decorator import reify
from pyramid.request import Request
from pyramid.security import unauthenticated_userid

class RequestWithUserAttribute(Request):
    @reify
    def user(self):
        # <your database connection, however you get it, the below line
        # is just an example>
        dbconn = self.registry.settings['dbconn']
        userid = unauthenticated_userid(self)
        if userid is not None:
            # this should return None if the user doesn't exist
            # in the database
            return dbconn['users'].query({'id':userid})
```

Here's how you should use your new request factory in configuration code:

```python
config.set_request_factory(RequestWithUserAttribute)
```

1.1.4 Wiki Flow of Authentication

This tutorial describes the "flow of authentication" of the result of the completing the Adding authorization tutorial chapter from the main Pyramid documentation.

This text was contributed by John Shipman.

Overall flow of an authentication

Now that you have seen all the pieces of the authentication mechanism, here are some examples that show how they all work together.

1. Failed login: The user requests `/FrontPage/edit_page`. The site presents the login form. The user enters `editor` as the login, but enters an invalid password `bad`. The site redisplays the login form with the message "Failed login". See *Failed login*.
2. The user again requests `/FrontPage/edit_page`. The site presents the login form, and this time the user enters login `editor` and password `editor`. The site presents the edit form with the content of `/FrontPage`. The user makes some changes and saves them. See *Successful login*.
3. The user again revisits `/FrontPage/edit_page`. The site goes immediately to the edit form without requesting credentials. See *Revisiting after authentication*.
4. The user clicks the `Logout` link. See *Logging out*.

Failed login

The process starts when the user enters URL `http://localhost:6543/FrontPage/edit_page`. Let's assume that this is the first request ever made to the application and the page database is empty except for the `Page` instance created for the front page by the `initialize_sql` function in `models.py`.

This process involves two complete request/response cycles.

1. From the front page, the user clicks *Edit page*. The request is to `/FrontPage/edit_page`. The view callable is `login.login`. The response is the `login.pt` template with blank fields.

2. The user enters invalid credentials and clicks *Log in*. A `POST` request is sent to `/FrontPage/edit_page`. The view callable is again `login.login`. The response is the `login.pt` template showing the message "Failed login", with the entry fields displaying their former values.

Cycle 1:

1. During URL dispatch, the route `'/{pagename}/edit_page'` is considered for matching. The associated view has a `view_permission='edit'` permission attached, so the dispatch logic has to verify that the user has that permission or the route is not considered to match.

   The context for all route matching comes from the configured root factory, `RootFactory()` in `models.py`. This class has an `__acl__` attribute that defines the access control list for all routes:

```
__acl__ = [ (Allow, Everyone, 'view'),
            (Allow, 'group:editors', 'edit') ]
```

   In practice, this means that for any route that requires the `edit` permission, the user must be authenticated and have the `group:editors` principal or the route is not considered to match.

2. To find the list of the user's principals, the authorization first policy checks to see if the user has a `paste.auth.auth_tkt` cookie. Since the user has never been to the site, there is no such cookie, and the user is considered to be unauthenticated.

3. Since the user is unauthenticated, the `groupfinder` function in `security.py` is called with `None` as its `userid` argument. The function returns an empty list of principals.

4. Because that list does not contain the `group:editors` principal, the `'/{pagename}/edit_page'` route's `edit` permission fails, and the route does not match.

5. Because no routes match, the *forbidden view* callable is invoked: the `login` function in module `login.py`.

6. Inside the `login` function, the value of `login_url` is `http://localhost:6543/login`, and the value of `referrer` is `http://localhost:6543/FrontPage/edit_page`.

   Because `request.params` has no key for `'came_from'`, the variable `came_from` is also set to `http://localhost:6543/FrontPage/edit_page`. Variables `message`, `login`, and `password` are set to the empty string.

   Because `request.params` has no key for `'form.submitted'`, the `login` function returns this dictionary:

```
{'message': '', 'url':'http://localhost:6543/login',
 'came_from':'http://localhost:6543/FrontPage/edit_page',
 'login':'', 'password':''}
```

7. This dictionary is used to render the `login.pt` template. In the form, the `action` attribute is `http://localhost:6543/login`, and the value of `came_from` is included in that form as a hidden field by this line in the template:

```
<input type="hidden" name="came_from" value="${came_from}"/>
```

Cycle 2:

1. The user enters incorrect credentials and clicks the *Log in* button, which does a `POST` request to URL `http://localhost:6543/login`. The name of the *Log in* button in this form is `form.submitted`.

2. The route with pattern `'/login'` matches this URL, so control is passed again to the `login` view callable.

3. The `login_url` and `referrer` have the same value this time (`http://localhost:6543/login`), so variable `referrer` is set to `'/'`.

   Since `request.params` does have a key `'form.submitted'`, the values of `login` and `password` are retrieved from `request.params`.

Because the login and password do not match any of the entries in the `USERS` dictionary in `security.py`, variable `message` is set to `'Failed login'`.

The view callable returns this dictionary:

```
{'message':'Failed login',
 'url':'http://localhost:6543/login', 'came_from':'/',
 'login':'editor', 'password':'bad'}
```

4. The `login.pt` template is rendered using those values.

Successful login

In this scenario, the user again requests URL `/FrontPage/edit_page`.

This process involves four complete request/response cycles.

1. The user clicks *Edit page*. The view callable is `login.login`. The response is template `login.pt`, with all the fields blank.
2. The user enters valid credentials and clicks *Log in*. The view callable is `login.login`. The response is a redirect to `/FrontPage/edit_page`.
3. The view callable is `views.edit_page`. The response renders template `edit.pt`, displaying the current page content.
4. The user edits the content and clicks *Save*. The view callable is `views.edit_page`. The response is a redirect to `/FrontPage`.

Execution proceeds as in *Failed login*, up to the point where the password `editor` is successfully matched against the value from the `USERS` dictionary.

Cycle 2:

1. Within the `login.login` view callable, the value of `login_url` is `http://localhost:6543/login`, and the value of `referrer` is `'/'`, and `came_from` is `http://localhost:6543/FrontPage/edit_page` when this block is executed:

```
if USERS.get(login) == password:
    headers = remember(request, login)
    return HTTPFound(location=came_from, headers=headers)
```

2. Because the password matches this time, `pyramid.security.remember` returns a sequence of header tuples that will set a `paste.auth.auth_tkt` authentication cookie in the user's browser for the login `'editor'`.
3. The `HTTPFound` exception returns a response that redirects the browser to `http://localhost:6543/FrontPage/edit_page`, including the headers that set the authentication cookie.

Cycle 3:

1. Route pattern `'/{pagename}/edit_page'` matches this URL, but the corresponding view is restricted by an `'edit'` permission.
2. Because the user now has an authentication cookie defining their login name as `'editor'`, the `groupfinder` function is called with that value as its `userid` argument.
3. The `groupfinder` function returns the list `['group:editors']`. This satisfies the access control entry `(Allow, 'group:editors', 'edit')`, which grants the `edit` permission. Thus, this route matches, and control passes to view callable `edit_page`.

4. Within `edit_page`, `name` is set to `'FrontPage'`, the page name from `request.matchdict['pagename']`, and `page` is set to an instance of `models.Page` that holds the current content of `FrontPage`.

5. Since this request did not come from a form, `request.params` does not have a key for `'form.submitted'`.

6. The `edit_page` function calls `pyramid.security.authenticated_userid()` to find out whether the user is authenticated. Because of the cookies set previously, the variable `logged_in` is set to the userid `'editor'`.

7. The `edit_page` function returns this dictionary:

```
{'page':page, 'logged_in':'editor',
 'save_url':'http://localhost:6543/FrontPage/edit_page'}
```

8. Template `edit.pt` is rendered with those values. Among other features of this template, these lines cause the inclusion of a *Logout* link:

```
<span tal:condition="logged_in">
  <a href="${request.application_url}/logout">Logout</a>
</span>
```

   For the example case, this link will refer to `http://localhost:6543/logout`.

   These lines of the template display the current page's content in a form whose `action` attribute is `http://localhost:6543/FrontPage/edit_page`:

```
<form action="${save_url}" method="post">
  <textarea name="body" tal:content="page.data" rows="10" cols="60"/>
  <input type="submit" name="form.submitted" value="Save"/>
</form>
```

Cycle 4:

1. The user edits the page content and clicks *Save*.

2. URL `http://localhost:6543/FrontPage/edit_page` goes through the same routing as before, up until the line that checks whether `request.params` has a key `'form.submitted'`. This time, within the `edit_page` view callable, these lines are executed:

```
page.data = request.params['body']
session.add(page)
return HTTPFound(location = route_url('view_page', request,
                                      pagename=name))
```

   The first two lines replace the old page content with the contents of the `body` text area from the form, and then update the page stored in the database. The third line causes a response that redirects the browser to `http://localhost:6543/FrontPage`.

Revisiting after authentication

In this case, the user has an authentication cookie set in their browser that specifies their login as `'editor'`. The requested URL is `http://localhost:6543/FrontPage/edit_page`.

This process requires two request/response cycles.

1. The user clicks *Edit page*. The view callable is `views.edit_page`. The response is `edit.pt`, showing the current page content.

2. The user edits the content and clicks *Save*. The view callable is `views.edit_page`. The response is a redirect to `/Frontpage`.

Cycle 1:

1. The route with pattern `/{pagename}/edit_page` matches the URL, and because of the authentication cookie, `groupfinder` returns a list containing the `group:editors` principal, which `models.RootFactory.__acl__` uses to grant the `edit` permission, so this route matches and dispatches to the view callable `views.edit_page()`.
2. In `edit_page`, because the request did not come from a form submission, `request.params` has no key for `'form.submitted'`.
3. The variable `logged_in` is set to the login name `'editor'` by calling `authenticated_userid`, which extracts it from the authentication cookie.
4. The function returns this dictionary:

```
{'page':page,
 'save_url':'http://localhost:6543/FrontPage/edit_page',
 'logged_in':'editor'}
```

5. Template `edit.pt` is rendered with the values from that dictionary. Because of the presence of the `'logged_in'` entry, a *Logout* link appears.

Cycle 2:

1. The user edits the page content and clicks *Save*.
2. The `POST` operation works as in *Successful login*.

Logging out

This process starts with a request URL `http://localhost:6543/logout`.

1. The route with pattern `'/logout'` matches and dispatches to the view callable `logout` in `login.py`.
2. The call to `pyramid.security.forget()` returns a list of header tuples that will, when returned with the response, cause the browser to delete the user's authentication cookie.
3. The view callable returns an `HTTPFound` exception that redirects the browser to named route `view_wiki`, which will translate to URL `http://localhost:6543`. It also passes along the headers that delete the authentication cookie.

1.1.5 Auth Tutorial

See Michael Merickel's authentication/authorization tutorial at http://michael.merickel.org/projects/pyramid_auth_demo/ with code on Github at https://github.com/mmerickel/pyramid_auth_demo.

1.1.6 Pyramid Auth with Akhet and SQLAlchemy

Learn how to set up Pyramid authorization using Akhet and SQLAlchemy at http://pyramid.chromaticleaves.com/simpleauth/.

1.1.7 Google, Facebook, Twitter, and any OpenID Authentication

See Wayne Witzel III's blog post about using Velruse and Pyramid together to do Google OAuth authentication.

See Matthew Housden and Chris Davies apex project for any basic and openid authentication such as Google, Facebook, Twitter and more at https://github.com/cd34/apex.

1.1.8 Integration with Enterprise Systems

When using Pyramid within an "enterprise" (or an intranet), it is often desirable to integrate with existing authentication and authorization (entitlement) systems. For example, in Microsoft Network environments, the user database is typically maintained in Active Directory. At present, there is no ready-to-use recipe, but we are listing places that may be worth looking at for ideas when developing one:

Authentication

- adpasswd project on pypi
- Tim Golden's Active Directory Cookbook
- python-ad
- python-ldap.org
- python-ntmlm
- Blog post on managing AD from Python in Linux

Authorization

- Microsoft Authorization Manager
- Fundamentals of WCF Security
- Calling WCF Services from C++ using gSOAP

For basic information on authentication and authorization, see the security section of the Pyramid documentation.

1.2 Configuration

1.2.1 A Whirlwind Tour of Advanced Pyramid Configuration Tactics

Concepts: Configuration, Directives, and Statements

This article attempts to demonstrate some of Pyramid's more advanced startup-time configuration features. The stuff below talks about "configuration", which is a shorthand word I'll use to mean the state that is changed when a developer adds views, routes, subscribers, and other bits. A developer adds configuration by calling configuration *directives*. For example, `config.add_route()` is a configuration directive. A particular *use* of `config.add_route()` is a configuration *statement*. In the below code block, the execution of the `add_route()` directive is a configuration statement. Configuration statements change pending configuration state:

```
config = pyramid.config.Configurator()
config.add_route('home', '/')
```

Here are a few core concepts related to Pyramid startup configuration:

1. Due to the way the configuration statements work, statement ordering is usually irrelevant. For example, calling `add_view`, then `add_route` has the same outcome as calling `add_route`, then `add_view`. There are some important exceptions to this, but in general, unless the documentation for a given configuration directive states otherwise, you don't need to care in what order your code adds configuration statements.
2. When a configuration statement is executed, it usually doesn't do much configuration immediately. Instead, it generates a *discriminator* and produces a *callback*. The discriminator is a hashable value that represents the configuration statement uniquely amongst all other configuration statements. The callback, when eventually called, actually performs the work related to the configuration statement. Pyramid adds the discriminator and the callback into a list of pending actions that may later be *committed*.
3. Pending configuration actions can be committed at any time. At commit time, Pyramid compares each of the discriminators generated by a configuration statement to every other discriminator generated by other configuration statements in the pending actions list. If two or more configuration statements have generated the same discriminator, this is a *conflict*. Pyramid will attempt to resolve the conflict automatically; if it cannot, startup will exit with an error. If all conflicts are resolved, each callback associated with a configuration statement is executed. Per-action sanity-checking is also performed as the result of a commit.
4. Pending actions can be committed more than once during startup in order to avoid a configuration state that contains conflicts. This is useful if you need to perform configuration overrides in a brute-force, deployment-specific way.
5. An application can be created via configuration statements (for example, calls to `add_route` or `add_view`) composed from logic defined in multiple locations. The configuration statements usually live within Python functions. Those functions can live anywhere, as long as they can be imported. If the `config.include()` API is used to stitch these configuration functions together, some configuration conflicts can be automatically resolved.
6. Developers can add *directives* which participate in Pyramid's phased configuration process. These directives can be made to work exactly like "built-in" directives like `add_route` and `add_view`.
7. Application configuration is never added as the result of someone or something just happening to import a Python module. Adding configuration is always more explicit than that.

Let's see some of those concepts in action. Here's one of the simplest possible Pyramid applications:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

If we run this application via `python app.py`, we'll get a `Hello world!` printed when we visit http://localhost:8080/ in a browser. Not very exciting.

What happens when we reorder our configuration statements? We'll change the relative ordering of `add_view()` and `add_route()` configuration statements. Instead of adding a route, then a view, we'll add a view then a route:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_view(hello_world, route_name='home') # moved this up
    config.add_route('home', '/')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

If you start this application, you'll note that, like before, visiting `/` serves up `Hello world!`. In other words, it works exactly like it did before we switched the ordering around. You might not expect this configuration to work, because we're referencing the name of a route (`home`) within our add_view statement (`config.add_view(hello_world, route_name='home')` that hasn't been added yet. When we execute `add_view`, `add_route('home', '/')` has not yet been executed. This out-of-order execution works because Pyramid defers configuration execution until a *commit* is performed as the result of `config.make_wsgi_app()` being called. Relative ordering between `config.add_route()` and `config.add_view()` calls is not important. Pyramid implicitly commits the configuration state when `make_wsgi_app()` gets called; only when it's committed is the configuration state sanity-checked. In particular, in this case, we're relying on the fact that Pyramid makes sure that all route configuration happens before any view configuration at commit time. If a view references a nonexistent route, an error will be raised at commit time rather than at configuration statement execution time.

Sanity Checks

We can see this sanity-checking feature in action in a failure case. Let's change our application, commenting out our call to `config.add_route()` temporarily within `app.py`:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_view(hello_world, route_name='home') # moved this up
    # config.add_route('home', '/') # we temporarily commented this line
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

When we attempt to run this Pyramid application, we get a traceback:

```
Traceback (most recent call last):
  File "app.py", line 12, in <module>
    app = config.make_wsgi_app()
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 955, in make_wsgi_app
```

```
    self.commit()
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 629, in commit
    self.action_state.execute_actions(introspector=self.introspector)
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 1083, in execute_actions
    tb)
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 1075, in execute_actions
    callable(*args, **kw)
  File "/home/chrism/projects/pyramid/pyramid/config/views.py", line 1124, in register
    route_name)
pyramid.exceptions.ConfigurationExecutionError: <class 'pyramid.exceptions.ConfigurationError'>: No
  in:
  Line 10 of file app.py:
    config.add_view(hello_world, route_name='home')
```

It's telling us that we attempted to add a view which references a nonexistent route. Configuration directives sometimes introduce sanity-checking to startup, as demonstrated here.

Configuration Conflicts

Let's change our application once again. We'll undo our last change, and add a configuration statement that attempts to add another view:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

def hi_world(request): # added
    return Response('Hi world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.add_view(hi_world, route_name='home') # added
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

If you notice above, we're now calling `add_view` twice with two different view callables. Each call to `add_view` names the same route name. What happens when we try to run this program now?:

```
Traceback (most recent call last):
  File "app.py", line 17, in <module>
    app = config.make_wsgi_app()
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 955, in make_wsgi_app
    self.commit()
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 629, in commit
    self.action_state.execute_actions(introspector=self.introspector)
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 1064, in execute_actions
    for action in resolveConflicts(self.actions):
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 1192, in resolveConflicts
    raise ConfigurationConflictError(conflicts)
```

```
pyramid.exceptions.ConfigurationConflictError: Conflicting configuration actions
  For: ('view', None, '', 'home', 'd41d8cd98f00b204e9800998ecf8427e')
    Line 14 of file app.py:
        config.add_view(hello_world, route_name='home')
    Line 15 of file app.py:
        config.add_view(hi_world, route_name='home')
```

This traceback is telling us that there was a *configuration conflict* between two configuration statements: the `add_view` statement on line 14 of app.py and the `add_view` statement on line 15 of app.py. This happens because the *discriminator* generated by `add_view` statement on line 14 turned out to be the same as the discriminator generated by the `add_view` statement on line 15. The discriminator is printed above the line conflict output: `For: ('view', None, '', 'home', 'd41d8cd98f00b204e9800998ecf8427e')`.

Note: The discriminator itself has to be opaque in order to service all of the use cases required by `add_view`. It's not really meant to be parsed by a human, and is kinda really printed only for consumption by core Pyramid developers. We may consider changing things in future Pyramid versions so that it doesn't get printed when a conflict exception happens.

Why is this exception raised? Pyramid couldn't work out what you wanted to do. You told it to serve up more than one view for exactly the same set of request-time circumstances ("when the route name matches `home`, serve this view"). This is an impossibility: Pyramid needs to serve one view or the other in this circumstance; it can't serve both. So rather than trying to guess what you meant, Pyramid raises a configuration conflict error and refuses to start.

Resolving Conflicts

Obviously it's necessary to be able to resolve configuration conflicts. Sometimes these conflicts are done by mistake, so they're easy to resolve. You just change the code so that the conflict is no longer present. We can do that pretty easily:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

def hi_world(request):
    return Response('Hi world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.add_view(hi_world, route_name='home', request_param='use_hi')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

In the above code, we've gotten rid of the conflict. Now the `hello_world` view will be called by default when `/` is visited without a query string, but if `/` is visted when the URL contains a `use_hi` query string, the `hi_world` view will be executed instead. In other words, visiting `/` in the browser produces `Hello world!`, but visiting `/?use_hi=1` produces `Hi world!`.

There's an alternative way to resolve conflicts that doesn't change the semantics of the code as much. You can issue a `config.commit()` statement to flush pending configuration actions before issuing more. To see this in action, let's change our application back to the way it was before we added the `request_param` predicate to our second `add_view` statement:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

def hi_world(request): # added
    return Response('Hi world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.add_view(hi_world, route_name='home') # added
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

If we try to run this application as-is, we'll wind up with a configuration conflict error. We can actually sort of brute-force our way around that by adding a manual call to `commit` between the two `add_view` statements which conflict:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

def hi_world(request): # added
    return Response('Hi world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.commit() # added
    config.add_view(hi_world, route_name='home') # added
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

If we run this application, it will start up. And if we visit `/` in our browser, we'll see `Hi world!`. Why doesn't this application throw a configuration conflict error at the time it starts up? Because we flushed the pending configuration action impled by the first call to `add_view` by calling `config.commit()` explicitly. When we called the `add_view` the second time, the discriminator of the first call to `add_view` was no longer in the pending actions list to conflict with. The conflict was resolved because the pending actions list got flushed. Why do we see `Hi world!` in our browser instead of `Hello world!`? Because the call to `config.make_wsgi_app()` implies a second commit. The second commit caused the second `add_view` configuration callback to be called, and this callback overwrote the view configuration added by the first commit.

Calling `config.commit()` is a brute-force way to resolve configuration conflicts.

Including Configuration from Other Modules

Now that we have played around a bit with configuration that exists all in the same module, let's add some code to `app.py` that causes configuration that lives in another module to be *included*. We do that by adding a call to `config.include()` within `app.py`:

```
# app.py

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

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.include('another.moreconfiguration')  # added
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

We added the line `config.include('another.moreconfiguration')` above. If we try to run the application now, we'll receive a traceback:

```
Traceback (most recent call last):
  File "app.py", line 12, in <module>
    config.include('another')
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 744, in include
    c = self.maybe_dotted(callable)
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 844, in maybe_dotted
    return self.name_resolver.maybe_resolve(dotted)
  File "/home/chrism/projects/pyramid/pyramid/path.py", line 318, in maybe_resolve
    return self._resolve(dotted, package)
  File "/home/chrism/projects/pyramid/pyramid/path.py", line 325, in _resolve
    return self._zope_dottedname_style(dotted, package)
  File "/home/chrism/projects/pyramid/pyramid/path.py", line 368, in _zope_dottedname_style
    found = __import__(used)
ImportError: No module named another
```

That's exactly as we expected, because we attempted to *include* a module that doesn't yet exist. Let's add a module named `another.py` right next to our `app.py` module:

```
# another.py

from pyramid.response import Response

def goodbye(request):
    return Response('Goodbye world!')

def moreconfiguration(config):
    config.add_route('goodbye', '/goodbye')
    config.add_view(goodbye, route_name='goodbye')
```

Now what happens when we run the application via `python app.py`? It starts. And, like before, if we visit `/` in a browser, it still show `Hello world!`. But, unlike before, now if we visit `/goodbye` in a browser, it will show us `Goodbye world!`.

When we called `include('another.moreconfiguration')` within app.py, Pyramid interpreted this call as "please find the function named `moreconfiguration` in a module or package named `another` and call it with a configurator as the only argument". And that's indeed what happened: the `moreconfiguration` function within `another.py` was called; it accepted a configurator as its first argument and added a route and a view, which is why we can now visit `/goodbye` in the browser and get a response. It's the same effective outcome as if we had issued the `add_route` and `add_view` statements for the "goodbye" view from within `app.py`. An application can be created via configuration statements composed from multiple locations.

You might be asking yourself at this point "So what?! That's just a function call hidden under an API that resolves a module name to a function. I could just import the moreconfiguration function from `another` and call it directly with the configurator!" You're mostly right. However, `config.include()` does more than that. Please stick with me, we'll get to it.

The `includeme()` Convention

Now, let's change our `app.py` slightly. We'll change the `config.include()` line in `app.py` to include a slightly different name:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.include('another')  # <-- changed
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

And we'll edit `another.py`, changing the name of the `moreconfiguration` function to `includeme`:

```python
# another.py

from pyramid.response import Response

def goodbye(request):
    return Response('Goodbye world!')

def includeme(config): # <-- previously named moreconfiguration
    config.add_route('goodbye', '/goodbye')
    config.add_view(goodbye, route_name='goodbye')
```

When we run the application, it works exactly like our last iteration. You can visit `/` and `/goodbye` and get the exact same results. Why is this so? We didn't tell Pyramid the name of our new `includeme` function like we did before for `moreconfiguration` by saying `config.include('another.includeme')`, we just pointed it at the module in which `includeme` lived by saying `config.include('another')`. This is a Pyramid convenience shorthand: if you tell Pyramid to include a Python *module* or *package*, it will assume that you're telling it to include the `includeme` function from within that module/package. Effectively, `config.include('amodule')` always means `config.include('amodule.includeme')`.

Nested Includes

Something which is included can also do including. Let's add a file named `yetanother.py` next to app.py:

```python
# yetanother.py

from pyramid.response import Response

def whoa(request):
    return Response('Whoa')

def includeme(config):
    config.add_route('whoa', '/whoa')
    config.add_view(whoa, route_name='whoa')
```

And let's change our `another.py` file to include it:

```python
# another.py

from pyramid.response import Response

def goodbye(request):
    return Response('Goodbye world!')

def includeme(config): # <-- previously named moreconfiguration
    config.add_route('goodbye', '/goodbye')
    config.add_view(goodbye, route_name='goodbye')
    config.include('yetanother')
```

When we start up this application, we can visit `/`, `/goodbye` and `/whoa` and see responses on each. `app.py` includes `another.py` which includes `yetanother.py`. You can nest configuration includes within configuration includes ad infinitum. It's turtles all the way down.

Automatic Resolution via Includes

As we saw previously, it's relatively easy to manually resolve configuration conflicts that are produced by mistake. But sometimes configuration conflicts are not injected by mistake. Sometimes they're introduced on purpose in the desire to override one configuration statement with another. Pyramid anticipates this need in two ways: by offering automatic conflict resolution via `config.include()`, and the ability to manually commit configuration before a conflict occurs.

Let's change our `another.py` to contain a `hi_world` view function, and we'll change its `includeme` to add that view that should answer when / is visited:

```python
# another.py

from pyramid.response import Response

def goodbye(request):
    return Response('Goodbye world!')

```

```python
def hi_world(request): # added
    return Response('Hi world!')

def includeme(config):
    config.add_route('goodbye', '/goodbye')
    config.add_view(goodbye, route_name='goodbye')
    config.add_view(hi_world, route_name='home') # added
```

When we attempt to start the application, it will start without a conflict error. This is strange, because we have what appears to be the same configuration that caused a conflict error before when all of the same configuration statements were made in `app.py`. In particular, `hi_world` and `hello_world` are both being registered as the view that should be called when the `home` route is executed. When the application runs, when you visit `/` in your browser, you will see `Hello world!` (not `Hi world!`). The registration for the `hello_world` view in `app.py` "won" over the registration for the `hi_world` view in `another.py`.

Here's what's going on: Pyramid was able to automatically *resolve* a conflict for us. Configuration statements which generate the same discriminator will conflict. But if one of those configuration statements was performed as the result of being included "below" the other one, Pyramid will make an assumption: it's assuming that the thing doing the including (`app.py`) wants to *override* configuration statements done in the thing being included (`another.py`). In the above code configuration, even though the discriminator generated by `config.add_view(hello_world, route_name='home')` in `app.py` conflicts with the discriminator generated by `config.add_view(hi_world, route_name='home')` in `another.py`, Pyramid assumes that the former should override the latter, because `app.py` *includes* `another.py`.

Note that the same conflict resolution behavior does not occur if you simply import `another.includeme` from within app.py and call it, passing it a `config` object. This is why using `config.include` is different than just factoring your configuration into functions and arranging to call those functions at startup time directly. Using `config.include()` makes automatic conflict resolution work properly.

Custom Configuration Directives

A developer needn't satisfy himself with only the directives provided by Pyramid like `add_route` and `add_view`. He can add directives to the Configurator. This makes it easy for him to allow other developers to add application-specific configuration. For example, let's pretend you're creating an extensible application, and you'd like to allow developers to change the "site name" of your application (the site name is used in some web UI somewhere). Let's further pretend you'd like to do this by allowing people to call a `set_site_name` directive on the Configurator. This is a bit of a contrived example, because it would probably be a bit easier in this particular case just to use a deployment setting, but humor me for the purpose of this example. Let's change our app.py to look like this:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.include('another')
    config.set_site_name('foo')
    app = config.make_wsgi_app()
    print app.registry.site_name
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

And change our another.py to look like this:

```
# another.py

from pyramid.response import Response

def goodbye(request):
    return Response('Goodbye world!')

def hi_world(request):
    return Response('Hi world!')

def set_site_name(config, site_name):
    def callback():
        config.registry.site_name = site_name
    discriminator = ('set_site_name',)
    config.action(discriminator, callable=callback)

def includeme(config):
    config.add_route('goodbye', '/goodbye')
    config.add_view(goodbye, route_name='goodbye')
    config.add_view(hi_world, route_name='home')
    config.add_directive('set_site_name', set_site_name)
```

When this application runs, you'll see printed to the console foo. You'll notice in the app.py above, we call config.set_site_name. This is not a Pyramid built-in directive. It was added as the result of the call to config.add_directive in another.includeme. We added a function that uses the config.action method to register a discriminator and a callback for a *custom* directive. Let's change app.py again, adding a second call to set_site_name:

```
# app.py

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

def hello_world(request):
    return Response('Hello world!')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.include('another')
    config.set_site_name('foo')
    config.set_site_name('bar') # added this
    app = config.make_wsgi_app()
    print app.registry.site_name
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

When we try to start the application, we'll get this traceback:

```
Traceback (most recent call last):
  File "app.py", line 15, in <module>
    app = config.make_wsgi_app()
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 955, in make_wsgi_app
    self.commit()
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 629, in commit
    self.action_state.execute_actions(introspector=self.introspector)
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 1064, in execute_actions
    for action in resolveConflicts(self.actions):
  File "/home/chrism/projects/pyramid/pyramid/config/__init__.py", line 1192, in resolveConflicts
    raise ConfigurationConflictError(conflicts)
pyramid.exceptions.ConfigurationConflictError: Conflicting configuration actions
  For: ('site-name',)
    Line 13 of file app.py:
        config.set_site_name('foo')
    Line 14 of file app.py:
        config.set_site_name('bar')
```

We added a custom directive that made use of Pyramid's configuration conflict detection. When we tried to set the site name twice, Pyramid detected a conflict and told us. Just like built-in directives, Pyramid custom directives will also participate in automatic conflict resolution. Let's see that in action by moving our first call to `set_site_name` into another included function. As a result, our `app.py` will look like this:

```python
# app.py

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

def hello_world(request):
    return Response('Hello world!')

def moarconfig(config):
    config.set_site_name('foo')

if __name__ == '__main__':
    config = Configurator()
    config.add_route('home', '/')
    config.add_view(hello_world, route_name='home')
    config.include('another')
    config.include('.moarconfig')
    config.set_site_name('bar')
    app = config.make_wsgi_app()
    print app.registry.site_name
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

If we start this application up, we'll see `bar` printed to the console. No conflict will be raised, even though we have two calls to `set_site_name` being executed. This is because our custom directive is making use of automatic conflict resolution: Pyramid determines that the call to `set_site_name('bar')` should "win" because it's "closer to the top of the application" than the other call which sets it to "bar".

Why This Is Great

Now for some general descriptions of what makes the way all of this works great.

You'll note that a mere import of a module in our tiny application doesn't cause any sort of configuration state to be added, nor do any of our existing modules rely on some configuration having occurred before they can be imported. Application configuration is never added as the result of someone or something just happening to import a module. This seems like an obvious design choice, but it's not true of all web frameworks. Some web frameworks rely on a particular import ordering: you might not be able to successfully import your application code until some other module has been initialized via an import. Some web frameworks depend on configuration happening as a side effect of decorator execution: as a result, you might be *required* to import all of your application's modules for it to be configured in its entirety. Our application relies on neither: importing our code requires no prior import to have happened, and no configuration is done as the side effect of importing any of our code. This explicitness helps you build larger systems because you're never left guessing about the configuration state: you are entirely in charge at all times.

Most other web frameworks don't have a conflict detection system, and when they're fed two configuration statements that are logically conflicting, they'll choose one or the other silently, leaving you sometimes to wonder why you're not seeing the output you expect. Likewise, the execution ordering of configuration statements in most other web frameworks matters deeply; Pyramid doesn't make you care much about it.

A third party developer can override parts of an existing application's configuration as long as that application's original developer anticipates it minimally by factoring his configuration statements into a function that is *includable*. He doesn't necessarily have to anticipate *what* bits of his application might be overridden, just that *something* might be overridden. This is unlike other web frameworks, which, if they allow for application extensibility at all, indeed tend to force the original application developer to think hard about what might be overridden. Under other frameworks, an application developer that wants to provide application extensibility is usually required to write ad-hoc code that allows a user to override various parts of his application such as views, routes, subscribers, and templates. In Pyramid, he is not required to do this: everything is overridable, and he just refers anyone who wants to change the way it works to the Pyramid docs. The `config.include()` system even allows a third-party developer who wants to change an application to not think about the mechanics of overriding at all; he just adds statements before or after including the original developer's configuration statements, and he relies on automatic conflict resolution to work things out for him.

Configuration logic can be included from anywhere, and split across multiple packages and filesystem locations. There is no special set of Pyramid-y "application" directories containing configuration that must exist all in one place. Other web frameworks introduce packages or directories that are "more special than others" to offer similar features. To extend an application written using other web frameworks, you sometimes have to add to the set of them by changing a central directory structure.

The system is *meta-configurable*. You can extend the set of configuration directives offered to users by using `config.add_directive()`. This means that you can effectively extend Pyramid itself without needing to rewrite or redocument a solution from scratch: you just tell people the directive exists and tell them it works like every other Pyramid directive. You'll get all the goodness of conflict detection and resolution too.

All of the examples in this article use the "imperative" Pyramid configuration API, where a user calls methods on a Configurator object to perform configuration. For developer convenience, Pyramid also exposes a declarative configuration mechanism, usually by offering a function, class, and method decorator that is activated via a *scan*. Such decorators simply attach a callback to the object they're decorating, and during the scan process these callbacks are called: the callbacks just call methods on a configurator on the behalf of the user as if he had typed them himself. These decorators participate in Pyramid's configuration scheme exactly like imperative method calls.

For more information about `config.include()` and creating extensible applications, see Advanced Configuration and Extending An Existing Pyramid Application in the Pyramid narrative documenation. For more information about creating directives, see Extending Pyramid Configuration.

1.2.2 Django-Style "settings.py" Configuration

If you enjoy accessing global configuration via import statements ala Django's `settings.py`, you can do something similar in Pyramid.

- Create a `settings.py` file in your application's package (for example, if your application is named "myapp", put it in the filesystem directory named `myapp`; the one with an `__init__.py` in it.
- Add values to it at its top level.

For example:

```
# settings.py
import pytz

timezone = pytz('US/Eastern')
```

Then simply import the module into your application:

```
from myapp import settings

def myview(request):
    timezone = settings.timezone
    return Response(timezone.zone)
```

This is all you really need to do if you just want some global configuration values for your application.

However, more frequently, values in your `settings.py` file need to be conditionalized based on deployment settings. For example, the timezone above is different between development and deployment. In order to conditionalize the values in your `settings.py` you can use *other* values from the Pyramid `development.ini` or `production.ini`. To do so, your `settings.py` might instead do this:

```
import os

ini = os.environ['PYRAMID_SETTINGS']
config_file, section_name = ini.split('#', 1)

from paste.deploy.loadwsgi import appconfig
config = appconfig('config:%s' % config_file, section_name)

import pytz

timezone = pytz.timezone(config['timezone'])
```

The value of `config` in the above snippet will be a dictionary representing your application's `development.ini` configuration section. For example, for the above code to work, you'll need to add a `timezone` key/value pair to a section of your `development.ini`:

```
[app:myapp]
use = egg:MyApp
timezone = US/Eastern
```

If your `settings.py` is written like this, before starting Pyramid, ensure you have an OS environment value (akin to Django's `DJANGO_SETTINGS`) in this format:

```
export PYRAMID_SETTINGS=/place/to/development.ini#myapp
```

`/place/to/development.ini` is the full path to the ini file. `myapp` is the section name in the config file that represents your app (e.g. `[app:myapp]`). In the above example, your application will refuse to start without this environment variable being present.

For more information on configuration see the following sections of the Pyramid documentation:

- basic configuration
- advanced configuration
- configuration introspection
- extending configuration
- PasteDeploy configuration

1.3 Databases

1.3.1 SQLAlchemy

Basic Usage

You can get basic application template to use with SQLAlchemy by using *alchemy* scaffold. Check the narrative docs for more information.

Alternatively, you can try to follow wiki tutorial or blogr tutorial.

Using a Non-Global Session

It's sometimes advantageous to not use SQLAlchemy's thread-scoped sessions (such as when you need to use Pyramid in an asynchronous system). Thankfully, doing so is easy. You can store a session factory in the application's registry, and have the session factory called as a side effect of asking the request object for an attribute. The session object will then have a lifetime matching that of the request.

We are going to use `Configurator.add_request_method` to add SQLAlchemy session to request object and `Request.add_finished_callback` to close said session.

Note: `Configurator.add_request_method` has been available since Pyramid 1.4. You can use `Configurator.set_request_property` for Pyramid 1.3.

We'll assume you have an `.ini` file with `sqlalchemy.` settings that specify your database properly:

```python
# __init__.py

from pyramid.config import Configurator
from sqlalchemy import engine_from_config
from sqlalchemy.orm import sessionmaker

def db(request):
    maker = request.registry.dbmaker
    session = maker()

    def cleanup(request):
        if request.exception is not None:
            session.rollback()
        else:
            session.commit()
        session.close()
    request.add_finished_callback(cleanup)

    return session


def main(global_config, **settings):
    config = Configurator(settings=settings)
    engine = engine_from_config(settings, prefix='sqlalchemy.')
    config.registry.dbmaker = sessionmaker(bind=engine)
    config.add_request_method(db, reify=True)

    # .. rest of configuration ...
```

The SQLAlchemy session is now available in view code as `request.db` or `config.registry.dbmaker()`.

Importing all SQLAlchemy Models

If you've created a Pyramid project using a paster template, your SQLAlchemy models will, by default, reside in a single file. This is just by convention. If you'd rather have a directory for SQLAlchemy models rather than a file, you can of course create a Python package full of model modules, replacing the `models.py` file with a `models` directory which is a Python package (a directory with an `__init__.py` in it), as per Modifying Package Structure. However, due to the behavior of SQLAlchemy's "declarative" configuration mode, all modules which hold active SQLAlchemy models need to be imported before those models can successfully be used. So, if you use model classes with a declarative base, you need to figure out a way to get all your model modules imported to be able to use them in your application.

You might first create a `models` directory, replacing the `models.py` file, and within it a file named `models/__init__.py`. At that point, you can add a submodule named `models/mymodel.py` that holds a single `MyModel` model class. The `models/__init__.py` will define the declarative base class and the global `DBSession` object, which each model submodule (like `models/mymodel.py`) will need to import. Then all you need is to add imports of each submodule within `models/__init__.py`.

However, when you add `models` package submodule import statements to `models/__init__.py`, this will lead to a circular import dependency. The `models/__init__.py` module imports `mymodel` and `models/mymodel.py` imports the `models` package. When you next try to start your application, it will fail with an import error due to this circular dependency.

Pylons 1 solves this by creating a `models/meta.py` module, in which the DBSession and declarative base objects are created. The `models/__init__.py` file and each submodule of `models` imports `DBSession` and `declarative_base` from it. Whenever you create a `.py` file in the `models` package, you're expected to add an import for it to `models/__init__.py`. The main program imports the `models` package, which has the side effect of ensuring that all model classes have been imported. You can do this too, it works fine.

However, you can alternately use `config.scan()` for its side effects. Using `config.scan()` allows you to avoid a circdep between `models/__init__.py` and `models/themodel.py` without creating a special `models/meta.py`.

For example, if you do this in `myapp/models/__init__.py`:

```python
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker

DBSession = scoped_session(sessionmaker())
Base = declarative_base()

def initialize_sql(engine):
    DBSession.configure(bind=engine)
    Base.metadata.bind = engine
    Base.metadata.create_all(engine)
```

And this in `myapp/models/mymodel.py`:

```python
from myapp.models import Base
from sqlalchemy import Column
from sqlalchemy import Unicode
from sqlalchemy import Integer

class MyModel(Base):
    __tablename__ = 'models'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode(255), unique=True)
    value = Column(Integer)

    def __init__(self, name, value):
        self.name = name
        self.value = value
```

And this in `myapp/__init__.py`:

```python
from sqlalchemy import engine_from_config

from myapp.models import initialize_sql

def main(global_config, **settings):
    """ This function returns a Pyramid WSGI application.
    """
    config = Configurator(settings=settings)
    config.scan('myapp.models') # the "important" line
    engine = engine_from_config(settings, 'sqlalchemy.')
    initialize_sql(engine)
    # other statements here
    config.add_handler('main', '/{action}',
                       'myapp.handlers:MyHandler')
    return config.make_wsgi_app()
```

The important line above is `config.scan('myapp.models')`. `config.scan` has a side effect of performing a recursive import of the package name it is given. This side effect ensures that each file in `myapp.models` is imported without requiring that you import each "by hand" within `models/__init__.py`. It won't import any models that live outside the `myapp.models` package, however.

Writing Tests For Pyramid + SQLAlchemy

John Anderson's blog entry describes a strategy for writing tests for systems which integrate Pyramid and SQLAlchemy.

1.3.2 CouchDB and Pyramid

If you want to use CouchDB (via the couchdbkit package) in Pyramid, you can use the following pattern to make your CouchDB database available as a `request` attribute. This example uses the starter scaffold. (This follows the same pattern as the MongoDB and Pyramid example.)

First add configuration values to your `development.ini` file, including your CouchDB URI and a database name (the CouchDB database name, can be anything).

```
 [app:main]
 # ... other settings ...
 couchdb.uri = http://localhost:5984/
 couchdb.db = mydb
```

Then in your `__init__.py`, set things up such that the database is attached to each new request:

```
from pyramid.config import Configurator
from couchdbkit import *


def main(global_config, \**settings):
    """ This function returns a Pyramid WSGI application.
    """
    config = Configurator(settings=settings)
    config.registry.db = Server(uri=settings['couchdb.uri'])

    def add_couchdb(request):
        db = config.registry.db.get_or_create_db(settings['couchdb.db'])
        return db

    config.add_request_method(add_couchdb, 'db', reify=True)

    config.add_static_view('static', 'static', cache_max_age=3600)
    config.add_route('home', '/')
    config.scan()
    return config.make_wsgi_app()
```

Note: `Configurator.add_request_method` has been available since Pyramid 1.4. You can use `Configurator.set_request_property` for Pyramid 1.3.

At this point, in view code, you can use `request.db` as the CouchDB database connection. For example:

```python
from pyramid.view import view_config

@view_config(route_name='home', renderer='templates/mytemplate.pt')
def my_view(request):
    """ Get info for server
    """
    return {
        'project': 'pyramid_couchdb_example',
        'info': request.db.info()
    }
```

Add info to home template:

```html
 <p>${info}</p>
```

CouchDB Views

First let's create a view for our page data in CouchDB. We will use the ApplicationCreated event and make sure our view containing our page data. For more information on views in CouchDB see Introduction to CouchDB views. In __init__.py:

```python
from pyramid.events import subscriber, ApplicationCreated

@subscriber(ApplicationCreated)
def application_created_subscriber(event):
    registry = event.app.registry
    db = registry.db.get_or_create_db(registry.settings['couchdb.db'])

    pages_view_exists = db.doc_exist('lists/pages')
    if pages_view_exists == False:
        design_doc = {
            '_id': '_design/lists',
            'language': 'javascript',
            'views': {
                'pages': {
                    'map': '''
                        function(doc) {
                            if (doc.doc_type === 'Page') {
                                emit([doc.page, doc._id], null)
                            }
                        }
                    '''
                }
            }
        }
        db.save_doc(design_doc)
```

CouchDB Documents

Now we can let's add some data to a document for our home page in a CouchDB document in our view code if it doesn't exist:

```
import datetime

from couchdbkit import *

class Page(Document):
    author = StringProperty()
    page = StringProperty()
    content = StringProperty()
    date = DateTimeProperty()

@view_config(route_name='home', renderer='templates/mytemplate.pt')
def my_view(request):

    def get_data():
        return list(request.db.view('lists/pages', startkey=['home'], \
                endkey=['home', {}], include_docs=True))

    page_data = get_data()

    if not page_data:
        Page.set_db(request.db)
        home = Page(
            author='Wendall',
            content='Using CouchDB via couchdbkit!',
            page='home',
            date=datetime.datetime.utcnow()
        )
        # save page data
        home.save()
        page_data = get_data()

    doc = page_data[0].get('doc')

    return {
        'project': 'pyramid_couchdb_example',
        'info': request.db.info(),
        'author': doc.get('author'),
        'content': doc.get('content'),
        'date': doc.get('date')
    }
```

Then update your home template again to add your custom values:

```
 <p>
     ${author}<br />
```

```
    ${content}<br />
    ${date}<br />
</p>
```

1.3.3 MongoDB and Pyramid

Basics

If you want to use MongoDB (via PyMongo and perhaps GridFS) via Pyramid, you can use the following pattern to make your Mongo database available as a request attribute.

First add the MongoDB URI to your `development.ini` file. (Note: `user`, `password` and `port` are not required.)

```
[app:myapp]
# ... other settings ...
mongo_uri = mongodb://user:password@host:port/database
```

Then in your `__init__.py`, set things up such that the database is attached to each new request:

```
from pyramid.config import Configurator

try:
    # for python 2
    from urlparse import urlparse
except ImportError:
    # for python 3
    from urllib.parse import urlparse

from gridfs import GridFS
import pymongo


def main(global_config, **settings):
    """ This function returns a Pyramid WSGI application.
    """
    config = Configurator(settings=settings)
    config.add_static_view('static', 'static', cache_max_age=3600)

    db_url = urlparse(settings['mongo_uri'])
    config.registry.db = pymongo.Connection(
        host=db_url.hostname,
        port=db_url.port,
    )

    def add_db(request):
        db = config.registry.db[db_url.path[1:]]
        if db_url.username and db_url.password:
            db.authenticate(db_url.username, db_url.password)
        return db

    def add_fs(request):
        return GridFS(request.db)

    config.add_request_method(add_db, 'db', reify=True)
    config.add_request_method(add_fs, 'fs', reify=True)

    config.add_route('dashboard', '/')
    # other routes and more config...
    config.scan()
    return config.make_wsgi_app()
```

Note: `Configurator.add_request_method` has been available since Pyramid 1.4. You can use `Configurator.set_request_property` for Pyramid 1.3.

At this point, in view code, you can use `request.db` as the PyMongo database connection. For example:

```
@view_config(route_name='dashboard',
             renderer="myapp:templates/dashboard.pt")
def dashboard(request):
    vendors = request.db['vendors'].find()
    return {'vendors':vendors}
```

Scaffolds

Niall O'Higgins provides a pyramid_mongodb scaffold for Pyramid that provides an easy way to get started with Pyramid and MongoDB.

Video

Niall O'Higgins provides a presentation he gave at a Mongo conference in San Francisco at https://www.10gen.com/presentation/mongosf-2011/mongodb-with-python-pylons-pyramid

Other Information

- Pyramid traversal and MongoDB: http://kusut.web.id/2011/03/27/pyramid-traversal-and-mongodb/
- Pyramid, Aket and MongoDB: http://niallohiggins.com/2011/05/18/mongodb-python-pyramid-akhet/

1.4 Debugging

1.4.1 Using PDB to Debug Your Application

`pdb` is an interactive tool that comes with Python, which allows you to break your program at an arbitrary point, examine values, and step through code. It's often much more useful than print statements or logging statements to examine program state. You can place a `pdb.set_trace()` statement in your Pyramid application at a place where you'd like to examine program state. When you issue a request to the application, and that point in your code is reached, you will be dropped into the `pdb` debugging console within the terminal that you used to start your application.

There are lots of great resources that can help you learn PDB.

- Doug Hellmann's PyMOTW blog entry entitled "pdb - Interactive Debugger" at http://blog.doughellmann.com/2010/09/pymotw-pdb-interactive-debugger.html is the canonical text resource to learning PDB.
- The PyCon video presentation by Chris McDonough entitled "Introduction to PDB" at http://pyvideo.org/video/644/introduction-to-pdb is a good place to start learning PDB.

- The video at http://marakana.com/forums/python/python/423.html shows you how to start how to start to using pdb. The video describes using `pdb` in a command-line program.

1.4.2 Debugging Pyramid

This tutorial provides a brief introduction to using the python debugger (`pdb`) for debugging pyramid applications.

This scenario assume you've created a Pyramid project already. The scenario assumes you've created a Pyramid project named `buggy` using the `alchemy` scaffold.

Introducing PDB

- This single line of python is your new friend:

```
import pdb;  pdb.set_trace()
```

- As valid python, that can be inserted practically anywhere in a Python source file. When the python interpreter hits it - execution will be suspended providing you with interactive control from the parent TTY.

PDB Commands

- pdb exposes a number of standard interactive debugging commands, including:

```
Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pp       run      unt
a      c          continue  exit    l     q        s        until
alias  cl         d         h       list  quit     step     up
args   clear      debug     help    n     r        tbreak   w
b      commands   disable   ignore  next  restart  u        whatis
break  condition  down      j       p     return   unalias  where

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Debugging Our `buggy` App

- Back to our demo `buggy` application we generated from the `alchemy` scaffold, lets see if we can learn anything debugging it.
- The traversal documentation describes how pyramid first acquires a root object, and then descends the resource tree using the `__getitem__` for each respective resource.

Huh?

- Let's drop a pdb statement into our root factory object's `__getitem__` method and have a look. Edit the project's `models.py` and add the aforementioned `pdb` line in `MyModel.__getitem__`:

```
    def __getitem__(self, key):
        import pdb; pdb.set_trace()
        session = DBSession()
        # ...
```

- Restart the Pyramid application, and request a page. Note the request requires a path to hit our break-point:

```
http://localhost:6543/   <- misses the break-point, no traversal
http://localhost:6543/1  <- should find an object
http://localhost:6543/2  <- does not
```

- For a very simple case, attempt to insert a missing key by default. Set item to a valid new MyModel in `MyRoot.__getitem__` if a match isn't found in the database:

```
item = session.query(MyModel).get(id)
if item is None:
    item = MyModel(name='test %d'%id, value=str(id))  # naive insertion
```

- Move the break-point within the if clause to avoid the false positive hits:

```
if item is None:
    import pdb; pdb.set_trace()
    item = MyModel(name='test %d'%id, value=str(id))  # naive insertion
```

- Run again, note multiple request to the same id continue to create new MyModel instances. That's not right!
- Ah, of course, we forgot to add the new item to the session. Another line added to our `__getitem__` method:

```
if item is None:
    import pdb; pdb.set_trace()
    item = MyModel(name='test %d'%id, value=str(id))
    session.add(item)
```

- Restart and test. Observe the stack; debug again. Examine the item returning from MyModel:

```
(pdb) session.query(MyModel).get(id)
```

- Finally, we realize the item.id needs to be set as well before adding:

```
if item is None:
    item = MyModel(name='test %d'%id, value=str(id))
    item.id = id
    session.add(item)
```

- Many great resources can be found describing the details of using pdb. Try the interactive `help` (hit 'h') or a search engine near you.

Note: There is a well known bug in `PDB` in UNIX, when user cannot see what he is typing in terminal window after any interruption during `PDB` session (it can be caused by `CTRL-C` or when the server restarts automatically). This can be fixed by launching any of this commands in broken terminal: `reset`, `stty sane`. Also one can add one of this commands into `~/.pdbrc` file, so they will be launched before `PDB` session:

```
from subprocess import Popen
Popen(["stty", "sane"])
```

1.4.3 Debugging with PyDev

`pdb` is a great tool for debugging python scripts, but it has some limitations to its usefulness. For example, you must modify your code to insert breakpoints, and its command line interface can be somewhat obtuse.

Many developers use custom text editors that that allow them to add wrappers to the basic command line environment, with support for git and other development tools. In many cases, however, debugging support basically ends up being simply a wrapper around basic `pdb` functionality.

PyDev is an Eclipse plugin for the Python language, providing an integrated development environment that includes a built in python interpreter, Git support, integration with task management, and other useful development functionality.

The PyDev debugger allows you to execute code without modifying the source to set breakpoints, and has a gui interface that allows you to inspect and modify internal state.

Lars Vogella has provided a clear tutorial on setting up pydev and getting started with the PyDev debugger. Full documentation on using the PyDev debugger may be found here. You can also debug programs not running under Eclipse using the Remote Debugging feature.

PyDev allows you to configure the system to use any python intepreter you have installed on your machine, and with proper configuration you can support both 2.x and 3.x syntax.

Configuring PyDev for a virtualenv

Most of the time you want to be running your code in a virtualenv in order to be sure that your code is isolated and all the right versions of your package dependencies are available. You can `pip install virtualenv` if you like, but I recommend virtualenvwrapper which eliminates much of the busywork of setting up virtualenvs.

PyDev will look through all the libraries on your `PYTHONPATH` to resolve all your external references, such as imports, etc. So you will want the virtualenv libraries on your `PYTHONPATH` to avoid unnecessary name-resolution problems.

To use PyDev with virtualenv takes some additional configuration that isn't covered in the above tutorial. Basically, you just need to make sure your virtualenv libraries are in the `PYTHONPATH`.

Note: If you have never configured a python interpreter for your workspace, you will not be able to create a project without doing so. You should follow the steps below to configure python, but you should NOT include any virtualenv libraries for it. Then you will be able to create projects using this primary python interpreter. After you create your project, you should then follow the steps below to configure a new interpreter specifically for your project which does include the virtualenv libraries. This way, each project can be related to a specific virtualenv without confusion.

First, open the project properties by right clicking over the project name and selecting *Properties*.

In the Properties dialog, select *PyDev - Interpreter/Grammar*, and make sure that the project type *Python* is selected. Click on the "Click here to configure an interpreter not listed" link. The *Preferences* dialog will come up with *Python Interpreters* page, and your current interpreter selected. Click on the *New...* button.

Enter a name (e.g. `pytest_python`) and browse to your virtualenv bin directory (e.g. `~/.virtual_envs/pytest/bin/python`) to select the python interpreter in that location, then select *OK*.

A dialog will then appear asking you to choose the libraries that should be on the `PYTHONPATH`. Most of the necessary libraries should be automatically selected. Hit *OK*, and your virtualenv python is now configured.

Note: On the Mac, the system libraries are not selected. Select them all.

You will finally be back on the dialog for configuring your project python interpreter/grammar. Choose the interpreter you just configured and click *OK*. You may also choose the grammar level (2.7, 3.0, etc.) at this time.

At this point, formerly unresolved references to libraries installed in your virtualenv should no longer be called out as errors. (You will have to close and reopen any python modules before the new interpreter will take effect.)

Remember also when using the PyDev console, to choose the interpreter associated with the project so that references in the console will be properly resolved.

Running/Debugging Pyramid under Pydev

(Thanks to Michael Wilson for much of this - see Setting up Eclipse (PyDev) for Pyramid)

Note: This section assumes you have created a virtualenv with Pyramid installed, and have configured your PyDev as above for this virtualenv. We further assume you are using `virtualenvwrapper` (see above) so that `$WORKON_HOME` is the location of your `.virtualenvs` directory and `proj_venv` is the name of your virtualenv. `$WORKSPACE` is the name of the PyDev workspace containing your project

To create a working example, copy the pyramid tutorial step03 code into $WORKSPACE/tutorial.

After copying the code, cd to `$WORKSPACE/tutorial` and run `python setup.py develop`

You should now be ready to setup PyDev to run the tutorial step03 code.

We will set up PyDev to run pserve as part of a run or debug configuration.

First, copy `pserve.py` from your virtualenv to a location outside of your project library path:

```
cp $WORKON_HOME/proj_venv/bin/pserve.py $WORKSPACE
```

Note: IMPORTANT: Do not put this in your project library path!

Now we need to have PyDev run this by default. To create a new run configuration, right click on the project and select *Run As -> Run Configurations....* Select *Python Run* as your configuration type, and click on the new configuration icon. Add your project name (or browse to it), in this case "tutorial".

Add these values to the *Main* tab:

```
Project: RunPyramid
Main Module: ${workspace_loc}/pserve.py
```

Add these values to the *Arguments* tab:

```
Program arguments: ${workspace_loc:tutorial/development.ini} --reload
```

Note: Do not add `--reload` if you are trying to debug with Eclipse. It has been reported that this causes problems.

We recommend you create a separate debug configuration without the `--reload`, and instead of checking "Run" in the "Display in favorites menu", check "Debug".

On the *Common* tab:

```
Uncheck "Launch in background"
In the box labeled "Display in favorites menu", check "Run"
```

Hit *Run* (*Debug*) to run (debug) your configuration immediately, or *Apply* to create the configuration without running it.

You can now run your application at any time by selecting the *Run/Play* button and selecting the *RunPyramid* command. Similarly, you can debug your application by selecting the *Debug* button and selecting the *DebugPyramid* command (or whatever you called it!).

The console should show that the server has started. To verify, open your browser to 127.0.0.1:6547. You should see the hello world text.

Note that when debugging, breakpoints can be set as with ordinary code, but they will only be hit when the view containing the breakpoint is served.

1.5 Deployment

1.5.1 Deploying Your Pyramid Application

So you've written a sweet application and you want to deploy it outside of your local machine. We're not going to cover caching here, but suffice it to say that there are a lot of things to consider when optimizing your pyramid application.

At a high level, you need to expose a server on ports 80 (HTTP) and 443 (HTTPS). Underneath this layer, however, is a plethora of different configurations that can be used to get a request from a client, into your application, and return the response.

```
Client <---> WSGI Server <---> Your Application
```

Due to the beauty of standards, many different configurations can be used to generate this basic setup, injecting caching layers, load balancers, etc into the basic workflow.

Disclaimer

It's important to note that the setups discussed here are meant to give some direction to newer users. Deployment is *almost always* highly dependent on the application's specific purposes. These setups have been used for many different projects in production with much success, but never verbatim.

What is WSGI?

WSGI is a Python standard dictating the interface between a server and an application. The entry point to your pyramid application is an object implementing the WSGI interface. Thus, your application can be served by any server supporting WSGI.

There are many different servers implementing the WSGI standard in existance. A short list includes:

- `waitress`
- `paste.httpserver`
- `CherryPy`
- `uwsgi`
- `gevent`
- `mod_wsgi`

For more information on WSGI, see the WSGI home

1.5.2 Nginx + pserve + supervisord

This setup can be accomplished simply and is capable of serving a large amount of traffic. The advantage in deployment is that by using `pserve`, it is not unlike the basic development environment you're probably using on your local machine.

Nginx is a highly optimized HTTP server, very capable of serving static content as well as acting as a proxy between other applications and the outside world. As a proxy, it also has good support for basic load balancing between multiple instances of an application.

```
Client <---> Nginx [0.0.0.0:80] <---> (static files)
              /|\
               |-------> WSGI App [localhost:5000]
               `-------> WSGI App [localhost:5001]
```

Our target setup is going to be an Nginx server listening on port 80 and load-balancing between 2 pserve processes. It will also serve the static files from our project's directory.

Let's assume a basic project setup:

```
/home/example/myapp
|
|-- env (your virtualenv)
|
|-- myapp
|    |
|    |-- __init__.py (defining your main entry point)
|    |
|    `-- static (your static files)
|
|-- production.ini
|
`-- supervisord.conf (optional)
```

Step 1: Configuring Nginx

Nginx needs to be configured as a proxy for your application. An example configuration is shown here:

```
# nginx.conf

user www-data;
worker_processes 4;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
    # multi_accept on;
}

http {

    ##
    # Basic Settings
    ##

    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;
    # server_tokens off;

    # server_names_hash_bucket_size 64;
    # server_name_in_redirect off;

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    ##
    # Logging Settings
    ##

    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    ##
    # Gzip Settings
    ##

    gzip on;
    gzip_disable "msie6";

    ##
    # Virtual Host Configs
    ##

    server {
        server_name _;
        return 444;
    }

    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

```
# myapp.conf

upstream myapp-site {
    server 127.0.0.1:5000;
    server 127.0.0.1:5001;
}

server {

    # optional ssl configuration

    listen 443 ssl;
    ssl_certificate /path/to/ssl/pem_file;
    ssl_certificate_key /path/to/ssl/certificate_key;

    # end of optional ssl configuration

    server_name  example.com;

    access_log  /home/example/env/access.log;

    location / {
        proxy_set_header        Host $http_host;
        proxy_set_header        X-Real-IP $remote_addr;
        proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header        X-Forwarded-Proto $scheme;

        client_max_body_size    10m;
        client_body_buffer_size 128k;
        proxy_connect_timeout   60s;
        proxy_send_timeout      90s;
        proxy_read_timeout      90s;
        proxy_buffering         off;
        proxy_temp_file_write_size 64k;
        proxy_pass http://myapp-site;
        proxy_redirect          off;
    }
}
```

Note: myapp.conf is actually included into the `http {}` section of the main nginx.conf file.

The optional `listen` directive, as well as the 2 following lines, are the only configuration changes required to enable SSL from the Client to Nginx. You will need to have already created your SSL certificate and key for this to work. More details on this process can be found in the OpenSSL howto. You will also need to update the paths that are shown to match the actual path to your SSL certificates.

The `upstream` directive sets up a round-robin load-balancer between two processes. The proxy is then configured to pass requests through the balancer with the `proxy_pass` directive. It's important to investigate the implications of many of the other settings as they are likely application-specific.

The `header` directives inform our application of the exact deployment setup. They will help the WSGI server configure our environment's `SCRIPT_NAME`, `HTTP_HOST`, and the actual IP address of the client.

Step 2: Starting pserve

Warning: Be sure to create a `production.ini` file to use for deployment that has debugging turned off and removing the pyramid_debugtoolbar.

This configuration uses PasteDeploy's `PrefixMiddleware` to automatically convert the `X-Forwarded-Proto` into the correct HTTP scheme in the WSGI environment. This is important so that the URLs generated by the application can distinguish between different domains, HTTP vs. HTTPS, and with some extra configuration to the `paste_prefix` filter it can even handle hosting the application under a different URL than `/`.

```
#---------- App Configuration ----------
[app:myapp]
use = egg:myapp#main
pyramid.reload_templates = false
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.default_locale_name = en

#---------- Pipeline Configuration ----------
[filter:paste_prefix]
use = egg:PasteDeploy#prefix

[pipeline:main]
```

```
pipeline =
    paste_prefix
    # a good spot for some logging middleware!
    myapp

#---------- Server Configuration ----------
[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = %(http_port)s

#---------- Logging Configuration ----------
# ...
```

Running the pserve processes:

```
pserve --daemon --pid-file=pserve_5000.pid production.ini http_port=5000
pserve --daemon --pid-file=pserve_5001.pid production.ini http_port=5001
```

Step 3: Serving Static Files with Nginx (Optional)

Assuming your static files are in a subdirectory of your pyramid application, they can be easily served using nginx's highly optimized web server. This will greatly improve performance because requests for this content will not need to be proxied to your WSGI application and can be served directly.

Warning: This is only a good idea if your static content is intended to be public. It will not respect any view permissions you've placed on this directory.

```
...

location / {
    # all of your proxy configuration
}

location /static {
    root                    /home/example/myapp/myapp;
    expires                 30d;
    add_header              Cache-Control public;
    access_log              off;
}
```

It's somewhat odd that the `root` doesn't point to the `static` directory, but it works because Nginx will append the actual URL to the specified path.

Step 4: Managing Your pserve Processes with Supervisord (Optional)

Turning on all of your `pserve` processes manually and daemonizing them works for the simplest setups, but for a really robust server, you're going to want to automate the startup and shutdown of those processes, as well as have some way of managing failures.

Enter `supervisord`:

```
pip install supervisor
```

This is a great program that will manage arbitrary processes, restarting them when they fail, providing hooks for sending emails, etc when things change, and even exposing an XML-RPC interface for determining the status of your system.

Below is an example configuration that starts up two instances of the pserve process, automatically filling in the `http_port` based on the `process_num`, thus 5000 and 5001.

This is just a stripped down version of `supervisord.conf`, read the docs for a full breakdown of all of the great options provided.

```
[unix_http_server]
file=%(here)s/env/supervisor.sock

[supervisord]
pidfile=%(here)s/env/supervisord.pid
logfile=%(here)s/env/supervisord.log
logfile_maxbytes=50MB
logfile_backups=10
loglevel=info
nodaemon=false
minfds=1024
minprocs=200

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix://%(here)s/env/supervisor.sock

[program:myapp]
autorestart=true
command=%(here)s/env/bin/pserve %(here)s/production.ini http_port=50%(process_num)02d
process_name=%(program_name)s-%(process_num)01d
numprocs=2
numprocs_start=0
redirect_stderr=true
stdout_logfile=%(here)s/env/%(program_name)s-%(process_num)01d.log
```

1.5.3 Apache + mod_wsgi

Pyramid mod_wsgi tutorial

1.5.4 gevent

gevent + pyramid_socketio

Alexandre Bourget explains how he uses gevent + socketio to add functionality +to a Pyramid application at http://blog.abourget.net/2011/3/17/new-and-hot-part-4-pyramid-socket-io-gevent/

gevent + long polling

http://michael.merickel.org/2011/6/21/tictactoe-and-long-polling-with-pyramid/

https://github.com/mmerickel/tictactoe

For more information on gevent see the gevent home page

1.5.5 Heroku

Heroku recently added support for a process model which allows deployment of Pyramid applications.

This recipe assumes that you have a Pyramid application setup using a Paste INI file, inside a package called `myapp`. This type of structure is found in the `pyramid_starter` scaffold, and other Paste scaffolds (previously called project templates). It can be easily modified to work with other Python web applications as well by changing the command to run the application as appropriate.

Step 0: Install Heroku

Install the heroku gem per their instructions.

Step 1: Add files needed for Heroku

You will need to add the following files with the contents as shown to the root of your project directory (the directory containing the `setup.py`).

`requirements.txt`

You can autogenerate this file with the following command.

```
$ pip freeze > requirements.txt
```

In your `requirements.txt` file, you will probably have a line with your project's name in it. It might look like either of the following two lines depending on how you setup your project. If either of these lines exist, delete them.

```
project-name=0.0

# or

-e git+git@xxxx:<git username>/xxxxx.git....#egg=project-name
```

Note: You can only use packages that can be installed with pip (e.g., those on PyPI, those in a git repo, using a git+git:// url, etc.). If you have any that you need to install in some special way, you will have to do that in your `run` file (see below). Also note that this will be done for every instance startup, so it needs to complete quickly to avoid being killed by Heroku (there's a 60-second instance startup timeout). Never include editable references when deploying to Heroku.

`Procfile`

Generate `Procfile` with the following command.

```
$ echo "web: ./run" > Procfile
```

`run`

Create `run` with the following command.

```
#!/bin/bash
set -e
python setup.py develop
python runapp.py
```

Note: Make sure to `chmod +x run` before continuing. The `develop` step is necessary because the current package must be installed before Paste can load it from the INI file.

runapp.py

If using a version greater than or equal to 1.3 (e.g. >= 1.3), use the following for `runapp.py`.

```
import os

from paste.deploy import loadapp
from waitress import serve

if __name__ == "__main__":
    port = int(os.environ.get("PORT", 5000))
    app = loadapp('config:production.ini', relative_to='.')

    serve(app, host='0.0.0.0', port=port)
```

For versions of Pyramid prior to 1.3 (e.g. < 1.3), use the following for `runapp.py`.

```
import os

from paste.deploy import loadapp
from paste import httpserver

if __name__ == "__main__":
    port = int(os.environ.get("PORT", 5000))
    app = loadapp('config:production.ini', relative_to='.')

    httpserver.serve(app, host='0.0.0.0', port=port)
```

Note: We assume the INI file to use is named `production.ini`, so change the content of `runapp.py` as necessary. The server section of the INI will be ignored as the server needs to listen on the port supplied in the OS environment.

Step 2: Setup git repo and Heroku app

Navigate to your project directory (directory with `setup.py`) if not already there. If your project is already under git version control, skip to the “Initialize the Heroku stack” section.

Inside your project’s directory, if this project is not tracked under git, it is recommended yet optional to create a good `.gitignore` file. You can get the recommended python one by running the following command.

```
$ wget -O .gitignore https://raw.github.com/github/gitignore/master/Python.gitignore
```

Once that is done, run the following command.

```
$ git init
$ git add .
$ git commit -m "initial commit"
```

Step 3: Initialize the Heroku stack

```
$ heroku create --stack cedar
```

Step 4: Deploy

To deploy a new version, push it to Heroku.

```
$ git push heroku master
```

Make sure to start one worker.

```
$ heroku scale web=1
```

Check to see if your app is running.

```
$ heroku ps
```

Take a look at the logs to debug any errors if necessary.

```
$ heroku logs -t
```

Tips and Tricks

The CherryPy WSGI server is fast, efficient, and multi-threaded to easily handle many requests at once. If you want to use it you can add `cherrpy` and `pastescript` to your `setup.py:requires` section (be sure to re-run `pip freeze` to update the requirements.txt file as explained above) and setup your `runapp.py` to look like the following.

```
import os

from paste.deploy import loadapp
from paste.script.cherrypy_server import cpwsgi_server

if __name__ == "__main__":
    port = int(os.environ.get("PORT", 5000))
    wsgi_app = loadapp('config:production.ini', relative_to='.')
    cpwsgi_server(wsgi_app, host='0.0.0.0', port=port,
                  numthreads=10, request_queue_size=200)
```

Heroku add-ons generally communicate their settings via OS environment variables. These can be easily incorporated into your applications settings as show in the following example.

```
# In your pyramid apps main init
import os

from pyramid.config import Configurator
from myproject.resources import Root

def main(global_config, **settings):
    """ This function returns a Pyramid WSGI application.
    """

    # Look at the environment to get the memcache server settings
    memcache_server = os.environ.get('MEMCACHE_SERVERS')

    settings['beaker.cache.url'] = memcache_server
    config = Configurator(root_factory=Root, settings=settings)
    config.add_view('myproject.views.my_view',
                    context='myproject.resources.Root',
                    renderer='myproject:templates/mytemplate.pt')
    config.add_static_view('static', 'myproject:static')
    return config.make_wsgi_app()
```

1.5.6 DotCloud

DotCloud offers support for all WSGI frameworks. Below is a quickstart guide for Pyramid apps. You can also read the DotCloud Python documentation for a complete overview.

Step 0: Install DotCloud

Install DotCloud's CLI by running:

```
$ pip install dotcloud
```

Step 1: Add files needed for DotCloud

DotCloud expects Python applications to have a few files in the root of the project. First, you need a pip `requirements.txt` file to instruct DotCloud which Python library dependencies to install for your app. Secondly you need a `dotcloud.yaml` file which informs DotCloud that your application has (at a minimum) a Python service. You may also want additional services such as a MongoDB database or PostgreSQL database and so on - these things are all specified in YAML.

Finally, you will need a file named `wsgi.py` which is what the DotCloud uWSGI server is configured to look for. This wsgi.py script needs to create a WSGI callable for your Pyramid app which must be present in a global named "application".

You'll need to add a requirements.txt, dotcloud.yml, and wsgi.py file to the root directory of your app. Here are some samples for a basic Pyramid app:

`requirements.txt`:

```
cherrypy
Pyramid==1.3
# Add any other dependencies that should be installed as well
```

`dotcloud.yml`:

```
www:
    type: python
db:
    type: postgresql
```

Learn more about the DotCloud buildfile.

`wsgi.py`:

```
# Your WSGI callable should be named “application”, be located in a
# "wsgi.py" file, itself located at the top directory of the service.
#
# For example, to load the app from your "production.ini" file in the same
# directory:
import os.path
from pyramid.scripts.pserve import cherrypy_server_runner
from pyramid.paster import get_app

application = get_app(os.path.join(os.path.dirname(__file__), 'production.ini'))

if __name__ == "__main__":
    cherrypy_server_runner(application, host="0.0.0.0")
```

Step 2: Configure your database

If you specified a database service in your dotcloud.yml, the connection info will be made available to your service in a JSON file at /home/dotcloud/environment.json. For example, the following code would read the environment.json file and add the PostgreSQL URL to the settings of your pyramid app:

```
import json

# if dotcloud, read PostgreSQL URL from environment.json
db_uri = settings['postgresql.url']
DOTCLOUD_ENV_FILE = "/home/dotcloud/environment.json"
if os.path.exists(DOTCLOUD_ENV_FILE):
    with open(DOTCLOUD_ENV_FILE) as f:
        env = json.load(f)
        db_uri = env["DOTCLOUD_DATA_POSTGRESQL_URL"]
```

Learn more about the DotCloud environment.json.

Step 3: Deploy your app

Now you can deploy your app. Remember to commit your changes if you’re using Mercurial or Git, then run these commands in the top directory of your app:

```
$ dotcloud create your_app_name
$ dotcloud push your_app_name
```

At the end of the push, you’ll see the URL(s) for your new app. Have fun!

1.5.7 OpenShift Express Cloud

This blog entry describes deploying a Pyramid application to RedHat’s OpenShift Express Cloud platform.

Luke Macken’s OpenShift Quickstarter also provides an easy way to get started using OpenShift.

1.5.8 Pyramid on Google’s App Engine (using buildout)

This is but one way to develop applications to run on Google’s App Engine. This one uses buildout . For a different approach, you may want to look at *Pyramid on Google’s App Engine (using appengine-monkey)*.

Install the pyramid_appengine scaffold

Let's take it step by step.

You can get pyramid_appengine from pypi via pip just as you typically would any other python package, however to reduce the chances of the system installed python packages intefering with tools you use for your own development you should install it in a local virtual environment

Creating a virtual environment

Update distribute

```
$ sudo pip install --upgrade distribute
```

Install virtualenv

```
$ sudo pip install virtualenv
```

create a virtual environment

```
$ virtualenv -p /usr/bin/python2.7 --no-site-packages --distribute myenv
```

install pyramid_appengine into your virtual environment

```
$ myenv/bin/pip install pyramid_appengine
```

Once successfully installed a new project template is available to use named "appengine_starter".

To get a list of all available templates.

```
$ myenv/bin/pcreate -l
```

Create the skeleton for your project

You create your project skeleton using the "appengine_starter" project scaffold just as you would using any other project scaffold.

```
$ myenv/bin/pcreate -t appengine_starter newproject
```

Once successfully ran, you will have a new buildout directory for your project. The app engine application source is located at newproject/src/newproject.

This buildout directory can be added to version control if you like, using any of the available version control tools available to you.

Bootstrap the buildout

Before you do anything with a new buildout directory you need to bootstrap it, which installs buildout locally and everything necessary to manage the project dependencies.

As with all buildouts, it can be bootstrapped running the following commands.

```
~/ $ cd newproject
~/newproject $ ../bin/python2.7 bootstrap.py
```

You typically only need to do this once to generate your buildout command. See the buildout documentation for more information.

Run buildout

As with all buildouts, after it has been bootstrapped, a "bin" directory is created with a new buildout command. This command is run to install things based on the newproject/buildout.cfg which you can edit to suit your needs.

```
~/newproject $ ./bin/buildout
```

In the case of this particular buildout, when run, it will take care of several things that you need to do....

1. install the app engine SDK in parts/google_appengine more info
2. Place tools from the appengine SDK in the buildout's "bin" directory.
3. Download/install the dependencies for your project including pyramid and all it's dependencies not already provided by the app engine SDK. more info
4. A directory structure appropriate for deploying to app engine at newproject/parts/newproject. more info
5. Download/Install tools to support unit testing including pytest, and coverage.

Run your tests

Your project is configured to run all tests found in files that begin with "test_"(example: newproject/src/newproject/newproject/test_views.py).

```
~/newproject/ $ cd src/newproject
~/newproject/src/newproject/ $ ../../bin/python setup.py test
```

Your project incorporates the unit testing tools provided by the app engine SDK to setUp and tearDown the app engine environment for each of your tests. In addition to that, running the unit tests will keep your projects index.yaml up to date. As a result, maintaining a thorough test suite will be your best chance at insuring that your application is ready for deployment.

You can adjust how the app engine api's are initialized for your tests by editing newproject/src/newproject/newproject/conftest.py.

Run your application locally

You can run your application using the app engine SDK's Development Server

```
~/newproject/ $ ./bin/devappserver parts/newproject
```

Point your browser at http://localhost:8080 to see it working.

Deploy to App Engine

Note: Before you can upload any appengine application you must create an application ID for it.

To upload your application to app engine, run the following command. For more information see App Engine Documentation for appcfg

```
~/newproject/ $ ./bin/appcfg update parts/newproject -A newproject -V dev
```

Point your browser at http://dev.newproject.appspot.com to see it working.

The above command will most likely not work for you, it is just an example. the "-A" switch indicates an Application ID to deploy to and overrides the setting in the app.yaml, use the Application ID you created when you registered the application instead. The "-V" switch specifies the version and overrides the setting in your app.yaml.

You can set which version of your application handles requests by default in the admin console. However you can also specify a version of your application to hit in the URL like so...

```
http://<app-version>.<application-id>.appspot.com
```

This can come in pretty handy in a variety of scenarios that become obvious once you start managing the development of your application while supporting a current release.

1.5.9 Pyramid on Google's App Engine (using appengine-monkey)

This is but one way to develop applications to run on Google's App Engine. This one uses *appengine-monkey*. For a different approach, you may want to look at *Pyramid on Google's App Engine (using buildout)*.

It is possible to run a Pyramid application on Google's App Engine. Content from this tutorial was contributed by YoungKing, based on the "appengine-monkey" tutorial for Pylons. This tutorial is written in terms of using the command line on a UNIX system; it should be possible to perform similar actions on a Windows system.

1. Download Google's App Engine SDK and install it on your system.
2. Use Subversion to check out the source code for `appengine-monkey`.

```
    $ svn co http://appengine-monkey.googlecode.com/svn/trunk/ \
        appengine-monkey
```

3. Use `appengine_homedir.py` script in `appengine-monkey` to create a virtualenv for your application.

```
    $ export GAE_PATH=/usr/local/google_appengine
    $ python2.5 /path/to/appengine-monkey/appengine-homedir.py --gae \
      $GAE_PATH pyramidapp
```

   Note that `$GAE_PATH` should be the path where you have unpacked the App Engine SDK. (On Mac OS X at least, `/usr/local/google_appengine` is indeed where the installer puts it).

   This will set up an environment in `pyramidapp/`, with some tools installed in `pyramidapp/bin`. There will also be a directory `pyramidapp/app/` which is the directory you will upload to appengine.

4. Install Pyramid into the virtualenv

```
    $ cd pyramidapp/
    $ bin/easy_install pyramid
```

   This will install Pyramid in the environment.

5. Create your application

   We'll use the standard way to create a Pyramid application, but we'll have to move some files around when we are done. The below commands assume your current working directory is the `pyramidapp` virtualenv directory you created in the third step above:

```
    $ cd app
    $ rm -rf pyramidapp
    $ bin/pcreate -s starter pyramidapp
    $ mv pyramidapp aside
    $ mv aside/pyramidapp .
    $ rm -rf aside
```

6. Edit `config.py`

   Edit the `APP_NAME` and `APP_ARGS` settings within `config.py`. The `APP_NAME` must be `pyramidapp:main`, and the APP_ARGS must be `({},)`. Any other settings in `config.py` should remain the same:

```
APP_NAME = 'pyramidapp:main'
APP_ARGS = ({},)
```

7. Edit `runner.py`

   To prevent errors for `import site`, add this code stanza before `import site` in app/runner.py:

```
import sys
sys.path = [path for path in sys.path if 'site-packages' not in path]
import site
```

   You will also need to comment out the line that starts with `assert sys.path` in the file:

```
# comment the sys.path assertion out
# assert sys.path[:len(cur_sys_path)] == cur_sys_path, (
#   "addsitedir() caused entries to be prepended to sys.path")
```

   For GAE development environment 1.3.0 or better, you will also need the following somewhere near the top of the `runner.py` file to fix a compatibility issue with `appengine-monkey`:

```
import os
os.mkdir = None
```

8. Run the application. `dev_appserver.py` is typically installed by the SDK in the global path but you need to be sure to run it with Python 2.5 (or whatever version of Python your GAE SDK expects).

```
$ cd ../..
$ python2.5 /usr/local/bin/dev_appserver.py pyramidapp/app/
```

   Startup success looks something like this:

```
[chrism@vitaminf pyramid_gae]$ python2.5 \
              /usr/local/bin/dev_appserver.py \
              pyramidapp/app/
INFO     2009-05-03 22:23:13,887 appengine_rpc.py:157] # ... more...
Running application pyramidapp on port 8080: http://localhost:8080
```

   You may need to run "Make Symlinks" from the Google App Engine Launcher GUI application if your system doesn't already have the `dev_appserver.py` script sitting around somewhere.

9. Hack on your pyramid application, using a normal run, debug, restart process. For tips on how to use the `pdb` module within Google App Engine, see this blog post. In particular, you can create a function like so and call it to drop your console into a pdb trace:

```
def set_trace():
    import pdb, sys
    debugger = pdb.Pdb(stdin=sys.__stdin__,
        stdout=sys.__stdout__)
    debugger.set_trace(sys._getframe().f_back)
```

10. Sign up for a GAE account and create an application. You'll need a mobile phone to accept an SMS in order to receive authorization.

11. Edit the application's ID in `app.yaml` to match the application name you created during GAE account setup.

```
application: mycoolpyramidapp
```

12. Upload the application

```
$ python2.5 /usr/local/bin/appcfg.py update pyramidapp/app
```

You almost certainly won't hit the 3000-file GAE file number limit when invoking this command. If you do, however, it will look like so:

```
HTTPError: HTTP Error 400: Bad Request
Rolling back the update.
Error 400: --- begin server output ---
Max number of files and blobs is 3000.
--- end server output ---
```

If you do experience this error, you will be able to get around this by zipping libraries. You can use `pip` to create zipfiles from packages. See *Zipping Files Via Pip* for more information about this.

A successful upload looks like so:

```
[chrism@vitaminf pyramidapp]$ python2.5 /usr/local/bin/appcfg.py \
                              update ../pyramidapp/app/
Scanning files on local disk.
Scanned 500 files.
# ... more output ...
Will check again in 16 seconds.
Checking if new version is ready to serve.
Closing update: new version is ready to start serving.
Uploading index definitions.
```

13. Visit `http://<yourapp>.appspot.com` in a browser.

Zipping Files Via Pip

If you hit the Google App Engine 3000-file limit, you may need to create zipfile archives out of some distributions installed in your application's virtualenv.

First, see which packages are available for zipping:

```
$ bin/pip zip -l
```

This shows your zipped packages (by default, none) and your unzipped packages. You can zip a package like so:

```
$ bin/pip zip pytz-2009g-py2.5.egg
```

Note that it requires the whole egg file name. For a Pyramid app, the following packages are good candidates to be zipped.

- Chameleon
- zope.i18n

Once the zipping procedure is finished you can try uploading again.

1.5.10 Windows

There are four possible deployment options for Windows:

1. Run as a Windows service with a Python based web server like CherryPy or Twisted
2. Run as a Windows service behind another web server (either IIS or Apache) using a reverse proxy
3. Inside IIS using the WSGI bridge with ISAPI-WSGI
4. Inside IIS using the WSGI bridge with PyISAPIe

Options 1 and 2: run as a Windows service

Both Options 1 and 2 are quite similar to running the development server, except that debugging info is turned off and you want to run the process as a Windows service.

Install dependencies

Running as a Windows service depends on the PyWin32 project. You will need to download the pre-built binary that matches your version of Python.

You can install directly into the virtualenv if you run `easy_install` on the downloaded installer. For example:

```
easy_install pywin32-217.win32-py2.7.exe
```

Since the web server for CherryPy has good Windows support, is available for Python 2 and 3, and can be gracefully started and stopped on demand from the service, we'll use that as the web server. You could also substitute another web server, like the one from Twisted.

To install CherryPy run:

```
pip install cherrypy
```

Create a Windows service

Create a new file called `pyramidsvc.py` with the following code to define your service:

```python
# uncomment the next import line to get print to show up or see early
# exceptions if there are errors then run
#   python -m win32traceutil
# to see the output
#import win32traceutil
import win32serviceutil

PORT_TO_BIND = 80
CONFIG_FILE = 'production.ini'
SERVER_NAME = 'www.pyramid.example'

SERVICE_NAME = "PyramidWebService"
SERVICE_DISPLAY_NAME = "Pyramid Web Service"
SERVICE_DESCRIPTION = """This will be displayed as a description \
of the serivice in the Services snap-in for the Microsoft \
Management Console."""

class PyWebService(win32serviceutil.ServiceFramework):
    """Python Web Service."""

    _svc_name_ = SERVICE_NAME
    _svc_display_name_ = SERVICE_DISPLAY_NAME
    _svc_deps_ = None        # sequence of service names on which this depends
    # Only exists on Windows 2000 or later, ignored on Windows NT
    _svc_description_ = SERVICE_DESCRIPTION

    def SvcDoRun(self):
        from cherrypy import wsgiserver
        from pyramid.paster import get_app
        import os, sys

        path = os.path.dirname(os.path.abspath(__file__))

        os.chdir(path)

        app = get_app(CONFIG_FILE)

        self.server = wsgiserver.CherryPyWSGIServer(
                ('0.0.0.0', PORT_TO_BIND), app,
                server_name=SERVER_NAME)

        self.server.start()


    def SvcStop(self):
        self.server.stop()


if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(PyWebService)
```

The `if __name__ == '__main__'` block provides an interface to register the service. You can register the service with the system by running:

```
python pyramidsvc.py install
```

Your service is now ready to start, you can do this through the normal service snap-in for the Microsoft Management Console or by running:

```
python pyramidsvc.py start
```

If you want your service to start automatically you can run:

```
python pyramidsvc.py update --start=auto
```

Reverse proxy (optional)

If you want to run many Pyramid applications on the same machine you will need to run each of them on a different port and in a separate Service. If you want to be able to access each one through a different host name on port 80, then you will need to run another web server (IIS or Apache) up front and proxy back to the appropriate service.

There are several options available for reverse proxy with IIS. In versions starting with IIS 7, you can install and use the Application Request Routing if you want to use a Microsoft-provided solution. Another option is one of the several solutions from Helicon Tech. Helicon Ape is available without cost for up to 3 sites.

If you aren't already using IIS, Apache is available for Windows and works well. There are many reverse proxy tutorials available for Apache, and they are all applicable to Windows.

Options 3 and 4: Inside IIS using the WSGI bridge with ISAPI-WSGI

IIS configuration

Turn on Windows feature for IIS.

Control panel -> "Turn Windows features on off" and select:

- Internet Information service (all)

- World Wide Web Services (all)

Create website

Go to Internet Information Services Manager and add website.

- Site name (your choice)
- Physical path (point to the directory of your Pyramid porject)
- select port
- select the name of your website

Python

- Install PyWin32, according to your 32- or 64-bit installation
- Install isapi-wsgi

Create bridging script

Create a file `install_website.py`, and place it in your pyramid project:

```
# path to your site packages in your environment
# needs to be put in here
import site
site.addsitedir('/path/to/your/site-packages')

# this is used for debugging
# after everything was installed and is ready to meka a http request
# run this from the command line:
# python -m python -m win32traceutil
# It will give you debug output from this script
# (remove the 3 lines for production use)
import sys
if hasattr(sys, "isapidllhandle"):
    import win32traceutil


# this is for setting up a path to a temporary
# directory for egg cache.
import os
os.environ['PYTHON_EGG_CACHE'] = '/path/to/writable/dir'

# The entry point for the ISAPI extension.
def __ExtensionFactory__():
    from paste.deploy import loadapp
    import isapi_wsgi
    from logging.config import fileConfig

    appdir = '/path/to/your/pyramid/project'
    configfile = 'production.ini'
    con = appdir + configfile

    fileConfig(con)
    application = loadapp('config:' + configfile, relative_to=appdir)
    return isapi_wsgi.ISAPIThreadPoolHandler(application)

# ISAPI installation
if __name__ == '__main__':
    from isapi.install import ISAPIParameters, ScriptMapParams, VirtualDirParameters, HandleCommandLi

    params = ISAPIParameters()
    sm = [
        ScriptMapParams(Extension="*", Flags=0)
    ]

    # if name = "/" then it will install on root
    # if any other name then it will install on virtual host for that name
    vd = VirtualDirParameters(Name="/",
                              Description="Description of your proj",
                              ScriptMaps=sm,
                              ScriptMapUpdate="replace"
    )

    params.VirtualDirs = [vd]
    HandleCommandLine(params)
```

Install your Pyramid project as Virtual Host or root feature

Activate your virtual env and run the stript:

```
python install_website.py install --server=<name of your website>
```

Restart your website from IIS.

1.6 Forms

Pyramid does not include a form library because there are several good ones on PyPI, but none that is obviously better than the others.

Deform is a form library written for Pyramid, and maintained by the Pylons Project. It has a demo.

You can use WebHelpers and FormEncode in Pyramid just like in Pylons. Use pyramid_simpleform to organize your view code. (This replaces Pylons' @validate decorator, which has no equivalent in Pyramid.) FormEncode's documentation is a bit obtuse and sparse, but it's so widely flexible that you can do things in FormEncode that you can't in other libraries, and you can also use it for non-HTML validation; e.g., to validate the settings in the INI file.

Some Pyramid users have had luck with WTForms, Formish, ToscaWidgets, etc.

There are also form packages tied to database records, most notably FormAlchemy. These will publish a form to add/modify/delete records of a certain ORM class.

1.6.1 Articles

File Uploads

There are two parts necessary for handling file uploads. The first is to make sure you have a form that's been setup correctly to accept files. This means adding `enctype` attribute to your `form` element with the value of multipart/form-data. A very simple example would be a form that accepts an mp3 file. Notice we've setup the form as previously explained and also added an input element of the file type.

```
<form action="/store_mp3_view" method="post" accept-charset="utf-8"
      enctype="multipart/form-data">

    <label for="mp3">Mp3</label>
    <input id="mp3" name="mp3" type="file" value="" />

    <input type="submit" value="submit" />
</form>
```

The second part is handling the file upload in your view callable (above, assumed to answer on /store_mp3_view). The uploaded file is added to the request object as a cgi.FieldStorage object accessible through the request.POST multidict. The two properties we're interested in are the file and filename and we'll use those to write the file to disk:

```
import os
import uuid
import shutil
from pyramid.response import Response

def store_mp3_view(request):
    # ``filename`` contains the name of the file in string format.
    #
    # WARNING: this example does not deal with the fact that IE sends an
    # absolute file *path* as the filename.  This example is naive; it
    # trusts user input.

    filename = request.POST['mp3'].filename

    # ``input_file`` contains the actual file data which needs to be
    # stored somewhere.

    input_file = request.POST['mp3'].file

    # Note that we are generating our own filename instead of trusting
    # the incoming filename since that might result in insecure paths.
    # Please note that in a real application you would not use /tmp,
    # and if you write to an untrusted location you will need to do
    # some extra work to prevent symlink attacks.

    file_path = os.path.join('/tmp', '%s.mp3' % uuid.uuid4())

    # We first write to a temporary file to prevent incomplete files from
    # being used.

    temp_file_path = file_path + '~'

    # Finally write the data to a temporary file
    input_file.seek(0)
    with open(temp_file_path, 'wb') as output_file:
        shutil.copyfileobj(input_file, output_file)

    # Now that we know the file has been fully saved to disk move it into place.

    os.rename(temp_file_path, file_path)

    return Response('OK')
```

1.7 Logging

1.7.1 Logging Exceptions To Your SQLAlchemy Database

So you'd like to log to your database, rather than a file. Well, here's a brief rundown of exactly how you'd do that.

First we need to define a Log model for SQLAlchemy (do this in `myapp.models`):

```python
from sqlalchemy import Column
from sqlalchemy.types import DateTime, Integer, String
from sqlalchemy.sql import func
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Log(Base):
    __tablename__ = 'logs'
    id = Column(Integer, primary_key=True) # auto incrementing
    logger = Column(String) # the name of the logger. (e.g. myapp.views)
    level = Column(String) # info, debug, or error?
    trace = Column(String) # the full traceback printout
    msg = Column(String) # any custom log you may have included
    created_at = Column(DateTime, default=func.now()) # the current timestamp

    def __init__(self, logger=None, level=None, trace=None, msg=None):
        self.logger = logger
        self.level = level
        self.trace = trace
        self.msg = msg

    def __unicode__(self):
        return self.__repr__()

    def __repr__(self):
        return "<Log: %s - %s>" % (self.created_at.strftime('%m/%d/%Y-%H:%M:%S'), self.msg[:50])
```

Not too much exciting is occuring here. We've simply created a new table named 'logs'.

Before we get into how we use this table for good, here's a quick review of how `logging` works in a script:

```python
# http://docs.python.org/howto/logging.html#configuring-logging
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
```

```
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

What you should gain from the above intro is that your `handler` uses a `formatter` and does the heavy lifting of executing the output of the `logging.LogRecord`. The output actually comes from `logging.Handler.emit`, a method we will now override as we create our SQLAlchemyHandler.

Let's subclass Handler now (put this in `myapp.handlers`):

```python
import logging
import traceback

import transaction

from models import Log, DBSession

class SQLAlchemyHandler(logging.Handler):
    # A very basic logger that commits a LogRecord to the SQL Db
    def emit(self, record):
        trace = None
        exc = record.__dict__['exc_info']
        if exc:
            trace = traceback.format_exc(exc)
        log = Log(
            logger=record.__dict__['name'],
            level=record.__dict__['levelname'],
            trace=trace,
            msg=record.__dict__['msg'],)
        DBSession.add(log)
        transaction.commit()
```

For a little more depth, `logging.LogRecord`, for which `record` is an instance, contains all it's nifty log information in it's `__dict__` attribute.

Now, we need to add this logging handler to our .ini configuration files. Before we add this, our production.ini file should contain something like:

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

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = WARN
handlers = console

[logger_myapp]
level = WARN
handlers =
qualname = myapp

[logger_sqlalchemy]
level = WARN
handlers =
qualname = sqlalchemy.engine
# "level = INFO" logs SQL queries.
# "level = DEBUG" logs SQL queries and results.
# "level = WARN" logs neither.  (Recommended for production systems.)

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

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

We must add our `SQLAlchemyHandler` to the mix. So make the following changes to your production.ini file.

```
[handlers]
keys = console, sqlalchemy

[logger_myapp]
level = DEBUG
handlers = sqlalchemy
qualname = myapp

[handler_sqlalchemy]
class = myapp.handlers.SQLAlchemyHandler
args = ()
level = NOTSET
formatter = generic
```

The changes we made simply allow Paster to recognize a new handler - `sqlalchemy`, located at `[handler_sqlalchemy]`. Most everything else about this configuration should be straightforward. If anything is still baffling, then use this as a good opportunity to read the Python `logging` documentation.

Below is an example of how you might use the logger in `myapp.views`:

```python
import logging
from pyramid.view import view_config
from pyramid.response import Response

log = logging.getLogger(__name__)

@view_config(route_name='home')
def root(request):
    log.debug('exception impending!')
    try:
        1/0
    except:
        log.exception('1/0 error')
    log.info('test complete')
    return Response("test complete!")
```

When this view code is executed, you'll see up to three (depending on the level of logging you allow in your configuation file) records!

For more power, match this up with pyramid_exclog at http://docs.pylonsproject.org/projects/pyramid_exclog/en/latest/

For more information on logging, see the Logging section of the Pyramid documentation.

1.8 Porting Applications to Pyramid

Note: Other articles about Pylons applications are in the Pyramid for Pylons Users section.

1.8.1 Porting a Legacy Pylons Application Piecemeal

You would like to move from Pylons 1.0 to Pyramid, but you're not going to be able manage a wholesale port any time soon. You're wondering if it would be practical to start using some parts of Pyramid within an existing Pylons project.

One idea is to use a Pyramid "NotFound view" which delegates to the existing Pylons application, and port piecemeal:

```python
# ... obtain pylons WSGI application object ...
from mypylonsproject import thepylonsapp

class LegacyView(object):
    def __init__(self, app):
        self.app = app
    def __call__(self, request):
        return request.get_response(self.app)

if __name__ == '__main__':
   legacy_view = LegacyView(thepylonsapp)
   config = Configurator()
   config.add_view(context='pyramid.exceptions.NotFound', view=legacy_view)
   # ... rest of config ...
```

At that point, whenever Pyramid cannot service a request because the URL doesn't match anything, it will invoke the Pylons application as a fallback, which will return things normally. At that point you can start moving logic incrementally into Pyramid from the Pylons application until you've ported everything.

1.8.2 Porting an Existing WSGI Application to Pyramid

Pyramid is cool, but already-working code is cooler. You may not have the time, money or energy to port an existing Pylons, Django, Zope, or other WSGI-based application to Pyramid wholesale. In such cases, it can be useful to *incrementally* port an existing application to Pyramid.

The broad-brush way to do this is:

- Set up an *exception view* that will be called whenever a NotFound exception is raised by Pyramid.
- In this exception view, delegate to your already-written WSGI application.

Here's an example:

```python
from pyramid.wsgi import wsgiapp2
from pyramid.exceptions import NotFound

if __name__ == '__main__':
    # during Pyramid configuration (usually in your Pyramid project's
    # __init__.py), get a hold of an instance of your existing WSGI
    # application.
    original_app = MyWSGIApplication()

    # using the pyramid.wsgi.wsgiapp2 wrapper function, wrap the
    # application into something that can be used as a Pyramid view.
    notfound_view = wsgiapp2(original_app)

    # in your configuration, use the wsgiapp2-wrapped application as
    # a NotFound exception view
    config = Configurator()

    # ... your other Pyramid configuration ...
    config.add_view(notfound_view, context=NotFound)
    # .. the remainder of your configuration ...
```

When Pyramid cannot resolve a URL to a view, it will raise a NotFound exception. The `add_view` statement in the example above configures Pyramid to use your original WSGI application as the NotFound view. This means that whenever Pyramid cannot resolve a URL, your original application will be called.

Incrementally, you can begin moving features from your existing WSGI application to Pyramid; if Pyramid can resolve a request to a view, the Pyramid "version" of the application logic will be used. If it cannot, the original WSGI application version of the logic will be used. Over time, you can move *all* of the logic into Pyramid without needing to do it all at once.

1.9 Pyramid for Pylons Users

Updated 2012-06-12

Versions Pyramid 1.3

Author Mike Orr

Contributors

This guide discusses how Pyramid 1.3 differs from Pylons 1, and a few ways to make it more like Pylons. The guide may also be helpful to readers coming from Django or another Rails-like framework. The author has been a Pylons developer since 2007. The examples are based on Pyramid's default SQLAlchemy application and on the Akhet demo.

If you haven't used Pyramid yet you can read this guide to get an overview of the differences and the Pyramid API. However, to actually start using Pyramid you'll want to read at least the first five chapters of the Pyramid manual (through Creating a Pyramid Project) and go through the Tutorials. Then you can come back to this guide to start designing your application, and skim through the rest of the manual to see which sections cover which topics.

1.9.1 Introduction and Creating an Application

Following along with the examples

The examples in this guide are based on (A) Pyramid 1.3's default SQLAlchemy application and (B) the Akhet demo. (Akhet is an add-on package containing some Pylons-like support features for Pyramid.) Here are the basic steps to install and run these applications on Linux Ubuntu 11.10, but you should read Creating a Pyramid Project in the Pyramid manual before doing so:

```
# Prepare virtual Python environment.

$ cd ~/workspace
$ virtualenv myvenv
$ source myvenv/bin/activate
(myvenv)$ pip install 'Pyramid>=1.3'
```

```
7
8   # Create a Pyramid "alchemy" application and run it.
9
10  (myvenv)$ pcreate -s alchemy PyramidApp
11  (myvenv)$ cd PyramidApp
12  (myvenv)$ pip install -e .
13  (myvenv)$ initialize_PyramidApp_db development.ini
14  (myvenv)$ pserve development.ini
15  Starting server in PID 3871.
16  serving on http://0.0.0.0:6543
17
18  # Press ctrl-C to quit server
19
20  # Check out the Akhet demo and run it.
21
22  (myvenv)$ git clone git://github.com/mikeorr/akhet_demo
23  (myvenv)$ cd akhet_demo
24  (myvenv)$ pip install -e .
25  (myvenv)$ pserve development.ini
26  Starting server in PID 3871.
27  serving on http://0.0.0.0:6543
28
29  # Check out the Pyramid source and Akhet source to study.
30
31  (myvenv)$ git clone git://github.com/pylons/pyramid
32  (myvenv)$ git clone git://github.com/pylons/akhet
33
34  (myvenv)$ ls -F
35  akhet/
36  akhet_demo/
37  PyramidApp/
38  pyramid/
39  myvenv/
```

Things to look for: the "DT" icon at the top-right of the page is the debug toolbar, which Pylons doesn't have. The "populate_PyramidApp" script (line 13) creates the database. If you skip this step you'll get an exception on the home page; you can "accidentally" do this to see Pyramid's interactive traceback.

The p* Commands

Pylons uses a third-party utility *paster* to create and run applications. Pyramid replaces these subcommands with a series of top-level commands beginning with "p":

Pylons	Pyramid	Description	Caveats
paster create	pcreate	Create an app	Option -s instead of -t
paster serve	pserve	Run app based on INI file	-
paster shell	pshell	Load app in Python shell	Fewer vars initialized
paster setup-app	populate_App	Initialize database	"App" is application name
paster routes	proutes	List routes	-
-	ptweens	List tweens	-
-	pviews	List views	-

In many cases the code is the same, just copied into Pyramid and made Python 3 compatible. Paste has not been ported to Python 3, and the Pyramid developers decided it contained too much legacy code to make porting worth it. So they just ported the parts they needed. Note, however, that PasteDeploy *is* ported to Python 3 and Pyramid uses it, as we'll see in the next chapter. Likewise, several other packages that were earlier spun out of Paste – like WebOb – have been ported to Python 3 and Pyramid still uses them. (They were ported parly by Pyramid developers.)

Scaffolds

Pylons has one paster template that asks questions about what kind of application you want to create. Pyramid does not ask questions, but instead offers several scaffolds to choose from. Pyramid 1.3 includes the following scaffolds:

Routing mechanism	Database	Pyramid scaffold
URL dispatch	SQLAlchemy	alchemy
URL dispatch	-	starter
Traversal	ZODB	zodb

The first two scaffolds are the closest to Pylons because they use URL dispatch, which is similar to Routes. The only difference between them is whether a SQLAlchemy database is configured for you. The third scaffold uses Pyramid's other routing mechanism, Traversal. We won't cover traversal in this guide, but it's useful in applications that allow users to create URLs at arbitrary depths. URL dispatch is more suited to applications with fixed-depth URL hierarchies.

To see what other kinds of Pyramid applications are possible, take a look at the Kotti and Ptah distributions. Kotti is a content management system, and serves as an example of traversal using SQLAlchemy.

Directory Layout

The default 'alchemy' application contains the following files after you create and install it:

```
PyramidApp
-- CHANGES.txt
-- MANIFEST.in
-- README.txt
-- development.ini
-- production.ini
-- setup.cfg
-- setup.py
-- pyramidapp
|   -- __init__.py
|   -- models.py
|   -- scripts
|   |   -- __init__.py
|   |   -- populate.py
|   -- static
|   |   -- favicon.ico
|   |   -- pylons.css
|   |   -- pyramid.png
|   -- templates
|   |   -- mytemplate.pt
|   -- tests.py
|   -- views.py
-- PyramidApp.egg-info
    -- PKG-INFO
    -- SOURCES.txt
    -- dependency_links.txt
    -- entry_points.txt
    -- not-zip-safe
    -- requires.txt
    -- top_level.txt
```

(We have omitted some static files.) As you see, the directory structure is similar to Pylons but not identical.

1.9.2 Launching the Application

Pyramid and Pylons start up identically because they both use PasteDeploy and its INI-format configuration file. This is true even though Pyramid 1.3 replaced “paster serve” with its own “pserve” command. Both “pserve” and “paster serve” do the following:

1. Read the INI file.
2. Instantiate an application based on the “[app:main]” section.
3. Instantiate a server based on the “[server:main]” section.
4. Configure Python logging based on the logging sections.
5. Call the server with the application.

Steps 1-3 and 5 are essentially wrappers around PasteDeploy. Only step 2 is really “using Pyramid”, because only the application depends on other parts of Pyramid. The rest of the routine is copied directly from “paster serve” and does not depend on other parts of Pyramid.

The way the launcher instantiates an application is often misunderstood so let’s stop for a moment and detail it. Here’s part of the app section in the Akhet Demo:

```
[app:main]
use = egg:akhet_demo#main
pyramid.reload_templates = true
pyramid.debug_authorization = false
```

The “use=” line indirectly names a Python callable to load. “egg:” says to look up a Python object by entry point. (Entry points are a feature provided by Setuptools, which is why Pyramid/Pylons require it or Distribute to be installed.) “akhet_demo” is the name of the Python distribution to look in (the Pyramid application), and “main” is the entry point. The launcher calls `pkg_resources.require("akhet_demo#main")` in Setuptools, and Setuptools returns the Python object. Entry points are defined in the distribution’s setup.py, and the installer writes them to an entry points file. Here’s the *akhet_demo.egg-info/entry_points.txt* file:

```
[paste.app_factory]
main = akhet_demo:main
```

“paste.app_factory” is the entry point group, a name publicized in the PasteDeploy docs for all applications that want to be compatible with it. “main” (on the left side of the equal sign) is the entry point. “akhet_demo:main” says to import the `akhet_demo` package and load a “main” attribute. This is our `main()` function defined in *akhet_demo/__init__.py*. The other options in the “[app:main]” section become keyword arguments to this callable. These options are called “settings” in Pyramid and “config variables” in Pylons. (The options in the “[DEFAULT]” section are also passed as default values.) Both frameworks provide a way to access these variables in application code. In Pyramid they’re in the `request.registry.settings` dict. In Pylons they’re in the `pylons.config` magic global.

The launcher loads the server in the same way, using the “[server:main]” section.

More details: The heavy lifting is done by `loadapp()` and `loadserver()` in `paste.deploy.loadwsgi`. Loadwsgi is obtuse and undocumented, but `pyramid.paster` has some convenience functions that either call or mimic some of its routines.

Alternative launchers such as mod_wsgi read only the “[app:main]” section and ignore the server section, but they’re still using PasteDeploy or the equivalent. It’s also possible to instantiate the application manually without an INI file or PasteDeploy, as we’ll see in the chapter called “The Main Function”.

Now that we know more about how the launcher loads the application, let’s look closer at a Pyramid application itself.

1.9.3 INI File

The "[app:main]" section in Pyramid apps has different options than its Pylons counterpart. Here's what it looks like in Pyramid's "alchemy" scaffold:

```
[app:main]
use = egg:{{project}}

pyramid.reload_templates = true
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.debug_routematch = false
pyramid.debug_templates = true
pyramid.default_locale_name = en
pyramid.includes =
    pyramid_debugtoolbar
    pyramid_tm

sqlalchemy.url = sqlite:///%(here)s/{{project}}.db
```

The "pyramid.includes=" variable lists a number of "tweens" to activate. A tween is like a WSGI middleware but specific to Pyramid. "pyramid_debugtoolbar" is the debug toolbar; it provides information on the request variables and runtime state on every page.

"pyramid_tm" is a transaction manager. This has no equivalent in Pylons but is used in TurboGears and BFG. It provides a request-wide transaction that manages your SQLAlchemy session(s) and potentially other kinds of transactions like email sending. This means you don't have to call `DBSession.commit()` in your view. At the end of the request, it will automatically commit the database session(s) and send any pending emails, unless an uncaught exception was raised during the session, in which case it will roll them back. It has functions to allow you to commit or roll back the request-wide transaction at any time, or to "doom" it to prevent any other code from committing anything.

The other "pyramid.*" options are for debugging. Set any of these to true to tell that subsystem to log what it's doing. The messages will be logged at the DEBUG level. (The reason these aren't in the logging configuration in the bottom part of the INI file is that they were established early in Pyramid's history before it had adopted INI-style logging configuration.)

If "pyramid.reload_templates=true", the template engine will check the timestamp of the template source file every time it renders a template, and recompile the template if its source has changed. This works only for template engines and Pyramid-template adapaters that support this feature. Mako and Chameleon do.

The "sqlalchemy.url=" line is for SQLAlchemy. "%(here)s" expands to the path of the directory containing the INI file. You can add settings for any library that understands them, including SQLAlchemy, Mako, and Beaker. You can also define custom settings that your application code understands, so that you can deploy it with different configurations without changing the code. This is all the same as in Pylons.

production.ini has the same app settings as *development.ini*, except that the "pyramid_debugtoolbar" tween is *not* present, and all the debug settings are false. The debug toolbar *must* be disabled in production because it's a potential security hole: anybody who can force an exception and get an interactive traceback can run arbitrary Python commmands in the application process, and thus read or modify files or execute programs. So never enable the debug toolbar when the site is accessible on the Internet, except perhaps in a wide-area development scenario where higher-level access restrictions (Apache) allow only trusted developers and beta testers to get to the site.

Pyramid no longer uses WSGI middleware by default. In most cases you can find a tween or Pyramid add-on package that does the equivalent. If you need to activate your own middleware, do it the same way as in Pylons; the syntax is in the PasteDeploy manual. But first consider whether making a Pyramid tween would be just as convenient. Tweens have a much simpler API than middleware, and have access to the view's request and response objects. The WSGI protocol is extraordinarily difficult to implement correctly due to edge cases, and many existing middlewares are incorrect. Let server developers and framework developers worry about those issues; you can just write a tween and be out on the golf course by 3pm.

1.9.4 The Main Function

Both Pyramid and Pylons have a top-level function that returns a WSGI application. The Pyramid function is `main` in *pyramidapp/__init__.py*. The Pylons function is `make_app` in *pylonsapp/config/middleware.py*. Here's the main function generated by Pyramid's 'starter' scaffold:

```
1  from pyramid.config import Configurator
2  
3  def main(global_config, **settings):
4      """ This function returns a Pyramid WSGI application.
5      """
6      config = Configurator(settings=settings)
7      config.add_static_view('static', 'static', cache_max_age=3600)
8      config.add_route('home', '/')
9      config.scan()
10     return config.make_wsgi_app()
```

Pyramid has less boilerplate code than Pylons, so the main function subsumes Pylons' middleware.py, environment.py, *and* routing.py modules. Pyramid's configuration code is just 5 lines long in the default application, while Pylons' is 35.

Most of the function's body deals with the Configurator (`config`). That isn't the application object; it's a helper that will instantiate the application for us. You pass in the settings as a dict to the constructor (line 6), call various methods to set up routes and such, and finally call `config.make_wsgi_app()` to get the application, which the main function returns. The application is an instance of `pyramid.router.Router`. (A Pylons application is an instance of a `PylonsApp` subclass.)

Dotted Python names and asset specifications

Several config methods accept either an object (e.g., a module or callable) or a string naming the object. The latter is called a *dotted Python name*. It's a dot-delimited string specifying the absolute name of a module or a top-level object in a module: "module", "package.module", "package.subpackage.module.attribute". Passing string names allows you to avoid importing the object merely to pass it to a method.

If the string starts with a leading dot, it's relative to some parent package. So in this `main` function defined in *mypyramiapp/__init__.py*, the parent package is `mypyramidapp`. So the name ".views" refers to *mypyramidapp/views.py*. (Note: in some cases it can sometimes be tricky to guess what Pyramid thinks the parent package is.)

Closely associated with this is a *static asset specification*, which names a non-Python file or directory inside a Python package. A colon separates the package name from the non-Python subpath: "myapp:templates/mytemplate.pt", "myapp:static", "myapp:assets/subdir1". If you leave off the first part and the colon (e.g., "templates/mytemplate.pt", it's relative to some current package.

An alternative syntax exists, with a colon between a module and an attribute: "package.module:attribute". This usage is discouraged; it exists for compatibility with Setuptools' resource syntax.

Configurator methods

The Configurator has several methods to customize the application. Below are the ones most commonly used in Pylons-like applications, in order by how widely they're used. The full list of methods is in Pyramid's Configurator API.

`add_route`(...)
: Register a route for URL dispatch.

`add_view`(...)
: Register a view. Views are equivalent to Pylons' controller actions.

scan (...)
A wrapper for registering views and certain other things. Discussed in the views chapter.

add_static_view (...)
Add a special view that publishes a directory of static files. This is somewhat akin to Pylons' public directory, but see the static fiels chapter for caveats.

include (*callable*, *route_prefix=None*)
Allow a function to customize the configuration further. This is a wide-open interface which has become very popular in Pyramid. It has three main use cases:

- To group related code together; e.g., to define your routes in a separate module.
- To initialize a third-party add-on. Many add-ons provide an include function that performs all the initialization steps for you.
- To mount a subapplication at a URL prefix. A subapplication is just any bundle of routes, views and templates that work together. You can use this to split your application into logical units. Or you can write generic subapplications that can be used in several applications, or mount a third-party subapplication.

If the add-on or subapplication has options, it will typically read them from the settings, looking for settings with a certain prefix and converting strings to their proper type. For instance, a session manager may look for keys starting with "session." or "thesessionmanager." as in "session.type". Consult the add-on's documentation to see what prefix it uses and which options it recognizes.

The `callable` argument should be a function, a module, or a dotted Python name. If it resolves to a module, the module should contain an `includeme` function which will be called. The following are equivalent:

```python
config.include("pyramid_beaker")

import pyramid_beaker
config.include(pyramid_beaker)

import pyramid_beaker
config.include(pyramid_beaker.includeme)
```

If `route_prefix` is specified, it should be a string that will be prepended to any URLs generated by the subconfigurator's `add_route` method. Caution: the route *names* must be unique across the main application and all subapplications, and `route_prefix` does not touch the names. So you'll want to name your routes "subapp1.route1" or "subapp1_route1" or such.

add_subscriber (*subscriber*, *iface=None*)
Insert a callback into Pyramid's event loop to customize how it processes requests. The Renderers chapter has an example of its use.

add_renderer (*name*, *factory*)
Add a custom renderer. An example is in the Renderers chapter.

set_authentication_policy, set_authorization_policy, set_default_permission
Configure Pyramid's built-in authorization mechanism.

Other methods sometimes used: `add_notfound_view`, `add_exception_view`, `set_request_factory`, `add_tween`, `override_asset` (used in theming). Add-ons can define additional config methods by calling `config.add_directive`.

Route arguments

`config.add_route` accepts a large number of keyword arguments. They are logically divided into *predicate argumets* and *non-predicate arguments*. Predicate arguments determine whether the route matches the current request.

All predicates must succeed in order for the route to be chosen. Non-predicate arguments do not affect whether the route matches.

name

[Non-predicate] The first positional arg; required. This must be a unique name for the route. The name is used to identify the route when registering views or generating URLs.

pattern

[Predicate] The second positional arg; required. This is the URL path with optional "{variable}" placeholders; e.g., "/articles/{id}" or "/abc/{filename}.html". The leading slash is optional. By default the placeholder matches all characters up to a slash, but you can specify a regex to make it match less (e.g., "{variable:d+}" for a numeric variable) or more ("{variable:.*}" to match the entire rest of the URL including slashes). The substrings matched by the placeholders will be available as *request.matchdict* in the view.

A wildcard syntax "*varname" matches the rest of the URL and puts it into the matchdict as a tuple of segments instead of a single string. So a pattern "/foo/{action}/*fizzle" would match a URL "/foo/edit/a/1" and produce a matchdict `{'action': u'edit', 'fizzle': (u'a', u'1')}`.

Two special wildcards exist, "*traverse" and "*subpath". These are used in advanced cases to do traversal on the remainder of the URL.

XXX Should use raw string syntax for regexes with backslashes (d) ?

request_method

[Predicate] An HTTP method: "GET", "POST", "HEAD", "DELETE", "PUT". Only requests of this type will match the route.

request_param

[Predicate] If the value doesn't contain "=" (e.g., "q"), the request must have the specified parameter (a GET or POST variable). If it does contain "=" (e.g., "name=value"), the parameter must also have the specified value.

This is especially useful when tunnelling other HTTP methods via POST. Web browsers can't submit a PUT or DELETE method via a form, so it's customary to use POST and to set a parameter `_method="PUT"`. The framework or application sees the "_method" parameter and pretends the other HTTP method was requested. In Pyramid you can do this with `request_param="_method=PUT`.

xhr

[Predicate] True if the request must have an "X-Requested-With" header. Some Javascript libraries (JQuery, Prototype, etc) set this header in AJAX requests to distinguish them from user-initiated browser requests.

custom_predicates

[Predicate] A sequence of callables which will be called to determine whether the route matches the request. Use this feature if none of the other predicate arguments do what you need. The request will match the route only if *all* callables return `True`. Each callable will receive two arguments, `info` and `request`. `request` is the current request. `info` is a dict containing the following:

```
info["match"]  =>  the match dict for the current route
info["route"].name  =>  the name of the current route
info["route"].pattern  =>  the URL pattern of the current route
```

You can modify the match dict to affect how the view will see it. For instance, you can look up a model object based on its ID and put the object in the match dict under another key. If the record is not found in the model, you can return False.

Other arguments available: accept, factory, header, path_info, traverse.

1.9.5 Models

Models are essentially the same in Pyramid and Pylons because the framework is only minimally involved with the model unlike, say, Django where the ORM (object-relational mapper) is framework-specific and other parts of the framework assume it's that specific kind. In Pyramid and Pylons, the application skeleton merely suggests where to put the models and initializes a SQLAlchemy database connection for you. Here's the default Pyramid configuration (comments stripped and imports squashed):

```python
# pyramidapp/__init__.py
from sqlalchemy import engine_from_config
from .models import DBSession

def main(global_config, **settings):
    engine = engine_from_config(settings, 'sqlalchemy.')
    DBSession.configure(bind=engine)
    ...


# pyramidapp/models.py
from sqlalchemy import Column, Integer, Text
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker
from zope.sqlalchemy import ZopeTransactionExtension

DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
Base = declarative_base()

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

    def __init__(self, name, value):
        self.name = name
        self.value = value
```

and its INI files:

```ini
# development.ini
[app:main]

# Pyramid only
pyramid.includes =
    pyramid_tm

# Pyramid and Pylons
sqlalchemy.url = sqlite:///%(here)s/PyramidApp.db


[logger_sqlalchemy]

# Pyramid and Pylons
level = INFO
handlers =
qualname = sqlalchemy.engine
```

```
# "level = INFO" logs SQL queries.
# "level = DEBUG" logs SQL queries and results.
# "level = WARN" logs neither.  (Recommended for production systems.)
```

It has the following differences from Pylons:

1. ZopeTransactionExtension and the "pyramid_tm" tween.
2. "models" (plural) instead of "model" (singular).
3. A module rather than a subpackage.
4. "DBSession" instead of "Session".
5. No init_model() function.
6. Slightly different import style and variable naming.

Only the first one is an essential difference; the rest are just aesthetic programming styles. So you can change them without harming anything.

The model-models difference is due to an ambiguity in how the word "model" is used. Some people say "a model" to refer to an individual ORM class, while others say "the model" to refer to the entire collection of ORM classes in an application. This guide uses the word both ways.

What belongs in the model?

Good programming practice recommends keeping your data classes separate from user-interface classes. That way the user interface can change without affecting the data and vice-versa. The model is where the data classes go. For instance, a Monopoly game has players, a board, squares, title deeds, cards, etc, so a Monopoly program would likely have classes for each of these. If the application requires saving data between runs (persistence), the data will have to be stored in a database or equivalent. Pyramid can work with a variety of database types: SQL database, object database, key-value database ("NoSQL"), document database (JSON or XML), CSV files, etc. The most common choice is SQLAlchemy, so that's the first configuration provided by Pyramid and Pylons.

At minimum you should define your ORM classes in the model. You can also add any business logic in the form of functions, class methods, or regular methods. It's sometimes difficult to tell whether a particular piece of code belongs in the model or the view, but we'll leave that up to you.

Another principle is that the model should not depend on the rest of the application so that it can be used on its own; e.g., in utility programs or in other applications. That also allows you to extract the data if the framework or application breaks. So the view knows about the model but not vice-versa. Not everybody agrees with this but it's a good place to start.

Larger projects may share a common model between multiple web applications and non-web programs. In that case it makes sense to put the model in a separate top-level package and import it into the Pyramid application.

Transaction manger

Pylons has never used a transaction manager but it's common in TurboGears and Zope. A transaction manager takes care of the commit-rollback cycle for you. The database session in both applications above is a *scoped* session, meaning it's a threadlocal global and must be cleared out at the end of every request. The Pylons app has special code in the base controller to clear out the session. A transaction manager takes this a step further by committing any changes made during the request, or if an exception was raised during the request, it rolls back the changes. The ZopeTransactionExtension provides a module-level API in case the view wants to customize when/whether committing occurs.

The upshot is that your view method does not have to call `DBSession.commit()`: the transaction manager will do it for you. Also, it doesn't have to put the changes in a try-except block because the transaction manager will call `DBSession.rollback()` if an exception occurs. (Many Pylons actions don't do this so they're technically incorrect.) A side effect is that you *cannot* call `DBSession.commit()` or `DBSession.rollback()` directly. If you want to precisely control when something is committed, you'll have to do it this way:

```
import transaction

transaction.commit()
# Or:
transaction.rollback()
```

There's also a `transaction.doom()` function which you can call to prevent *any* database writes during this request, including those performed by other parts of the application. Of course, this doesn't affect changes that have already been committed.

You can customize the circumstances under which an automatic rollback occurs by defining a "commit veto" function. This is described in the pyramid_tm documentation.

Using traversal as a model

Pylons doesn't have a traversal mode, so you have to fetch database objects in the view code. Pyramid's traversal mode essentially does this for you, delivering the object to the view as its *context*, and handling "not found" for you. Traversal resource tree thus almost looks like a second kind of model, separate from `models`. (It's typically defined in a `resources` module.) This raises the question of, what's the difference between the two? Does it make sense to convert my model to traversal, or to traversal under the control of a route? The issue comes up further with authorization, because Pyramid's default authorization mechanism is designed for permissions (an access-control list or ACL) to be attached to the *context* object. These are advanced questions so we won't cover them here. Traversal has a learning curve, and it may or may not be appropriate for different kinds of applications. Nevertheless, it's good to know it exists so that you can explore it gradually over time and maybe find a use for it someday.

SQLAHelper and a "models" subpackage

Earlier versions of Akhet used the SQLAHelper library to organize engines and sessions. This is no longer documented because it's not that much benefit. The main thing to remember is that if you split *models.py* into a package, beware of circular imports. If you define the `Base` and `DBSession` in *models/__ini__.py* and import them into submodules, and the init module imports the submodules, there will be a circular import of two modules importing each other. One module will appear semi-empty while the other module is running its global code, which could lead to exceptions.

Pylons dealt with this by putting the Base and Session in a submodule, *models/meta.py*, which did not import any other model modules. SQLAHelper deals with it by providing a third-party library to store engines, sessions, and bases. The Pyramid developers decided to default to the simplest case of the putting entire model in one module, and let you figure out how to split it if you want to.

Model Examples

These examples were written a while ago so they don't use the transaction manager, and they have yet at third importing syntax. They should work with SQLAlchemy 0.6, 0.7, and 0.8.

A simple one-table model

```
import sqlalchemy as sa
import sqlalchemy.orm as orm
import sqlalchemy.ext.declarative as declarative
from zope.sqlalchemy import ZopeTransactionExtension as ZTE
```

```

DBSession = orm.scoped_session(orm.sessionmaker(extension=ZTE()))
Base = declarative.declarative_base()

class User(Base):
    __tablename__ = "users"

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    email = sa.Column(sa.Unicode(100), nullable=False)
```

This model has one ORM class, `User` corresponding to a database table `users`. The table has three columns: `id`, `name`, and `user`.

A three-table model

We can expand the above into a three-table model suitable for a medium-sized application.

```
import sqlalchemy as sa
import sqlalchemy.orm as orm
import sqlalchemy.ext.declarative as declarative
from zope.sqlalchemy import ZopeTransactionExtension as ZTE

DBSession = orm.scoped_session(orm.sessionmaker(extension=ZTE()))
Base = declarative.declarative_base()

class User(Base):
    __tablename__ = "users"

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    email = sa.Column(sa.Unicode(100), nullable=False)

    addresses = orm.relationship("Address", order_by="Address.id")
    activities = orm.relationship("Activity",
        secondary="assoc_users_activities")

    @classmethod
    def by_name(class_):
        """Return a query of users sorted by name."""
        User = class_
        q = DBSession.query(User)
        q = q.order_by(User.name)
        return q


class Address(Base):
    __tablename__ = "addresses"

    id = sa.Column(sa.Integer, primary_key=True)
    user_id = foreign_key_column(None, sa.Integer, "users.id")
    street = sa.Column(sa.Unicode(40), nullable=False)
    city = sa.Column(sa.Unicode(40), nullable=False)
    state = sa.Column(sa.Unicode(2), nullable=False)
    zip = sa.Column(sa.Unicode(10), nullable=False)
    country = sa.Column(sa.Unicode(40), nullable=False)
    foreign_extra = sa.Column(sa.Unicode(100, nullable=False))

    def __str__(self):
        """Return the address as a string formatted for a mailing label."""
        state_zip = u"{0} {1}".format(self.state, self.zip).strip()
        cityline = filterjoin(u", ", self.city, state_zip)
        lines = [self.street, cityline, self.foreign_extra, self.country]
        return filterjoin(u"|n", *lines) + u"\n"


class Activity(Base):
    __tablename__ = "activities"

    id = sa.Column(sa.Integer, primary_key=True)
    activity = sa.Column(sa.Unicode(100), nullable=False)


assoc_users_activities = sa.Table("assoc_users_activities", Base.metadata,
    foreign_key_column("user_id", sa.Integer, "users.id"),
    foreign_key_column("activities_id", sa.Unicode(100), "activities.id"))

# Utility functions
def filterjoin(sep, *items):
    """Join the items into a string, dropping any that are empty.
    """
    items = filter(None, items)
    return sep.join(items)

def foreign_key_column(name, type_, target, nullable=False):
    """Construct a foreign key column for a table.

    ``name`` is the column name. Pass ``None`` to omit this arg in the
    ``Column`` call; i.e., in Declarative classes.

    ``type_`` is the column type.

    ``target`` is the other column this column references.

    ``nullable``: pass True to allow null values. The default is False
    (the opposite of SQLAlchemy's default, but useful for foreign keys).
    """
    fk = sa.ForeignKey(target)
    if name:
        return sa.Column(name, type_, fk, nullable=nullable)
    else:
        return sa.Column(type_, fk, nullable=nullable)
```

This model has a `User` class corresponding to a `users` table, an `Address` class with an `addresses` table, and an `Activity` class with `activities` table. `users` is in a 1:Many relationship with `addresses`. `users` is also in a Many:Many‘‘ relationship with `activities` using the association table `assoc_users_activities`. This is the SQLAlchemy "declarative" syntax, which defines the tables in terms of ORM classes subclassed from a declarative `Base` class. Association tables do not have an ORM class in SQLAlchemy, so we define it using the `Table` constructor as if we weren't using declarative, but it's still tied to the Base's "metadata".

We can add instance methods to the ORM classes and they will be valid for one database record, as with the `Address.__str__` method. We can also define class methods that operate on several records or return a query object, as with the `User.by_name` method.

There's a bit of disagreement on whether `User.by_name` works better as a class method or static method. Normally with class methods, the first argument is called `class_` or `cls` or `klass` and you use it that way throughout the method, but in ORM queries it's more normal to refer to the ORM class by its proper name. But if you do that you're not using the `class_` variable so why not make it a static method? But the method does belong to the class in a way that an ordinary static method does not. I go back and forth on this, and sometimes assign `User = class_` at the beginning of the method. But none of these ways feels completely satisfactory, so I'm not sure which is best.

Common base class

You can define a superclass for all your ORM classes, with common class methods that all of them can use. It will be the parent of the declarative base:

```
class ORMClass(object):
    @classmethod
    def query(class_):
        return DBSession.query(class_)

    @classmethod
    def get(class_, id):
        return Session.query(class_).get(id)

Base = declarative.declarative_base(cls=ORMClass)

class User(Base):
    __tablename__ = "users"

    # Column definitions omitted
```

Then you can do things like this in your views:

```
user_1 = models.User.get(1)
q = models.User.query()
```

Whether this is a good thing or not depends on your perspective.

Multiple databases

The default configuration in the main function configures one database. To connect to multiple databases, list them all in *development.ini* under distinct prefixes. You can put additional engine arguments under the same prefixes. For instance:

Then modify the main function to add each engine. You can also pass even more engine arguments that override any same-name ones in the INI file.

```
engine = sa.engine_from_config(settings, prefix="sqlalchemy.",
    pool_recycle=3600, convert_unicode=True)
stats = sa.engine_from_config(settings, prefix="stats.")
```

At this point you have a choice. Do you want to bind different tables to different databases in the same DBSession? That's easy:

```
DBSession.configure(binds={models.Person: engine, models.Score: stats})
```

The keys in the `binds` dict can be SQLAlchemy ORM classes, table objects, or mapper objects.

But some applications prefer multiple DBSessions, each connected to a different database. Some applications prefer multiple declarative bases, so that different groups of ORM classes have a different declarative base. Or perhaps you want to bind the engine directly to the Base's metadata for low-level SQL queries. Or you may be using a third-party package that defines its own DBSession or Base. In these cases, you'll have to modify the model itself, e.g., to add a DBSession2 or Base2. If the configuration is complex you may want to define a model initialization function like Pylons does, so that the top-level routine (the main function or a standalone utility) only has to make one simple call. Here's a pretty elaborate init routine for a complex application:

```
DBSession1 = orm.scoped_session(orm.sessionmaker(extension=ZTE())
DBSession2 = orm.scoped_session(orm.sessionmaker(extension=ZTE())
Base1 = declarative.declarative_base()
Base2 = declarative.declarative_base()
engine1 = None
engine2 = None

def init_model(e1, e2):
    # e1 and e2 are SQLAlchemy engines. (We can't call them engine1 and
    # engine2 because we want to access globals with the same name.)
    global engine1, engine2
    engine1 = e1
    engine2 = e2
    DBSession1.configure(bind=e1)
    DBSession2.configure(bind=e2)
    Base1.metadata.bind = e1
    Base2.metadata.bind = e2
```

Reflected tables

Reflected tables pose a dilemma because they depend on a live database connection in order to be initialized. But the engine is not known when the model is imported. This situation pretty much requires an initialization function; or at least we haven't found any way around it. The ORM classes can still be defined as module globals (not using the declarative syntax), but the table definitions and mapper calls will have to be done inside the function when the engine is known. Here's how you'd do it non-declaratively:

```
DBSession = orm.scoped_session(orm.sessionmaker(extension=ZTE())
# Not using Base; not using declarative syntax
md = sa.MetaData()
persons = None   # Table, set in init_model().

class Person(object):
    pass

def init_model(engine):
    global persons
    DBSession.configure(bind=engine)
    md.bind = engine
    persons = sa.Table("persons", md, autoload=True, autoload_with=engine)
    orm.mapper(Person, persons)
```

With the declarative syntax, we *think* Michael Bayer has posted recipies for this somewhere, but you'll have to poke around the SQLAlchmey planet to find them. At worst you could put the entire declarative class inside the init_model function and assign it to a global variable.

1.9.6 Views

The biggest difference between Pyramid and Pylons is how views are structured, and how they invoke templates and access state variables. This is a large topic because it touches on templates, renderers, request variables, URL generators, and more, and several of these topics have many facets. So we'll just start somewhere and keep going, and let it organize itself however it falls.

First let's review Pylons' view handling. In Pylons, a view is called an "action", and is a method in a controller class. Pylons has specific rules about the controller's module name, class name, and base class. When Pylons matches a URL to a route, it uses the routes 'controller' and 'action' variables to look up the controller and action. It instantiates the controller and calls the action. The action may take arguments with the same name as routing variables in the route; Pylons will pass in the current values from the route. The action normally returns a string, usually by calling `render(template_name)` to render a template. Alternatively, it can return a WebOb `Response`. The request's state data is handled by magic global variables which contain the values for the current request. (This includes equest parameters, response attributes, template variables, session variables, URL generator, cache object, and an "application globals" object.)

View functions and view methods

A Pyramid *view callable* can be a function or a method, and it can be in any location. The most basic form is a function that takes a request and returns a response:

```
from pyramid.response import Response

def my_view(request):
    return Response("Hello, world!")
```

A view method may be in any class. A class containing view methods is conventionally called a "view class" or a "handler". If a view is a method, the request is passed to the class constructor, and the method is called without arguments.

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

    def my_view(self):
        return Response("Hello, classy world!")
```

The Pyramid structure has three major benefits.

- Most importantly, it's easier to test. A unit test can call a view with a fake request, and get back the dict that would have been passed to the template. It can inspect the data variables directly rather than parsing them out of the HTML.
- It's simpler and more modular. No magic globals.
- You have the freedom to organize views however you like.

Typical view usage

Merely defining a view is not enough to make Pyramid use it. You have to *register* the view, either by calling `config.add_view()` or using the `@view_config` decorator.

The most common way to use views is with the `@view_config` decorator. This both marks the callable as a view and allows you to specify a template. It's also common to define a base class for common code shared by view classes. The following is borrowed from the Akhet demo.

```
from pyramid.view import view_config

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

class Main(Handler):

    @view_config(route_name="home", renderer="index.mako")
    def index(self):
        return {"project": "Akhet Demo"}
```

The application's main function has a `config.scan()` line, which imports all application modules looking for `@view_config` decorators. For each one it calls `config.add_view(view)` with the same keyword arguments. The scanner also recognizes a few other decorators which we'll see later. If you know that all your views are in a certain module or subpackage, you can scan only that one: `config.scan(".views")`.

The example's `@view_config` decorator has two arguments, 'route_name' and 'renderer'. The 'route_name' argument is *required* when using URL dispatch, to tell Pyramid which route should invoke this view. The "renderer" argument names a template to invoke. In this case, the view's return value is a dict of data variables for the template. (This takes the place of Pylons' 'c' variable, and mimics TurboGears' usage pattern.) The renderer takes care of creating a Response object for you.

View configuration arguments

The following arguments can be passed to `@view_config` or `config.add_view`. If you have certain argument values that are the same for all of the views in a class, you can use `@view_defaults` on the *class* to specify them in one place.

This list includes only arguments commonly used in Pylons-like applications. The full list is in View Configuration in the Pyramid manual. The arguments have the same predicate/non-predicate distinction as `add_route` arguments. It's possible to register multiple views for a route, each with different predicate arguments, to invoke a different view in different circumstances.

Some of the arguments are common to `add_route` and `add_view`. In the route's case it determines whether the route will match a URL. In the view's case it determines whether the view will match the route.

route_name

[predicate] The route to attach this view to. Required when using URL dispatch.

renderer

[non-predicate] The name of a renderer or template. Discussed in Renderers below.

permission

[non-predicate] A string naming a permission that the current user must have in order to invoke the view.

http_cache

[non-predicate] Affects the 'Expires' and 'Cache-Control' HTTP headers in the response. This tells the browser whether to cache the response and for how long. The value may be an integer specifying the number of seconds to cache, a `datetime.timedelta` instance, or zero to prevent caching. This is equivalent to calling `request.response.cache_expires(value)` within the view code.

context

[predicate] This view will be chosen only if the *context* is an instance of this class or implements this interface. This is used with traversal, authorization, and exception views.

request_method

[predicate] One of the strings "GET", "POST", "PUT", "DELETE', "HEAD". The request method must equal this in order for the view to be chosen. REST applications often register multiple views for the same route, each with a different request method.

request_param

[predicate] This can be a string such as "foo", indicating that the request must have a query parameter or POST variable named "foo" in order for this view to be chosen. Alternatively, if the string contains "=" such as "foo=1", the request must both have this parameter *and* its value must be as specified, or this view won't be chosen.

match_param

[predicate] Like request_param but refers to a routing variable in the matchdict. In addition to the "foo" and "foo=1" syntax, you can also pass a dict of key/value pairs: all these routing variables must be present and have the specified values.

xhr, accept, header, path_info

[predicate] These work like the corresponding arguments to `config.add_route`.

custom_predicates

[predicate] The value is a list of functions. Each function should take a `context` and `request` argument, and return true or false whether the arguments are acceptable to the view. The view will be chosen only if all functions return true. Note that the function arguments are different than the corresponding option to `config.add_route`.

One view option you will *not* use with URL dispatch is the "name" argument. This is used only in traversal.

Renderers

A *renderer* is a post-processor for a view. It converts the view's return value into a Response. This allows the view to avoid repetitive boilerplate code. Pyramid ships with the following renderers: Mako, Chameleon, String, JSON, and JSONP. The Mako and Chameleon renderers takes a dict, invoke the specified template on it, and return a Response. The String renderer converts any type to a string. The JSON and JSONP renderers convert any type to JSON or JSONP. (They use Python's `json` serializer, which accepts a limited variety of types.)

The non-template renderers have a constant name: `renderer="string"`, `renderer="json"`, `renderer="jsonp"`. The template renderers are invoked by a template's filename extension, so `renderer="mytemplate.mako"` and `renderer="mytemplate.mak"` go to Mako. Note that you'll need to specify a Mako search path in the INI file or main function:

```
[app:main]
mako.directories = my_app_package:templates
```

Supposedly you can pass an asset spec rather than a relative path for the Mako renderer, and thus avoid defining a Mako search path, but I couldn't get it to work. Chameleon templates end in .pt and must be specified as an asset spec.

You can register third-party renderers for other template engines, and you can also re-register a renderer under a different filename extension. The Akhet demo has an example of making pyramid send templates ending in .html through Mako.

You can also invoke a renderer inside view code.

```python
from pyramid.renderers import render, render_to_response

variables = {"dear": "Mr A", "sincerely": "Miss Z",
    "date": datetime.date.today()}

# Render a template to a string.
letter = render("form_letter.mako", variables, request=self.request)

# Render a template to a Response object.
return render_to_response("mytemplate.mako", variables,
    request=self.request)
```

Debugging views

If you're having trouble with a route or view not being chosen when you think it should be, try setting "pyramid.debug_notfound" and/or "pyramid.debug_routematch" to true in *development.ini*. It will log its reasoning to the console.

Multiple views using the same callable

You can stack multiple `@view_config` onto the same view method or function, in cases where the templates differ but the view logic is the same.

```
@view_config(route_name="help", renderer="help.mak")
@view_config(route_name="faq", renderer="faq.mak")
@view_config(route_name="privacy", renderer="privacy_policy.mak")
def template(request):
    return {}

@view_config(route_name="info", renderer="info.mak")
@view-config(route_name="info_json", renderer="json")
def info(request):
    return {}
```

1.9.7 Route and View Examples

Here are the most common kinds of routes and views.

1. Fixed controller and action.

```
# Pylons
map.connect("faq", "/help/faq", controller="help", action="faq")
class HelpController(Base):
    def faq(self):
        ...
```

```
# Pyramid
config.add_route("faq", "/help/faq")
@view_config(route_name="faq", renderer="...")
def faq(self):    # In some arbitrary class.
    ...
```

.

2. Fixed controller and action, plus other routing variables.

```
# Pylons
map.connect("article", "/article/{id}", controller="foo",
    action="article")
class FooController(Base):
    def article(self, id):
        ...
```

```
# Pyramid
config.add_route("article", "/article/{id}")
@view_config(route_name="article")
def article(self):    # In some arbitrary class.
    id = self.request.matchdict["id"]
```

.

3. Variable controller and action.

```
# Pylons
map.connect("/{controller}/{action}")
map.connect("/{controller/{action}/{id}")
```

```
# Pyramid
# Not possible.
```

You can't choose a view class via a routing variable in Pyramid.

4. Fixed controller, variable action.

```
# Pylons
map.connect("help", "/help/{action}", controller="help")
```

```
# Pyramid
config.add_route("help", "/help/{action}")

@view_config(route_name="help", match_param="action=help", ...)
def help(self):    # In some arbitrary class.
    ...
```

The 'pyramid_handlers' package provides an alternative for this.

Other Pyramid examples:

```
# Home route.
config.add_route("home", "/")

# Multi-action route, excluding certain static URLs.
config.add_route("main", "/{action}",
    path_info=r"/(?!favicon\.ico|robots\.txt|w3c)")
```

pyramid_handlers

"pyramid_handlers" is an add-on package that provides a possibly more convenient way to handle case #4 above, a route with an 'action' variable naming a view. It works like this:

```
# In the top-level __init__.py
from .handlers import Hello
def main(global_config, **settings):
    ...
    config.include("pyramid_handlers")
    config.add_handler("hello", "/hello/{action}", handler=Hello)

# In zzz/handlers.py
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 {}
```

The `add_handler` method (line 6) registers the route and then scans the Hello class. It registers as views all methods that have an `@action` decorator, using the method name as a view predicate, so that when that method name appears in the 'action' part of the URL, Pyramid calls this view.

The `__autoexpose__` class attribute (line 11) can be a regex. If any method name matches it, it will be registered as a view even if it doesn't have an `@action` decorator. The default autoexpose regex matches all methods that begin with a letter, so you'll have to set it to None to prevent methods from being automatically exposed. You can do this in a base class if you wish.

Note that `@action` decorators are *not* recognized by `config.scan()`. They work only with `config.add_hander`.

User reaction to "pyramid_handlers" has been mixed. A few people are using it, but most people use `@view_config` because it's "standard Pyramid".

Resouce routes

"pyramid_routehelper" provides a `config.add_resource` method that behaves like Pylons' `map.resource`. It adds a suite of routes to list/view/add/modify/delete a resource in a RESTful manner (following the Atom publishing protocol). See the source docstrings in the link for details.

Note: the word "resource" here is *not* related to traversal resources.

1.9.8 Request and Response

Pylons magic globals

Pylons has several magic globals that contain state data for the current request. Here are the closest Pyramid equivalents:

pylons.request

> The request URL, query parameters, etc. In Pyramid it's the `request` argument to view functions and `self.request` in view methods (if your class constructor follows the normal pattern). In templates it's `request` or `req` (starting in Pyramid 1.3). In pshell or unit tests where you can't get it any other way, use `request = pyramid.threadlocal.get_current_request()`.

pylons.response

> The HTTP response status and document. Pyramid does not have a global response object. Instead, your view should create a `pyramid.response.Response` instance and return it. If you're using a renderer, it will create a response object for you.
>
> For convenience, there's a `request.response` object available which you can set attributes on and return, but it will have effect only if you return it. If you're using a renderer, it will honor changes you make to `request.response`.

pylons.session

> Session variables. See the Sessions chapter.

pylons.tmpl_context

A scratch object for request-local data, usually used to pass varables to the template. In Pyramid, you return a dict of variables and let the renderer apply them to a template. Or you can render a template yourself in view code.

If the view is a method, you can also set instance variables. The view instance is visible as `view` in templates. There are two main use cses for this. One, to set variables for the site template that would otherwise have to be in *every* return dict. Two, for variables that are specific to HTML rendering, when the view is registered with both an HTML renderer and a non-HTML renderer (e.g., JSON).

Pyramid does have a port of "tmpl_context" at `request.tmpl_context`, which is visible in templates as `c`. However, it never caught on among Pyramid-Pylons users and is no longer documented.

pylons.app_globals

Global variables shared across all requests. The nearest equivalent is `request.registry.settings`. This normally contains the application settings, but you can also store other things in it too. (The registery is a singleton used internally by Pyramid.)

pylons.cache

A cache object, used to automatically save the results of expensive calculations for a period of time, across multiple requests. Pyramid has no built-in equivalent, but you can set up a cache using "pyramid_beaker". You'll probably want to put the cache in the settings?

pylons.url

A URL generator. Pyramid's request object has methods that generate URLs. See also the URL Generator chapter for a convenience object that reduces boilerplate code.

Request and response API

Pylons uses WebOb's request and response objects. Pyramid uses subclasses of these so all the familiar attributes and methods are there: `params`, `GET`, `POST`, `headers`, `method`, `charset`, `date`, `environ`, `body`, and `body_file`. The most commonly-used attribute is `params`, which is the query parameters and POST variables.

Pyramid adds several attributes and methods. `context`, `matchdict`, `matched_route`, `registry`, `registry.settings`, `session`, and `tmpl_context` access the request's state data and global application data. `route_path`, `route_url`, `resource_url`, and `static_url` generate URLs.

Rather than repeating the existing documentation for these attributes and methods, we'll just refer you to the original docs:

- Pyramd Request and Response Objects
- Pyramid Request API
- Pyramid Response API
- WebOb Request API
- WebOb Response API

Response examples:

```python
response = request.response

# -OR-
from pyramid.response import Response
response = Response

# In either case.
response.status = "200 OK"
response.status_int = 200
response.content_type = "text/plain"
response.charset = "utf-8"
response_headerlist = [
    ("Set-Cookie", "abc=123"), ("X-My-Header", "foo")]
response_cache_for = 3600    # Seconds
return response
```

1.9.9 Templates

Pyramid includes adapters for two template engines, Mako and Chameleon. Mako is Pylons' default engine so it will be familiar. Third-party adapters are available for other engines: "pyramid_jinja2" (a Jinja2 adapter), "pyramid_chameleon_gensi" (a partial Genshi emulator), etc.

Mako configuration

In order to use Mako as in Pylons, you must specify a template search path in the settings:

```
[app:main]
...
mako.directories = pyramidapp:templates
```

This enables relative template paths like `renderer="/mytemplate.mak"` and quasi-URL paths like `renderer="/mytemplate.mak"`. It also allows templates to inherit from other templates, import other templates, and include other templates. Without this setting, the renderer arg will have to be in asset spec syntax, and templates won't be able to invoke other templates.

All settings with the "mako." prefix are passed to Mako's `TemplateLookup` constructor. E.g.,

```
mako.strict_undefined = true
mako.imports =
     from mypackage import myfilter
mako.filters = myfilter
mako.module_directory = %(here)s/data/templates
mako.preprocessor = mypackage.mako_preprocessor
```

Template filenames ending in ".mak" or ".mako" are sent to the Mako renderer. If you prefer a different extension such as ".html", you can put this in your main function:

```
config.add_renderer(".html", "pyramid.mako_templating.renderer_factory")
```

If you have further questions about exactly how the Mako renderer is implemented, it's best to look at the source: `pyramid.mako_templating`. You can reconcile that with the Mako documentation to confirm what argument values cause what.

Caution: When I set "mako.strict_undefined" to true in an application that didn't have Beacon sessons configured, it broke the debug toolbar. The toolbar templates may have some sloppy placeholders not guarded by "% if".

Caution 2: Supposedly you can pass an asset spec instead of a template path but I couldn't get it to work.

Chameleon

Chameleon is an XML-based template language descended from Zope. It has some similarities with Genshi. Its filename extension is .pt ("page template").

Advantages of Chameleon:

- XML-based syntax.
- Template must be well-formed XHTML, suggesting (but not guaranteeing) that the output will be well-formed. If any variable placeholder is marked "structure", it's possible to insert invalid XML into the template.
- Good internationalization support in Pyramid.
- Speed is as fast as Mako. (Unusual for XML template languages.)
- Placeholder syntax "${varname or expression}" is common to Chameleon, Mako, and Genshi.
- Chameleon does have a text mode which accepts non-XML input, but you lose all control structures except "${varname}".

Disadvantages of Chameleon:

- XML-based syntax.
- Filenames must be in asset spec syntax, not relative paths: `renderer="mypackage:templates/foo.pt"`, `renderer="templates/foo.pt"`. You can't get rid of that "templates/" prefix without writing a wrapper `view_config` decorator.
- No template lookup, so you can't invoke one template from inside another without pre-loading the template into a variable.
- If template is not well-formed XML, the user will get an unconditional "Internal Server Error" rather than something that might look fine in the browser and which the user can at least read some content from.
- It doesn't work on all platforms Mako and Pyramid do. (Only CPython and Google App Engine.)

Renderer globals

Whenever a renderer invokes a template, the template namespace includes all the variables in the view's return dict, plus the following system variables:

`request, req`
  The current request.

`view`
  The view instance (for class-based views) or function (for function-based views). You can read instance attributes directly: `view.foo`.

`context`
  The context (same as `request.context`). (Not visible in Mako because Mako has a built-in variable with this name; use `request.context` instead.)

`renderer_name`
  The fully-qualified renderer name; e.g., "zzz:templates/foo.mako".

`renderer_info`
  An object with attributes `name`, `package`, and `type`.

The Akhet demo shows how to inject other variables into all templates, such as a helpers module `h`, a URL generator `url`, the session variable `session`, etc.

Site template

Most sites will use a site template combined with page templates to ensure that all the pages have the same look and feel (header, sidebars, and footer). Mako's inheritance makes it easy to make page templates inherit from a site template. Here's a very simple site template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Application</title>
  </head>
  <body>

<!-- *** BEGIN page content *** -->
${self.body()}
<!-- *** END page content ***-->

  </body>
</html>
```

... and a page template that uses it:

```
<%inherit file="/site.html" />

<p>
  Welcome to <strong>${project}</strong>, an application ...
</p>
```

A more elaborate example is in the Akhet demo.

1.9.10 Exceptions, HTTP Errors, and Redirects

Issuing redirects and HTTP errors

Here's how to send redirects and HTTP errors in Pyramid compared to Pylons:

```
# Pylons -- in controller action
from pylons.controllers.util import abort, redirect
abort(404)    # Not Found
abort(403)    # Forbidden
abort(400)    # Bad request; e.g., invalid query parameter
abort(500)    # Internal server error
redirect(url("section1"))    # Redirect (default 302 Found)

# Pyramid -- in view code
import pyramid.httpexceptions as exc
raise exc.exception_response(400)    # Not Found
raise exc.HTTPNotFound()             # Same thing
return exc.HTTPNotFound()            # Same thing
raise exc.HTTPForbidden()
raise exc.HTTPBadRequest()
raise exc.HTTPInternalServerError()
raise exc.HTTPFound(request.route_url("section1"))    # Redirect
```

The `pyramid.httpexceptions` module has classes for all official HTTP statuses. These classes inherit from both `Response` and `Exception`, so you can either return them or raise them. Raising HTTP exceptions can make your code structurally more readable. It's particularly useful in subroutines where it can cut through several calling stack frames that would otherwise each need an `if` to pass the error condition through.

Exception rules:

1. Pyramid internally raises `HTTPNotFound` if no route matches the request, or if no view matches the route and request. It raises `HTTPForbidden` if the request is denied based on the current authorization policy.

2. If an uncaught exception occurs during request processing, Pyramid will catch it and look for an "exception view" that matches it. An exception view is one whose *context* argument is the exception's class, an ancestor of it, or an interface it implements. All other view predicates must also match; e.g., if a 'route_name' argument is specified, it must match the actual route name. (Thus an exception view is typically registered *without* a route name.) The view is called with the exception object as its *context*, and whatever response the view returns will be sent to the browser. You can thus use an exception view to customize the error screen shown to the user.
3. If no matching exception view is found, HTTP exceptions are their own response so they are sent to the browser. Standard HTTPExceptions have a simple error message and layout; subclasses can customize this.
4. Non-HTTPException responses propagate to the WSGI server. If the debug toolbar tween is enabled, it will catch the exception and display the interactive traceback. Otherwise the WSGI server will catch it and send its own "500 Internal Server Error" screen.

Here are the most popular HTTP exceptions:

Class	Code	Location	Meaning
HTTPMovedPermanently	301	Y	Permanent redirect; client should change bookmarks.
HTTPFound	302	Y	Temporary redirect. [1]
HTTPSeeOther	303	Y	Temporary redirect; client should use GET. [1]
HTTPTemporaryRedirect	307	Y	Temporary redirect. [1]
HTTPClientError	400	N	General user error; e.g., invalid query param.
HTTPUnauthorized	401	N	User must authenticate.
HTTPForbidden	403	N	Authorization failure, or general refusal.
HTTPNotFound	404	N	The URL is not recognized.
HTTPGone	410	N	The resource formerly at this URL is permanently gone; client should delete bookmarks.
HTTPInternalServerError	500	N	The server could not process the request due to an internal error.

The constructor args for classes with a "Y" in the location column are `(location="", detail=None, headers=None, comment=None, ...)`. Otherwise the constructor args are `(detail=None, headers=None, comment=None, ...)`.

The `location` argument is optional at the Python level, but the HTTP spec requires a location that's an absolute URL, so it's effectively required.

The `detail` argument may be a plain-text string which will be incorporated into the error screen. `headers` may be a list of HTTP headers (name-value tuples) to add to the response. `comment` may be a plain-text string which is not shown to the user. (XXX Is it logged?)

[1] The three temporary redirect statuses are largely interchangeable but have slightly different purposes. Details in the HTTP status reference.

Exception views

You can register an exception view for any exception class, although it's most commonly used with `HTTPNotFound` or `HTTPForbidden`. Here's an example of an exception view with a custom exception, borrowed from the Pyramid manual:

```
from pyramid.response import Response

class ValidationFailure(Exception):
    pass

@view_config(context=ValidationFailure)
def failed_validation(exc, request):
    # If the view has two formal arguments, the first is the context.
    # The context is always available as ``request.context`` too.
    msg = exc.args[0] if exc.args else ""
    response =  Response('Failed validation: %s' % msg)
    response.status_int = 500
    return response
```

For convenience, Pyramid has special decorators and configurator methods to register a "Not Found" view or a "Forbidden" view. `@notfound_view_config` and `@forbidden_view_config` (defined in `pyramid.view`) takes care of the context argument for you.

Additionally, `@notfound_view_config` accepts an `append_slash` argument, which can be used to enforce a trailing-slash convention. If your site defines all its routes to end in a slash and you set `append_slash=True`, then when a slashless request doesn't match any route, Pyramid try again with a slash appended to the request URL. If *that* matches a route, Pyramid will issue a redirect to it. This is useful only for sites that prefer a trailing slash ("/dir/" and "/dir/a/"). Other sites prefer *not* to have a trailing slash ("/dir" and "/dir/a"), and there are no special features for this.

Reference

- HTTP exceptions
- HTTP exception usage and exception views

1.9.11 Static Files

In Pylons, the application's "public" directory is configured as a static overlay on "/", so that URL "/images/logo.png" goes to "pylonsapp/public/images/logo.png". This is done using a middleware. Pyramid does not have an exact equivalent but it does have a way to serve static files, and add-on packages provide additional ways.

Static view

This is Pyramid's default way to serve static files. As you'll remember from the main function in an earlier chapter:

```
config.add_static_view('static', 'static', cache_max_age=3600)
```

This tells Pyramid to publish the directory "pyramidapp/static" under URL "/static", so that URL "/static/images/logo.png" goes to "pyramidapp/static/images/logo.png".

It's implemented using *traversal*, which we haven't talked about much in this Guide. Traversal-based views have a *view name* which serves as a URL prefix or component. The first argument is the view name ("static"), which implies it matches URL "/static". The second argument is the asset spec for the directory (relative to the application's Python package). The keyword arg is an option which sets the HTTP expire headers to 3600 seconds (1 hour) in the future. There are other keyword args for permissions and such.

Pyramid's static view has the following advantages over Pylons:

- It encourages all static files to go under a single URL prefix, so they're not scattered around the URL space.
- Methods to generate URLs are provided: `request.static_url()` and `request.static_path()`.
- The deployment configuration (INI file) can override the base URL ("/static") to serve files from a separate static media server ("http://static.example.com/").
- The deployment configuration can also override items in the static directory, pointing to other subdirectories or files instead. This is called "overriding assets" in the Pyramid manual.

It has the following disadvantages compared to Pylons:

- Static URLs have the prefix "/static".
- It can't serve top-level file URLs such as "/robots.txt" and "/favicon.ico".

You can serve any URL directory with a static view, so you could have a separate view for each URL directory like this:

```
config.add_images_view('images', 'static/images')
config.add_stylesheets_view('stylesheets', 'static/stylesheets')
config.add_javascript_view('javascript', 'static/javascript')
```

This configures URL "/images" pointing to directory "pyramidapp/static/images", etc.

If you're using Pyramid's authorization system, you can also make a separate view for files that require a certain permission:

```
config.add_static_view("private", "private", permission="admin")
```

Generating static URLs

You can generate a URL to a static file like this:

```
href="${request.static_url('static/images/logo.png')}
```

Top-level file URLs

So how do you get around the problem of top-level file URLs? You can register normal views for them, as shown later below. For "/favicon.ico", you can replace it with an HTTP header in your site template:

```
<link rel="shortcut icon" href="${request.static_url('pyramidapp:static/favicon.ico')}" />
```

The standard Pyramid scaffolds actually do this. For "/robots.txt", you may decide that this actually belongs to the webserver rather than the application, and so you might have Apache serve it directly like this:

```
Alias   /robots.txt   /var/www/static/norobots.txt
```

You can of course have Apache serve your static directory too:

```
Alias   /static   /PATH-TO/PyramidApp/pyramidapp/static
```

But if you're using mod_proxy you'll have to disable proxying that directory *early* in the virtualhost configuration:

```
Alias ProxyPass   /static   !
```

If you're using RewriteRule in combination with other path directives like Alias, read the RewriteRule flags documentation (especially "PT" and "F") to ensure the directives cooperate as expected.

External static media server

To make your configuration flexible for a static media server:

```
# In INI file
static_assets = "static"
# -OR-
static_assets = "http://staticserver.com/"
```

Main function:

```
config.add_static_view(settings["static_assets"], "zzz:static")
```

Now it will generate "http://mysite.com/static/foo.jpg" or "http://staticserver.com/foo.jpg" depending on the configuration.

Static route

This strategy is available in Akhet. It overlays the static directory on top of "/" like Pylons does, so you don't have to change your URLs or worry about top-level file URLs.

```
config.include('akhet')
# Put your regular routes here.
config.add_static_route('zzz', 'static', cache_max_age=3600)
# Arg 1 is the Python package containing the static files.
# Arg 2 is the subdirectory in the package containing the files.
```

This registes a static route matching all URLs, and a view to serve it. Actually, the route will have a predicate that checks whether the file exists, and if it doesn't, the route won't match the URL. Still, it's good practice to register the static route after your other routes.

If you have another catchall route before it that might match some static URLs, you'll have to exclude those URLs from the route as in this example:

```
config.add_route("main", "/{action}",
    path_info=r"/(?!favicon\.ico|robots\.txt|w3c)")
config.add_static_route('zzz', 'static', cache_max_age=3600)
```

The static route implementation does *not* generate URLs to static files, so you'll have to do that on your own. Pylons never did it very effectively either.

Other ways to serve top-level file URLs

If you're using the static view and still need to serve top-level file URLs, there are several ways to do it.

A manual file view

This is documented in the Pyramid manual in the Static Assets chapter.

```
# Main function.
config.add_route("favicon", "/favicon.ico")

# Views module.
import os
from pyramid.response import FileResponse

@view_config(route_name="favicon")
def favicon_view(request):
    here = os.path.dirname(__file__)
    icon = os.path.join(here, "static", "favicon.ico")
    return FileResponse(icon, request=request)
```

Or if you're really curious how to configure the view for traversal without a route:

```
@view_config(name="favicon.ico")
```

pyramid_assetviews

"pyramid_assetviews" is a third-party package for top-level file URLs.

```
# In main function.
config.include("pyramid_assetviews")
config.add_asset_views("static", "robots.txt")  # Defines /robots.txt .

# Or to register multiple files at once.
filenames = ["robots.txt", "humans.txt", "favicon.ico"]
config.add_asset_views("static", filenames=filenames, http_cache=3600)
```

Of course, if you have the files in the static directory they'll still be visible as "/static/robots.txt" as well as "/robots.txt". If that bothers you, make another directory outside the static directory for them.

1.9.12 Sessions

Pyramid uses Beaker sessions just like Pylons, but they're not enabled by default. To use them you'll have to add the "pyramid_beaker" package as a dependency, and put the following line in your `main()` function:

```
config.include("pyramid_beaker")
```

(To add a dependency, put it in the `requires` list in setup.py, and reinstall the application.)

The default configuration is in-memory sessions and (I think) no caching. You can customize this by putting configuration settings in your INI file or in the `settings` dict at the beginning of the `main()` function (before the Configurator is instantiated). The Akhet Demo configures Beaker with the following settings, borrowed from the Pylons configuration:

```
# Beaker cache
cache.regions = default_term, second, short_term, long_term
cache.type = memory
cache.second.expire = 1
cache.short_term.expire = 60
cache.default_term.expire = 300
cache.long_term.expire = 3600

# Beaker sessions
#session.type = file
```

```
#session.data_dir = %(here)s/data/sessions/data
#session.lock_dir = %(here)s/data/sessions/lock
session.type = memory
session.key = akhet_demo
session.secret = 0cb243f53ad865a0f70099c0414ffe9cfcfe03ac
```

To use file-based sessions like in Pylons, uncomment the first three session settings and comment out the "session.type = memory" line.

You should set the "session.secret=" setting to a random string. It's used to digitally sign the session cookie to prevent session hijacking.

Beaker has several persistence backends available, including memory, files, SQLAlchemy, memcached, and cookies (which stores each session variable in a client-side cookie, and has size limitationss). The most popular deployment backend nowadays is memcached, which can act as a shared storage between several processes and servers, thus providing the speed of memory with the ability to scale to a multi-server cluster. Pylons defaults to disk-based sessions.

Beaker plugs into Pyramid's built-in session interface, which is accessed via `request.session`. Use it like a dict. Unlike raw Beaker sessions, you don't have to call `session.save()` every time you change something, but you should call `session.changed()` if you've modified a *mutable* item in the session; e.g., `session["mylist"].append(1)`.

The Pyramid session interface also has some extra features. It can store a set of "flash messages" to display on the next page view, which is useful when you want to push a success/failure message and redirect, and the message will be displayed on the target page. It's based on `webhelpers.flash`, which is incompatible with Pyramid because it depends on Pylons' magic globals. There are also methods to set a secure form token, which prevent form submissions that didn't come from a form requested earlier in the session (and thus may be a cross-site forgery attack). (Note: flash messages are not related to the Adobe Flash movie player.)

See the Sessions chapter in the Pyramid manual for the API of all these features and other features. The Beaker manual will help you configure a backend. The Akhet Demo is an example of using Pyramid with Beaker, and has flash messages.

Note: I sometimes get an exception in the debug toolbar when sessions are enabled. They may be a code discrepency between the distributions. If this happens to you, you can disable the toolbar until the problem is fixed.

1.9.13 Deployment

Deployment is the same for Pyramid as for Pylons. Specify the desired WSGI server in the "[server:main]" and run "pserve" with it. The default server in Pyramid is Waitress, compared to PasteHTTPServer in Pylons.

Waitress' advantage is that it runs on Python 3. Its disadvantage is that it doesn't seek and destroy stuck threads like PasteHTTPServer does. If you're like me, that's enough reason not to use Waitress in production. You can switch to PasteHTTPServer or CherryPy server if you wish, or use a method like mod_wsgi that doesn't require a Python HTTP server.

1.9.14 Authentication and Authorization

This chapter is contributed by Eric Rasmussen.

Pyramid has built-in authentication and authorization capibalities that make it easy to restrict handler actions. Here is an overview of the steps you'll generally need to take:

1. Create a root factory in your model that associates allow/deny directives with groups and permissions
2. Create users and groups in your model
3. Create a callback function to retrieve a list of groups a user is subscribed to based on their user ID

4. Make a "forbidden view" that will be invoked when a Forbidden exception is raised.
5. Create a login action that will check the username/password and remember the user if successful
6. Restrict access to handler actions by passing in a permission='somepermission' argument to `@view_config`.
7. Wire it all together in your config

You can get started by adding an import statement and custom root factory to your model:

```python
from pyramid.security import Allow, Everyone

class RootFactory(object):
    __acl__ = [ (Allow, Everyone, "everybody"),
                (Allow, "basic", "entry"),
                (Allow, "secured", ("entry", "topsecret"))
              ]
    def __init__(self, request):
        pass
```

The custom root factory generates objects that will be used as the context of requests sent to your web application. The first attribute of the root factory is the ACL, or access control list. It's a list of tuples that contain a directive to handle the request (such as Allow or Deny), the group that is granted or denied access to the resource, and a permission (or optionally a tuple of permissions) to be associated with that group.

The example access control list above indicates that we will allow everyone to view pages with the 'everybody' permission, members of the basic group to view pages restricted with the 'entry' permission, and members of the secured group to view pages restricted with either the 'entry' or 'topsecret' permissions. The special principal 'Everyone' is a built-in feature that allows any person visiting your site (known as a principal) access to a given resource.

For a user to login, you can create a handler that validates the login and password (or any additional criteria) submitted through a form. You'll typically want to add the following imports:

```python
from pyramid.httpexceptions import HTTPFound
from pyramid.security import remember, forget
```

Once you validate a user's login and password against the model, you can set the headers to "remember" the user's ID, and then you can redirect the user to the home page or url they were trying to access:

```python
# retrieve the userid from the model on valid login
headers = remember(self.request, userid)
return HTTPFound(location=someurl, headers=headers)
```

Note that in the call to the remember function, we're passing in the user ID we retrieved from the database and stored in the variable 'userid' (an arbitrary name used here as an example). However, you could just as easily pass in a username or other unique identifier. Whatever you decide to "remember" is what will be passed to the groupfinder callback function that returns a list of groups a user belongs to. If you import `authenticated_userid`, which is a useful way to retrieve user information in a handler action, it will return the information you set the headers to "remember".

To log a user out, you "forget" them, and use HTTPFound to redirect to another url:

```python
headers = forget(self.request)
return HTTPFound(location=someurl, headers=headers)
```

Before you restrict a handler action with a permission, you will need a callback function to return a list of groups that a user ID belongs to. Here is one way to implement it in your model, in this case assuming you have a Groups object with a groupname attribute and a Users object with a mygroups relation to Groups:

```
def groupfinder(userid, request):
    user = Users.by_id(userid)
    return [g.groupname for g in user.mygroups]
```

As an example, you could now import and use the @action decorator to restrict by permission, and authenticated_userid to retrieve the user's ID from the request:

```
from pyramid_handlers import action
from pyramid.security import authenticated_userid
from models import Users

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

    @action(renderer="welcome.html", permission="entry")
    def index(self):
        userid = authenticated_userid(self.request)
        user = Users.by_id(userid)
        username = user.username
        return {"currentuser": username}
```

This gives us a very simple way to restrict handler actions and also obtain information about the user. This example assumes we have a Users class with a convenience class method called by_id to return the user object. You can then access any of the object's attributes defined in your model (such as username, email address, etc.), and pass those to a template as dictionary key/values in your return statement.

If you would like a specific handler action to be called when a forbidden exception is raised, you need to add a forbidden view. This was covered earlier, but for completelness:

```
@view_config(renderer='myapp:templates/forbidden.html',
             context='pyramid.exceptions.Forbidden')
@action(renderer='forbidden.html')
def forbidden(request):
    ...
```

The last step is to configure __init__.py to use your auth policy. Make sure to add these imports:

```
from pyramid.authentication import AuthTktAuthenticationPolicy
from pyramid.authorization import ACLAuthorizationPolicy
from .models import groupfinder
```

In your main function you'll want to define your auth policies so you can include them in the call to Configurator:

```
authn_policy = AuthTktAuthenticationPolicy('secretstring',
   callback=groupfinder)
authz_policy = ACLAuthorizationPolicy()
config = Configurator(settings=settings,
   root_factory='myapp.models.RootFactory',
   authentication_policy=authn_policy,
   authorization_policy=authz_policy)
config.scan()
```

The capabilities for authentication and authorization in Pyramid are very easy to get started with compared to using Pylons and repoze.what. The advantage is easier to maintain code and built-in methods to handle common tasks like remembering or forgetting users, setting permissions, and easily modifying the groupfinder callback to work with your model. For cases where it's manageable to set permissions in advance in your root factory and restrict individual handler actions, this is by far the simplest way to get up and running while still offering robust user and group management capabilities through your model.

However, if your application requires the ability to create/edit/delete permissions (not just access through group membership), or you require the use of advanced predicates, you can either build your own auth system (see the Pyramid docs for details) or integrate an existing system like repoze.what.

You can also use "repoze.who" with Pyramid's authorization system if you want to use Who's authenticators and configuration.

1.9.15 Other Pyramid Features

Shell

Pyramid has a command to preload your application into an interactive Python prompt. This can be useful for debugging or experimentation. The command is "pshell", akin to "paster shell" in Pylons.

```
$ pshell development.ini
Python 2.6.5 (r265:79063, Apr 29 2010, 00:31:32)
[GCC 4.4.3] on linux2
Type "help" for more information.

Environment:
  app          The WSGI application.
  registry     Active Pyramid registry.
  request      Active request object.
  root         Root of the default resource tree.
  root_factory Default root factory used to create `root`.

>>>
```

It doesn't initialize quite as many globals as Pylons, but `app` and `request` will be the most useful.

Other commands

Other commands available:

- proutes: list the application's routes. (Akin to Pylons "paster routes".)
- pviews: list the application's views.
- ptweens: list the application's tweens.
- prequest: load the application, process a specified URL, and print the response body on standard output.

Forms

Pyramid does not include a form library. Pylons includes WebHelpers for form generation and FormEncode for validation and error messages. These work under Pyramid too. However, there's no built-in equivalent to Pylons' `@validate` decorator. Instead we recommend the "pyramid_simpleform" package, which replaces @validate with a more flexible structure.

There are several other form libraries people use with Pyramid. These are discussed in the regular Forms section in the Pyramid Cookbook.

WebHelpers

WebHelpers is a third-party package containing HTML tag builders, text functions, number formatting and statistical functions, and other generic functions useful in templates and views. It's a Pylons dependency but is optional in Pyramid.

The `webhelpers.pylonslib` subpackage does not work with Pyramid because it depends on Pylons' special globals. `webhelpers.mimehelper` and `webhelpers.paginate` have Pylons-specific features that are disabled under other frameworks. WebHelpers has not been tested on Python 3.

The next version of WebHelpers may be released as a different distribution (WebHelpers2) with a subset of the current helpers ported to Python 3. It will probably spin off Paginate and the Feed Generator to separate distributions.

Events

The events framework provides hooks where you can insert your own code into the request-processing sequence, similar to how Apache modules work. It standarizes some customizations that were provided ad-hoc in Pylons or not at all. To use it, write a callback function for one of the event types in `pyramid.events`: `ApplicationCreated`, `ContextFound`, `NewResponse`, `BeforeRender`. The callback takes an event argument which is specific to the event type. You can register the event with `@asubscriber` or `config.add_subscriber()`. The Akhet demo has examples.

For more details see:

- Using Events *
- Using The Before Render Event
- pyramid.event API

URL generation

Pyramid does not come with a URL generator equivalent to "pylons.url". Individual methods are available on the Request object to generate specific kinds of URLs. Of these, route_url covers the normal case of generating a route by name:

```
request.route_url("route_name", variable1="value1")
request.route_path("route_name", variable1="value1")
request.route_url("search", _query={"q": "search term"}, _anchor="results")
```

As with all the *_url vs *_path methods, `route_url` generates an absolute URL, while `route_path` generates a "slash" URL (without the scheme or host). The `_query` argument is a dict of query parameters (or a sequence of key-value pairs). The `_anchor` argument makes a URL with a "#results" fragment. Other special keyword arguments are `_scheme`, `_host`, `_port`, and `_app_url`.

The advantage of using these methods rather than hardcoding the URL, is that it automatically addds the application prefix (which may be something more than "/" if the application is mounted on a sub-URL).

You can also pass additional positional arguments, and they will be appended to the URL as components. This is not very useful with URL dispatch, it's more of a traversal thing.

If the route is defined with a *pregenerator*, it will be called with the positional and keyword arguments, and can modify them before the URL is generated.

Akhet has a URLGenerator class, which you can use as shown in the Akhet demo to make a `url` variable for your templates, using an event subscriber. Then you can do things like this:

```
url.route("route_name")           # Generate URL by route name.
url("route_name")                 # The same.
url.app                           # The application's top-level URL.
url.current()                     # The current request URL. (Used to
                                  # link to the same URL with different
                                  # match variables or query params.)
```

You can also customize it to do things like this:

```
url.static("images/logo.png")
url.image("logo.png")             # Serve an image from the images dir.
url.deform("...")                 # Static file in the Deform package.
```

If “url” is too long for you, you can even name it “u”!

Utility scripts

Pyramid has a documented way to write utility scripts for maintenance and the like. See Writing a Script.

Testing

Pyramid makes it easier to write unit tests for your views.

(XXX Need a comparison example.)

Internationalization

Pyramid has support for internationalization. At this time it’s documented mainly for Chameleon templates, not Mako.

Higher-level frameworks

Pyramid provides a flexible foundation to build higher-level frameworks on. Several have already been written. There are also application scaffolds and tarballs.

- Kotti is a content management system that both works out of the box and can be extended.
- Ptah is a framework that aims to have as many features as Django. (But no ponies, and no cowbells.) It has a minimal CMS component.
- Khufu is a suite of scaffolds and utilities for Pyramid.
- The Akhet demo we have mentioned before. It’s a working application in a tarball that you can copy code from.

At the opposite extreme, you can make a tiny Pyramid application in 14 lines of Python without a scaffold. The Pyramid manual has an example: Hello World. This is not possible with Pylons – at least, not without distorting it severely.

1.9.16 Migrating an Existing Pylons Application

There are two general ways to port a Pylons application to Pyramid. One is to start from scratch, expressing the application’s behavior in Pyramid. Many aspects such as the models, templates, and static files can be used unchanged or mostly unchanged. Other aspects like such as the controllers and globals will have to be rewritten. The route map can be ported to the new syntax, or you can take the opportunity to restructure your routes.

The other way is to port one URL at a time, and let Pyramid serve the ported URLs and Pylons serve the unported URLs. There are several ways to do this:

- Run both the Pyramid and Python applications in Apache, and use mod_rewrite to send different URLs to different applications.
- Set up `paste.cascade` in the INI file, so that it will first try one application and then the other if the URL returns "Not Found". (This is how Pylons serves static files.)
- Wrap the Pylons application in a Pyramid view. See pyramid.wsgiapp.wsgiapp2.

Also see the Porting Applications to Pyramid section in the Cookbook.

Caution: running a Pyramid and a Pylons application simultaneously may bring up some tricky issues such as coordiating database connections, sessions, data files, etc. These are beyond the scope of this Guide.

You'll also have to choose whether to write the Pyramid application in Python 2 or 3. Pyramid 1.3 runs on Python 3, along with Mako and SQLAlchemy, and the Waitress and CherryPy HTTP servers (but not PasteHTTPServer). But not all optional libraries have been ported yet, and your application may depend on libraries which haven't been.

1.10 Routing: Traversal and URL Dispatch

1.10.1 Comparing and Combining Traversal and URL Dispatch

(adapted from Bayle Shank's contribution at https://github.com/bshanks/pyramid/commit/c73b462c9671b5f2c3be26cf088ee983952ab61

Here's is an example which compares URL dispatch to traversal.

Let's say we want to map

`/hello/login` to a function `login` in the file `myapp/views.py`

`/hello/foo` to a function `foo` in the file `myapp/views.py`

`/hello/listDirectory` to a function `listHelloDirectory` in the file `myapp/views.py`

`/hello/subdir/listDirectory` to a function `listSubDirectory` in the file `myapp/views.py`

With URL dispatch, we might have:

```
config.add_route('helloLogin', '/hello/login')
config.add_route('helloFoo', '/hello/foo')
config.add_route('helloList', '/hello/listDirectory')
config.add_route('list', '/hello/{subdir}/listDirectory')

config.add_view('myapp.views.login', route_name='helloLogin')
config.add_view('myapp.views.foo', route_name='helloFoo')
config.add_view('myapp.views.listHelloDirectory', route_name='helloList')
config.add_view('myapp.views.listSubDirectory', route_name='list')
```

When the listSubDirectory function from `myapp/views.py` is called, it can tell what the subdirectory's name was by checking `request.matchdict['subdir']`. This is about all you need to know for URL-dispatch-based apps.

With traversal, we have a more complex setup:

```
class MyResource(dict):
    def __init__(self, name, parent):
        self.__name__ = name
        self.__parent__ = parent

```

```python
class MySubdirResource(MyResource):
    def __init__(self, name, parent):
        self.__name__ = name
        self.__parent__ = parent

    # returns a MyResource object when the key is the name
    # of a subdirectory
    def __getitem__(self, key):
        return MySubdirResource(key, self)

class MyHelloResource(MySubdirResource):
    pass

def myRootFactory(request):
    rootResource = MyResource('', None)
    helloResource = MyHelloResource('hello', rootResource)
    rootResource['hello'] = helloResource
    return rootResource

config.add_view('myapp.views.login', name='login')
config.add_view('myapp.views.foo', name='foo')
config.add_view('myapp.views.listHelloDirectory', context=MyHelloResource,
                name='listDirectory')
config.add_view('myapp.views.listSubDirectory', name='listDirectory')
```

In the traversal example, when a request for `/hello/@@login` comes in, the framework calls `myRootFactory(request)`, and gets back the root resource. It calls the MyResource instance's `__getitem__('hello')`, and gets back a `MyHelloResource`. We don't traverse the next path segment `(@@login`), because the ``@@` means the text that follows it is an explicit view name, and traversal ends. The view name 'login' is mapped to the `login` function in `myapp/views.py`, so this view callable is invoked.

When a request for `/hello/@@foo` comes in, a similar thing happens.

When a request for `/hello/@@listDirectory` comes in, the framework calls `myRootFactory(request)`, and gets back the root resource. It calls MyRootResource's `__getitem__('hello')`, and gets back a `MyHelloResource` instance. It *does not* call MyHelloResource's `__getitem__('listDirectory')` (due to the `@@` at the lead of `listDirectory`). Instead, 'listDirectory' becomes the view name and traversal ends. The view name 'listDirectory' is mapped to `myapp.views.listRootDirectory`, because the context (the last resource traversed) is an instance of `MyHelloResource`.

When a request for `/hello/xyz/@@listDirectory` comes in, the framework calls `myRootFactory(request)`, and gets back an instance of `MyRootResource`. It calls MyRootResource's `__getitem__('hello')`, and gets back a `MyHelloResource` instance. It calls MyHelloResource's `__getitem__('xyz')`, and gets back another `MySubdirResource` instance. It *does not* call `__getitem__('listDirectory')` on the `MySubdirResource` instance. 'listDirectory' becomes the view name and traversal ends. The view name 'listDirectory' is mapped to `myapp.views.listSubDirectory`, because the context (the final traversed resource object) is not an instance of `MyHelloResource`. The view can access the `MySubdirResource` via `request.context`.

At we see, traversal is more complicated than URL dispatch. What's the benefit? Well, consider the URL `/hello/xyz/abc/listDirectory`. This is handled by the above traversal code, but the above URL dispatch code would have to be modified to describe another layer of subdirectories. That is, traversal can handle arbitrarily deep, dynamic hierarchies in a general way, and URL dispatch can't.

You can, if you want to, combine URL dispatch and traversal (in that order). So, we could rewrite the above as:

```python
class MyResource(dict):
    def __init__(self, name, parent):
        self.__name__ = name
        self.__parent__ = parent

    # returns a MyResource object unconditionally
    def __getitem__(self, key):
        return MyResource(key, self)

def myRootFactory(request):
    return MyResource('', None)

config = Configurator()

config.add_route('helloLogin', '/hello/login')
config.add_route('helloFoo', '/hello/foo')
config.add_route('helloList', '/hello/listDirectory')
config.add_route('list', '/hello/*traverse', factory=myRootFactory)

config.add_view('myapp.views.login', route_name='helloLogin')
config.add_view('myapp.views.foo', route_name='helloFoo')
config.add_view('myapp.views.listHelloDirectory', route_name='helloList')
config.add_view('myapp.views.listSubDirectory', route_name='list',
                name='listDirectory')
```

You will be able to visit e.g. http://localhost:8080/hello/foo/bar/@@listDirectory to see the listSubDirectory view.

This is simpler and more readable because we are using URL dispatch to take care of the hardcoded URLs at the top of the tree, and we are using traversal only for the arbitrarily nested subdirectories.

See Also

- *Using Traversal in Pyramid Views*

1.10.2 Using Traversal in Pyramid Views

A trivial example of how to use traversal in your view code.

You may remember that a Pyramid view is called with a context argument:

```
def my_view(context, request):
    return render_view_to_response(context, request)
```

When using traversal, `context` will be the resource object that was found by traversal. Configuring which resources a view responds to can be done easily via either the `@view.config` decorator...

```
from models import MyResource

@view_config(context=MyResource)
def my_view(context, request):
    return render_view_to_response(context, request)
```

or via `config.add_view`:

```
from models import MyResource
config = Configurator()
config.add_view('myapp.views.my_view', context=MyResource)
```

Either way, any request that triggers traversal and traverses to a `MyResource` instance will result in calling this view with that instance as the `context` argument.

Optional: Using Interfaces

If your resource classes implement interfaces, you can configure your views by interface. This is one way to decouple view code from a specific resource implementation:

```python
# models.py
from zope.interface import implements
from zope.interface import Interface

class IMyResource(Interface):
    pass

class MyResource(object):
    implements(IMyResource)

# views.py
from models import IMyResource

@view_config(context=IMyResource)
def my_view(context, request):
    return render_view_to_response(context, request)
```

See Also

- Much Ado About Traversal
- *Comparing and Combining Traversal and URL Dispatch*
- The "Virginia" sample application: https://github.com/Pylons/virginia/blob/master/virginia/views.py
- ZODB and Traversal in Pyramid tutorial: http://docs.pylonsproject.org/projects/pyramid/en/latest/tutorials/wiki/index.html#bfg-wiki-tutorial
- "Pyramid Traversal and Mongodb": http://kusut.web.id/2011/03/27/pyramid-traversal-and-mongodb/
- Resources which implement interfaces: http://readthedocs.org/docs/pyramid/en/latest/narr/resources.html#resources-which-implement-interfaces

1.10.3 Traversal with SQLAlchemy

This is a stub page, written by a non-expert. If you have expertise, please verify the content, add recipes, and consider writing a tutorial on this.

Traversal works most naturally with an object database like ZODB because both are naturally recursive. (I.e., "/a/b" maps naturally to `root["a"]["b"]`.) SQL tables are flat, not recursive. However, it's possible to use traversal with SQLAlchemy, and it's becoming increasingly popular. To see how to do this, it helps to consider recursive and non-recursive usage separately.

Non-recursive

A non-recursive use case is where a certain URL maps to a table, and the following component is a record ID. For instance:

```python
# /persons/123   =>   root["persons"][123]

import myapp.model as model
```

```python

class Resource(dict):
    def __init__(self, name, parent):
        self.__name__ = name
        self.__parent__ = parent

class Root(Resource):
    """The root resource."""

    def add_resource(self, name, orm_class):
        self[name] = ORMContainer(name, self, self.request, orm_class)

    def __init__(self, request):
        self.request = request
        self.add_resource('persons', model.Person)

root_factory = Root

class ORMContainer(dict):
    """Traversal component tied to a SQLAlchemy ORM class.

    Calling .__getitem__ fetches a record as an ORM instance, adds certain
    attributes to the object, and returns it.
    """
    def __init__(self, name, parent, request, orm_class):
        self.__name__  = name
        self.__parent__ = parent
        self.request = request
        self.orm_class = orm_class

    def __getitem__(self, key):
        try:
            key = int(key)
        except ValueError:
            raise KeyError(key)
        obj = model.DBSession.query(self.orm_class).get(key)
        # If the ORM class has a class method '.get' that performs the
        # query, you could do this:  ``obj = self.orm_class.get(key)``
        if obj is None:
            raise KeyError(key)
        obj.__name__ = key
        obj.__parent__ = self
        return obj
```

Here, `root["persons"]` is a container object whose `__getitem__` method fetches the specified database record, sets name and parent attribues on it, and returns it. (We've verified that SQLAlchemy does not define `__name__` or `__parent__` attributes in ORM instances.) If the record is not found, raise KeyError to indicate the resource doesn't exist.

TODO: Describe URL generation, access control lists, and other things needed in a complete application.

One drawback of this approach is that you have to fetch the entire record in order to generate a URL to it. This does not help if you have index views that display links to records, by querying the database directly for the IDs that match a criterion (N most recent records, all records by date, etc). You don't want to fetch the entire record's body, or do something silly like asking traversal for the resource at "/persons/123" and then generate the URL – which would be "/persons/123"! There are a few ways to generate URLs in this case:

- Define a generation-only route; e.g., `config.add_route("person", "/persons/{id}", static=True)`

- Instead of returning an ORM instance, return a proxy that lazily fetches the instance when its attributes are accessed. This causes traversal to behave somewhat incorrectly. It *should* raise KeyError if the record doesn't exist, but it can't know whether the record exists without fetching it. If traversal returns a possibly-invalid resource, it puts a burden on the view to check whether its context is valid. Normally the view can just assume it is, otherwise the view wouldn't have been invoked.

Recursive

The prototypical recursive use case is a content management system, where the user can define URLs arbitrarily deep; e.g., "/a/b/c". It can also be useful with "canned" data, where you want a small number of views to respond to a large variety of URL hierarchies.

Kotti is the best current example of using traversal with SQLAlchemy recursively. Kotti is a content management system that, yes, lets users define arbitrarily deep URLs. Specifically, Kotti allows users to define a page with subpages; e.g., a "directory" of pages.

Kotti is rather complex and takes some time to study. It uses SQLAlchemy's polymorphism to make tables "inherit" from other tables. This is an advanced feature which can be complex to grok. On the other hand, if you have the time, it's a great way to learn how to do recursive traversal and polymorphism.

The main characteristic of a recursive SQL setup is a self-referential table; i.e., table with a foreign key colum pointing to the same table. This allows each record to point to its parent. (The root record has NULL in the parent field.)

For more information on URL dispatch, see the URL Dispatch section of the Pyramid documentation.

For more information traversal, see the following sections of the Pyramid documentation:

- Hello Traversal
- Much Ado about Traversal
- Traversal
- Hybrid Dispatching
- Virtual Hosting

1.11 Static Assets (Static Files)

1.11.1 Serving File Content Dynamically

Usually you'll use a static view (via "config.add_static_view") to serve file content that lives on the filesystem. But sometimes files need to be composed and read from a nonstatic area, or composed on the fly by view code and served out (for example, a view callable might construct and return a PDF file or an image).

By way of example, here's a Pyramid application which serves a single static file (a jpeg) when the URL `/test.jpg` is executed:

```
from pyramid.view import view_config
from pyramid.config import Configurator
from pyramid.response import FileResponse
from paste.httpserver import serve

@view_config(route_name='jpg')
def test_page(request):
    response = FileResponse(
        '/home/chrism/groundhog1.jpg',
        request=request,
        content_type='image/jpeg'
        )
    return response

if __name__ == '__main__':
    config = Configurator()
    config.add_route('jpg', '/test.jpg')
    config.scan('__main__')
    serve(config.make_wsgi_app())
```

Basically, use a `pyramid.response.FileResponse` as the response object and return it. Note that the `request` and `content_type` arguments are optional. If `request` is not supplied, any `wsgi.file_wrapper` optimization supplied by your WSGI server will not be used when serving the file. If `content_type` is not supplied, it will be guessed using the `mimetypes` module (which uses the file extension); if it cannot be guessed successfully, the `application/octet-stream` content type will be used.

1.11.2 Serving a Single File from the Root

If you need to serve a single file such as `/robots.txt` or `/favicon.ico` that *must* be served from the root, you cannot use a static view to do it, as static views cannot serve files from the root (a static view must have a nonempty prefix such as `/static`). To work around this limitation, create views "by hand" that serve up the raw file data. Below is an example of creating two views: one serves up a `/favicon.ico`, the other serves up `/robots.txt`.

At startup time, both files are read into memory from files on disk using plain Python. A Response object is created for each. This response is served by a view which hooks up the static file's URL.

```
# this module = myapp.views

import os

from pyramid.response import Response
from pyramid.view import view_config

# _here = /app/location/myapp

_here = os.path.dirname(__file__)

# _icon = /app/location/myapp/static/favicon.ico

_icon = open(os.path.join(
             _here, 'static', 'favicon.ico')).read()
_fi_response = Response(content_type='image/x-icon',
                        body=_icon)

# _robots = /app/location/myapp/static/robots.txt

_robots = open(os.path.join(
               _here, 'static', 'robots.txt')).read()
_robots_response = Response(content_type='text/plain',
                            body=_robots)

@view_config(name='favicon.ico')
def favicon_view(context, request):
    return _fi_response

@view_config(name='robots.txt')
```

```
def robotstxt_view(context, request):
    return _robots_response
```

1.11.3 Root-Relative Custom Static View (URL Dispatch Only)

The `pyramid.static.static_view` helper class generates a Pyramid view callable. This view callable can serve static assets from a directory. An instance of this class is actually used by the `pyramid.config.Configurator.add_static_view()` configuration method, so its behavior is almost exactly the same once it's configured.

Warning: The following example *will not work* for applications that use traversal, it will only work if you use URL dispatch exclusively. The root-relative route we'll be registering will always be matched before traversal takes place, subverting any views registered via `add_view` (at least those without a `route_name`). A `pyramid.static.static_view` cannot be made root-relative when you use traversal.

To serve files within a directory located on your filesystem at `/path/to/static/dir` as the result of a "catchall" route hanging from the root that exists at the end of your routing table, create an instance of the `pyramid.static.static_view` class inside a `static.py` file in your application root as below:

```
from pyramid.static import static_view
www = static_view('/path/to/static/dir', use_subpath=True)
```

Note: For better cross-system flexibility, use an asset specification as the argument to `pyramid.static.static_view` instead of a physical absolute filesystem path, e.g. `mypackage:static` instead of `/path/to/mypackage/static`.

Subsequently, you may wire the files that are served by this view up to be accessible as `/<filename>` using a configuration method in your application's startup code:

```
# .. every other add_route and/or add_handler declaration should come
# before this one, as it will, by default, catch all requests

config.add_route('catchall_static', '/*subpath', 'myapp.static.www')
```

The special name `*subpath` above is used by the `pyramid.static.static_view` view callable to signify the path of the file relative to the directory you're serving.

1.11.4 Basic File Uploads

There are two parts necessary for handling file uploads. The first is to make sure you have a form that's been setup correctly to accept files. This means adding `enctype` attribute to your `form` element with the value of `multipart/form-data`. A very simple example would be a form that accepts an mp3 file. Notice we've setup the form as previously explained and also added an `input` element of the `file` type.

```
<form action="/store_mp3_view" method="post" accept-charset="utf-8"
      enctype="multipart/form-data">

    <label for="mp3">Mp3</label>
    <input id="mp3" name="mp3" type="file" value="" />

    <input type="submit" value="submit" />
</form>
```

The second part is handling the file upload in your view callable (above, assumed to answer on `/store_mp3_view`). The uploaded file is added to the request object as a `cgi.FieldStorage` object accessible through the `request.POST` multidict. The two properties we're interested in are the `file` and `filename` and we'll use those to write the file to disk:

```python
import os
import shutil

from pyramid.response import Response

def store_mp3_view(request):
    # ``filename`` contains the name of the file in string format.
    #
    # WARNING: Internet Explorer is known to send an absolute file
    # *path* as the filename.  This example is naive; it trusts
    # user input.
    filename = request.POST['mp3'].filename

    # ``input_file`` contains the actual file data which needs to be
    # stored somewhere.
    input_file = request.POST['mp3'].file

    # Using the filename like this without cleaning it is very
    # insecure so please keep that in mind when writing your own
    # file handling.
    file_path = os.path.join('/tmp', filename)
    with open(file_path, 'wb') as output_file:
        shutil.copyfileobj(input_file, output_file)

    return Response('OK')
```

For more information on static assets, see the Static Assets section of the Pyramid documentation.

1.12 Templates and Renderers

1.12.1 Using a Before Render Event to Expose an `h` Helper Object

Pylons 1.X exposed a module conventionally named `helpers.py` as an `h` object in the top-level namespace of each Mako/Genshi/Jinja2 template which it rendered. You can emulate the same behavior in Pyramid by using a `BeforeRender` event subscriber.

First, create a module named `helpers.py` in your Pyramid package at the top level (next to `__init__.py`). We'll import the Python standard library `string` module to use later in a template:

```python
# helpers.py

import string
```

In the top of the main `__init__` module of your Pyramid application package, import the new `helpers` module you created, as well as the `BeforeRender` event type. Underneath the imports create a function that will act as an event subscriber:

```python
# __init__.py

from pyramid.events import BeforeRender
from myapp import helpers
```

```python

def add_renderer_globals(event):
   event['h'] = helpers
```

Within the `main` function in the same `__init__`, wire the subscriber up so that it is called when the `BeforeRender` event is emitted:

```python
def main(global_settings, **settings):
    config = Configurator(....) # existing code
    # .. existing config statements ... #
    config.add_subscriber(add_renderer_globals, BeforeRender)
    # .. other existing config statements and eventual config.make_app()
```

At this point, with in any view that uses any templating system as a Pyramid renderer, you will have an omnipresent `h` top-level name that is a reference to the `helpers` module you created. For example, if you have a view like this:

```python
@view_config(renderer='foo.pt')
def aview(request):
    return {}
```

In the `foo.pt` Chameleon template, you can do this:

```
 ${h.string.uppercase}
```

The value inserted into the template as the result of this statement will be `ABCDEFGHIJKLMNOPQRSTUVWXYZ` (at least if you are using an English system).

You can add more imports and functions to `helpers.py` as necessary to make features available in your templates.

1.12.2 Using a BeforeRender Event to Expose a Mako `base` Template

If you wanted to change templates using `%inherit` based on if a user was logged in you could do the following:

```python
@subscriber(BeforeRender)
def add_base_template(event):
    request = event.get('request')
    if request.user:
        base = 'myapp:templates/logged_in_layout.mako'
        event.update({'base': base})
    else:
        base = 'myapp:templates/layout.mako'
        event.update({'base': base})
```

And then in your mako file you can call %inherit like so:

```
<%inherit file="${context['base']}" />
```

You **must** call the variable this way because of the way Mako works. It will not know about any other variable other than `context` until after `%inherit` is called. Be aware that `context` here is not the Pyramid context in the traversal sense (which is stored in `request.context`) but rather the Mako rendering context.

1.12.3 Using a BeforeRender Event to Expose Chameleon `base` Template

To avoid defining the same basic things in each template in your application, you can define one `base` template, and inherit from it in other templates.

Note: Pyramid example application - shootout using this approach.

First, add subscriber within your Pyramid project's __init__.py:

```
config.add_subscriber('YOURPROJECT.subscribers.add_base_template',
                      'pyramid.events.BeforeRender')
```

Then add the `subscribers.py` module to your project's directory:

```python
from pyramid.renderers import get_renderer

def add_base_template(event):
    base = get_renderer('templates/base.pt').implementation()
    event.update({'base': base})
```

After this has been done, you can use your `base` template to extend other templates. For example, the `base` template looks like this:

```html
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      metal:define-macro="base">
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
        <title>My page</title>
    </head>
    <body>
        <tal:block metal:define-slot="content">
        </tal:block>
    </body>
</html>
```

Each template using the `base` template will look like this:

```html
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      metal:use-macro="base.macros['base']">
    <tal:block metal:fill-slot="content">
        My awesome content.
    </tal:block>
</html>
```

The `metal:use-macro="base.macros['base']"` statement is essential here. Content inside `<tal:block metal:fill-slot="content"></tal:block>` tags will replace corresponding block in `base` template. You can define as many slots in as you want. For more information please see Macro Expansion Template Attribute Language documentation.

1.12.4 Using Building Blocks with Chameleon

If you understood the `base` template chapter, using building blocks is very simple and straight forward. In the `subscribers.py` module extend the `add_base_template` function like this:

```python
from pyramid.events import subscriber
from pyramid.events import BeforeRender
from pyramid.renderers import get_renderer

@subscriber(BeforeRender)
def add_base_template(event):
    base = get_renderer('templates/base.pt').implementation()
    blocks = get_renderer('templates/blocks.pt').implementation()
    event.update({'base': base,
                  'blocks': blocks,
                  })
```

Make Pyramid scan the module so that it finds the `BeforeRender` event:

```
def main(global_settings, **settings):
    config = Configurator(....) # existing code
    # .. existing config statements ... #
    config.scan('subscriber')
    # .. other existing config statements and eventual config.make_app()
```

Now, define your building blocks in `templates/blocks.pt`. For example:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal">
  <tal:block metal:define-macro="base-paragraph">
    <p class="foo bar">
      <tal:block metal:define-slot="body">
      </tal:block>
    </p>
  </tal:block>

  <tal:block metal:define-macro="bold-paragraph"
             metal:extend-macro="macros['base-paragraph']">
    <tal:block metal:fill-slot="body">
      <b class="strong-class">
        <tal:block metal:define-slot="body"></tal:block>
      </b>
    </tal:block>
  </tal:block>
</html>
```

You can now use these building blocks like this:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      metal:use-macro="base.macros['base']">
  <tal:block metal:fill-slot="content">
    <tal:block metal:use-macro="blocks.macros['base-paragraph']">
      <tal:block metal:fill-slot="body">
        My awesome paragraph.
      </tal:block>
    </tal:block>

    <tal:block metal:use-macro="blocks.macros['bold-paragraph']">
      <tal:block metal:fill-slot="body">
        My awesome paragraph in bold.
      </tal:block>
    </tal:block>

  </tal:block>
</html>
```

1.12.5 Rendering `None` as the Empty String in Mako Templates

For the following Mako template:

```
<p>${nunn}</p>
```

By default, Pyramid will render:

```
<p>None</p>
```

Some folks prefer the value `None` to be rendered as the empty string in a Mako template. In other words, they'd rather the output be:

```
<p></p>
```

Use the following settings in your Pyramid configuration file to obtain this behavior:

```
[app:myapp]
mako.imports = from markupsafe import escape_silent
mako.default_filters = escape_silent
```

1.12.6 Mako Internationalization

Note: This recipe is extracted, with permission, from a blog post made by Alexandre Bourget.

First, add subscribers within your Pyramid project's `__init__.py`:

```
def main(...):
    ...
    config.add_subscriber('YOURPROJECT.subscribers.add_renderer_globals',
                          'pyramid.events.BeforeRender')
    config.add_subscriber('YOURPROJECT.subscribers.add_localizer',
                          'pyramid.events.NewRequest')
```

Then add, a `subscribers.py` module to your project's package directory:

```
# subscribers.py

from pyramid.i18n import get_localizer, TranslationStringFactory

def add_renderer_globals(event):
    ...
    request = event['request']
    event['_'] = request.translate
    event['localizer'] = request.localizer

tsf = TranslationStringFactory('YOUR_GETTEXT_DOMAIN')

def add_localizer(event):
    request = event.request
    localizer = get_localizer(request)
    def auto_translate(*args, **kwargs):
        return localizer.translate(tsf(*args, **kwargs))
    request.localizer = localizer
    request.translate = auto_translate
```

After this has been done, the next time you start your application, in your Mako template, you'll be able to use the simple `${_(u"Translate this string please")}` without having to use `get_localizer` explicitly, as its functionality will be enclosed in the _ function, which will be exposed as a top-level template name. `localizer` will also be available for plural forms and fancy stuff.

This will also allow you to use translation in your view code, using something like:

```
def my_view(request):
    _ = request.translate
    request.session.flash(_("Welcome home"))
```

For all that to work, you'll need to:

```
(env)$ easy_install Babel
```

And you'll also need to run these commands in your project's directory:

```
(env)$ python setup.py extract_messages
(env)$ python setup.py init_catalog -l en
(env)$ python setup.py init_catalog -l fr
(env)$ python setup.py init_catalog -l es
(env)$ python setup.py init_catalog -l it
(env)$ python setup.py update_catalog
(env)$ python setup.py compile_catalog
```

Repeat the `init_catalog` step for each of the langauges you need.

Note: The gettext sub-directory of your project is `locale/` in Pyramid, and not `i18n/` as it was in Pylons. You'll notice that in the default setup.cfg of a Pyramid project.

At this point you'll also need to add your local directory to your project's configuration:

```
def main(...):
   ...
   config.add_translation_dirs('YOURPROJECT:locale')
```

Lastly, you'll want to have your Mako files extracted when you run extract_messages, so add these to your setup.py (yes, you read me right, in setup.py so that Babel can use it when invoking it's commands):

```
setup(
    ...
    install_requires=[
        ...
        Babel,
        ...
        ],
    message_extractors = {'yourpackage': [
            ('**.py', 'python', None),
            ('templates/**.html', 'mako', None),
            ('templates/**.mako', 'mako', None),
            ('static/**', 'ignore', None)]},
    ...
    )
```

In the above triples the last element, `None` in this snippet, may be used to pass an options dictionary to the specified extractor. For instance, you may need to set Mako input encoding using the corresponding option:

```
...
           ('templates/**.mako', 'mako', {'input_encoding': 'utf-8'}),
...
```

1.12.7 Chameleon Internationalization

Note: This recipe was created to document the process of internationalization (i18n) and localization (l10n) of chameleon templates. There is not much to it, really, but as the author was baffled by this fact, it seems a good idea to describe the few necessary steps.

We start off with a virtualenv and a fresh Pyramid project created via paster:

```
$ virtualenv --no-site-packages env
$ env/bin/pip install pyramid
$ env bin/paster create -t pyramid_routesalchemy ChameleonI18n
```

Dependencies

First, add dependencies to your Pyramid project's `setup.py`:

```
requires = [
    ...
    'Babel',
    'lingua',
    ]
...
message_extractors = { '.': [
    ('**.py',   'lingua_python', None ),
    ('**.pt',   'lingua_xml', None ),
    ]},
```

You will have to run `../env/bin/python setup.py develop` after this to get Babel and lingua into your virtualenv and make the message extraction work.

A Folder for the locales

Next, add a folder for the locales POT & PO files:

```
$ mkdir chameleoni18n/locale
```

What to translate

Well, let's translate some parts of the given template `mytemplate.pt`. Add a namespace and an i18n:domain to the <html> tag:

```
-<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" xmlns:tal="http://xml.zope.org/namespaces/t
+<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" xmlns:tal="http://xml.zope.org/namespaces/t
+       xmlns:i18n="http://xml.zope.org/namespaces/i18n"
+       i18n:domain="ChameleonI18n">
```

The important bit – the one the author was missing – is that the i18n:domain must be spelled exactely like the POT/PO/MO files created later on, including case. Without this, the translations will not be picked up.

So now we can mark a part of the template for translation:

```
-           <h2>Search documentation</h2>
+           <h2 i18n:translate="search_documentation">Search documentation</h2>
```

The i18n:translate attribute tells lingua to extract the contents of the h2 tag to the catalog POT. You don't have to add a description (like in this example 'search_documentation'), but it makes it easier for translators.

Commands for Translations

Now you need to run these commands in your project's directory:

```
(env)$ python setup.py extract_messages
(env)$ python setup.py init_catalog -l de
(env)$ python setup.py init_catalog -l fr
(env)$ python setup.py init_catalog -l es
(env)$ python setup.py init_catalog -l it
(env)$ python setup.py update_catalog
(env)$ python setup.py compile_catalog
```

Repeat the `init_catalog` step for each of the langauges you need.

The first command will extract the strings for translation to your projects locale/<project-name>.pot file, in this case ChameleonI18n.pot

The `init` commands create new catalogs for different languages and the `update` command will sync entries from the main POT to the languages POs.

At this point you can tell your translators to go edit the po files :-) Otherwise the translations will remain empty and defaults will be used.

Finally, the `compile` command will translate the POs to binary MO files that are actually used to get the relevant translations.

Note: The gettext sub-directory of your project is `locale/` in Pyramid, and not `i18n/` as it was in Pylons. You'll notice that in the default setup.cfg of a Pyramid project, which has all the necessary settings to make the above commands work without parameters.

Add locale directory to projects config

At this point you'll also need to add your local directory to your project's configuration:

```
def main(...):
    ...
    config.add_translation_dirs('YOURPROJECT:locale')
```

where YOURPROJECT in our example would be 'chameleoni18n'.

Set a default locale

You can now change the default locale for your project in `development.ini` and see if the translations are being picked up.

```
-   pyramid.default_locale_name = en
+   pyramid.default_locale_name = de
```

Of course, you need to have edited your relevant PO file and added a translation of the relevant string, in this example `search_documentation` and have the PO file compiled to a MO file. Now you can fire up you app and check out the translated headline.

1.12.8 Custom Renderers

Pyramid supports custom renderers, alongside the default renderers shipped with Pyramid.

Here's a basic comma-separated value (CSV) renderer to output a CSV file to the browser. Add the following to a `renderers.py` module in your application (or anywhere else you'd like to place such things):

```
import csv
try:
    from StringIO import StringIO # python 2
except ImportError:
    from io import StringIO # python 3

class CSVRenderer(object):
   def __init__(self, info):
      pass

   def __call__(self, value, system):
      """ Returns a plain CSV-encoded string with content-type
      ``text/csv``. The content-type may be overridden by
      setting ``request.response.content_type``."""

      request = system.get('request')
      if request is not None:
         response = request.response
         ct = response.content_type
         if ct == response.default_content_type:
            response.content_type = 'text/csv'

      fout = io.StringIO()
      writer = csv.writer(fout, delimiter=',', quotechar=',', quoting=csv.QUOTE_MINIMAL)

      writer.writerow(value.get('header', []))
      writer.writerows(value.get('rows', []))

      return fout.getvalue()
```

Now you have a renderer. Let's register with our application's `Configurator`:

```
config.add_renderer('csv', 'myapp.renderers.CSVRenderer')
```

Of course, modify the dotted-string to point to the module location you decided upon. To use the renderer, create a view:

```
@view_config(route_name='data', renderer='csv')
def my_view(request):
   query = DBSession.query(table).all()
   header = ['First Name', 'Last Name']
   rows = [[item.first_name, item.last_name] for item in query]

   # override attributes of response
   filename = 'report.csv'
   request.response.content_disposition = 'attachment;filename=' + filename

   return {
      'header': header,
      'rows': rows,
   }

def main(global_config, **settings):
    config = Configurator(settings=settings)
    config.add_route('data', '/data')
    config.scan()
    return config.make_wsgi_app()
```

Query your database in your `query` variable, establish your `headers` and initialize `rows`.

Override attributes of response as required by your use case. We implement this aspect in view code to keep our custom renderer code focused to the task.

Lastly, we pass `headers` and `rows` to the CSV renderer.

For more information on how to add custom Renderers, see the following sections of the Pyramid documentation:

- Adding a new Renderer
- Varying Attributes of Rendered Responses

For more information on Templates and Renderers, see the following sections of the Pyramid documentation:

- Templates
- Renderers
- Internationalization and Localization

1.13 Testing

1.13.1 Testing a POST request using cURL

Using the following Pyramid application:

```
from wsgiref.simple_server import make_server
from pyramid.view import view_config
from pyramid.config import Configurator

@view_config(route_name='theroute', renderer='json',
             request_method='POST')
def myview(request):
    return {'POST': request.POST.items()}

if __name__ == '__main__':
    config = Configurator()
    config.add_route('theroute', '/')
    config.scan()
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 6543, app)
    print server.base_environ
    server.serve_forever()
```

Once you run the above application, you can test a POST request to the application via `curl` (available on most UNIX systems).

```
$ python application.py
{'CONTENT_LENGTH': '', 'SERVER_NAME': 'Latitude-XT2', 'GATEWAY_INTERFACE': 'CGI/1.1',
 'SCRIPT_NAME': '', 'SERVER_PORT': '6543', 'REMOTE_HOST': ''}
```

To access POST request body values (provided as the argument to the `-d` flag of `curl`) use `request.POST`.

```
$ curl -i -d "param1=value1&param2=value2" http://localhost:6543/
HTTP/1.0 200 OK
Date: Tue, 09 Sep 2014 09:34:27 GMT
```

```
Server: WSGIServer/0.1 Python/2.7.5+
Content-Type: application/json; charset=UTF-8
Content-Length: 54

{"POST": [["param1", "value1"], ["param2", "value2"]]}
```

To access QUERY_STRING parameters as well, use `request.GET`.

```
@view_config(route_name='theroute', renderer='json',
             request_method='POST')
def myview(request):
    return {'GET':request.GET.items(),
            'POST':request.POST.items()}
```

Append QUERY_STRING parameters to previously used URL and query with curl.

```
$ curl -i -d "param1=value1&param2=value2" http://localhost:6543/?param3=value3
HTTP/1.0 200 OK
Date: Tue, 09 Sep 2014 09:39:53 GMT
Server: WSGIServer/0.1 Python/2.7.5+
Content-Type: application/json; charset=UTF-8
Content-Length: 85

{"POST": [["param1", "value1"], ["param2", "value2"]], "GET": [["param3", "value3"]]}
```

Use `request.params` to have access to dictionary-like object containing both the parameters from the query string and request body.

```
@view_config(route_name='theroute', renderer='json',
             request_method='POST')
def myview(request):
    return {'GET':request.GET.items(),
            'POST':request.POST.items(),
            'PARAMS':request.params.items()}
```

Another request with curl.

```
$ curl -i -d "param1=value1&param2=value2" http://localhost:6543/?param3=value3
HTTP/1.0 200 OK
Date: Tue, 09 Sep 2014 09:53:16 GMT
Server: WSGIServer/0.1 Python/2.7.5+
Content-Type: application/json; charset=UTF-8
Content-Length: 163

{"POST": [["param1", "value1"], ["param2", "value2"]],
 "PARAMS": [["param3", "value3"], ["param1", "value1"], ["param2", "value2"]],
 "GET": [["param3", "value3"]]}
```

Here's a simple Python program that will do the same as the `curl` command above does.

```
import httplib
import urllib
from contextlib import closing

with closing(httplib.HTTPConnection("localhost", 6543)) as conn:
    headers = {"Content-type": "application/x-www-form-urlencoded"}
    params = urllib.urlencode({'param1': 'value1', 'param2': 'value2'})
    conn.request("POST", "?param3=value3", params, headers)
    response = conn.getresponse()
    print response.getheaders()
    print response.read()
```

Running this program on a console.

```
$ python request.py
[('date', 'Tue, 09 Sep 2014 10:18:46 GMT'), ('content-length', '163'), ('content-type', 'application/
{"POST": [["param2", "value2"], ["param1", "value1"]], "PARAMS": [["param3", "value3"], ["param2", "v
```

For more information on testing see the Testing section of the Pyramid documentation.

For additional information on other testing packages see:

- WebTest
- nose

1.14 Views

1.14.1 Chaining Decorators

Pyramid has a `decorator=` argument to its view configuration. It accepts a single decorator that will wrap the *mapped* view callable represented by the view configuration. That means that, no matter what the signature and return value of the original view callable, the decorated view callable will receive two arguments: `context` and `request` and will return a response object:

```python
# the decorator

def decorator(view_callable):
    def inner(context, request):
        return view_callable(context, request)
    return inner

# the view configuration

@view_config(decorator=decorator, renderer='json')
def myview(request):
    return {'a':1}
```

But the `decorator` argument only takes a single decorator. What happens if you want to use more than one decorator? You can chain them together:

```python
def combine(*decorators):
    def floo(view_callable):
        for decorator in decorators:
            view_callable = decorator(view_callable)
        return view_callable
    return floo

def decorator1(view_callable):
    def inner(context, request):
        return view_callable(context, request)
    return inner

def decorator2(view_callable):
    def inner(context, request):
        return view_callable(context, request)
    return inner

def decorator3(view_callable):
    def inner(context, request):
        return view_callable(context, request)
    return inner

alldecs = combine(decorator1, decorator2, decorator3)
two_and_three = combine(decorator2, decorator3)
one_and_three = combine(decorator1, decorator3)

@view_config(decorator=alldecs, renderer='json')
def myview(request):
    return {'a':1}
```

1.14.2 Using a View Mapper to Pass Query Parameters as Keyword Arguments

Pyramid supports a concept of a "view mapper". See Using a View Mapper for general information about view mappers. You can use a view mapper to support an alternate convenience calling convention in which you allow view callables to name extra required and optional arguments which are taken from the request.params dictionary. So, for example, instead of:

```
@view_config()
def aview(request):
    name = request.params['name']
    value = request.params.get('value', 'default')
    ...
```

With a special view mapper you can define this as:

```
@view_config(mapper=MapplyViewMapper)
def aview(request, name, value='default'):
    ...
```

The below code implements the `MapplyViewMapper`. It works as a mapper for function view callables and method view callables:

```
import inspect
import sys

from pyramid.view import view_config
from pyramid.response import Response
from pyramid.config import Configurator
from waitress import serve

PY3 = sys.version_info[0] == 3

if PY3:
    im_func = '__func__'
    func_defaults = '__defaults__'
    func_code = '__code__'
else:
    im_func = 'im_func'
    func_defaults = 'func_defaults'
    func_code = 'func_code'

def mapply(ob, positional, keyword):

    f = ob
    im = False

    if hasattr(f, im_func):
        im = True

    if im:
        f = getattr(f, im_func)
        c = getattr(f, func_code)
        defaults = getattr(f, func_defaults)
        names = c.co_varnames[1:c.co_argcount]
    else:
        defaults = getattr(f, func_defaults)
        c = getattr(f, func_code)
        names = c.co_varnames[:c.co_argcount]

    nargs = len(names)
    args = []
    if positional:
        positional = list(positional)
        if len(positional) > nargs:
            raise TypeError('too many arguments')
        args = positional

    get = keyword.get
    nrequired = len(names) - (len(defaults or ()))
    for index in range(len(args), len(names)):
        name = names[index]
        v = get(name, args)
        if v is args:
            if index < nrequired:
                raise TypeError('argument %s was omitted' % name)
            else:
                v = defaults[index-nrequired]
        args.append(v)

    args = tuple(args)
    return ob(*args)


class MapplyViewMapper(object):
    def __init__(self, **kw):
        self.attr = kw.get('attr')

    def __call__(self, view):
        def wrapper(context, request):
            keywords = dict(request.params.items())
            if inspect.isclass(view):
                inst = view(request)
                meth = getattr(inst, self.attr)
                response = mapply(meth, (), keywords)
            else:
                # it's a function
                response = mapply(view, (request,), keywords)
            return response

        return wrapper
```

```python

@view_config(name='function', mapper=MapplyViewMapper)
def view_function(request, one, two=False):
    return Response('one: %s, two: %s' % (one, two))

class ViewClass(object):
    __view_mapper__ = MapplyViewMapper
    def __init__(self, request):
        self.request = request

    @view_config(name='method')
    def view_method(self, one, two=False):
        return Response('one: %s, two: %s' % (one, two))

if __name__ == '__main__':
    config = Configurator()
    config.scan('.')
    app = config.make_wsgi_app()
    serve(app)

# http://localhost:8080/function --> (exception; no "one" arg supplied)

# http://localhost:8080/function?one=1 --> one: '1', two: False

# http://localhost:8080/function?one=1&two=2 --> one: '1', two: '2'

# http://localhost:8080/method --> (exception; no "one" arg supplied)

# http://localhost:8080/method?one=1 --> one: '1', two: False

# http://localhost:8080/method?one=1&two=2 --> one: '1', two: '2'
```

1.14.3 Conditional HTTP

Pyramid requests and responses support conditional HTTP requests via the `ETag` and `Last-Modified` header. It is useful to enable this for an entire site to save on bandwidth for repeated requests. Enabling `ETag` support for an entire site can be done using a tween:

```python
def conditional_http_tween_factory(handler, registry):
    def conditional_http_tween(request):
        response = handler(request)

        # If the Last-Modified header has been set, we want to enable the
        # conditional response processing.
        if response.last_modified is not None:
            response.conditional_response = True

        # We want to only enable the conditional machinery if either we
        # were given an explicit ETag header by the view or we have a
        # buffered response and can generate the ETag header ourself.
        if response.etag is not None:
            response.conditional_response = True
        elif (isinstance(response.app_iter, collections.abc.Sequence) and
                len(response.app_iter) == 1):
            response.conditional_response = True
            response.md5_etag()

        return response
    return conditional_http_tween
```

The effect of this tween is that it will first check the response to determine if it already has a `Last-Modified` or `ETag` header set. If it does, then it will enable the conditional response processing. If the response does not have an `ETag` header set, then it will attempt to determine if the response is already loaded entirely into memory (to avoid loading what might be a very large object into memory). If it is already loaded into memory, then it will generate an `ETag` header from the MD5 digest of the response body, and again enable the conditional response processing.

For more information on views, see the Views section of the Pyramid documentation.

1.15 Automating the Development Process

1.15.1 What is pyramid_starter_seed

This tutorial should help you to start developing with the Pyramid web framework using a very minimal starter seed project based on:

- a Pyramid's *pcreate -t starter* project
- a Yeoman *generator-webapp* project

You can find the Pyramid starter seed code here on Github:

- pyramid_starter_seed

Thanks to Yeoman you can improve your developer experience when you are in **development** or **production** mode thanks to:

- Javascript testing setup
- Javascript code linting
- Javascript/CSS concat and minification
- image assets optimization
- html template minification
- switch to CDN versions of you vendor plugins in production mode
- uncss
- much more (you can add features adding new Grunt tasks)

We will see later how you can clone *pyramid_starter_seed* from github, add new features (eg: authentication, SQLAlchemy support, user models, a json REST API, add a modern Javascript framework as AngularJS, etc) and then launch a console script that helps you to rename the entire project with your more opinionated modifications, for example *pyramid_yourawesomeproduct*.

Based on Davide Moro articles (how to integrate the Yeoman workflow with Pyramid):

- Pyramid starter seed template powered by Yeoman (part 1)
- Pyramid starter seed template powered by Yeoman (part 2)
- Pyramid starter seed template powered by Yeoman (part 3)

1.15.2 Prerequisites

If you want to play with *pyramid_starter_seed* you'll need to install NodeJS and, obviously, Python. Once installed Python and Pyramid, you'll have to clone the pyramid_starter_seed repository from github and initialize the Yeoman stuff.

Python and Pyramid

pyramid_starter_seed was tested with Python 2.7. Create an isolated Python environment as explained in the official Pyramid documentation and install Pyramid.

Official Pyramid installation documentation

- http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/install.html#installing-chapter

NodeJS

You won't use NodeJS at all in your code, you just need to install development dependencies required by the Yeoman tools.

Once installed NodeJS (if you want to easily install different versions on your system and manage them you can use the NodeJS Version Manager utility: NVM), you need to enable the following tools:

```
$ npm install -g bower
$ npm install -g grunt-cli
$ npm install -g karma
```

Tested with NodeJS version 0.10.31.

How to install pyramid_starter_seed

Clone *pyramid_starter_seed* from github:

```
$ git clone git@github.com:davidemoro/pyramid_starter_seed.git
$ cd pyramid_starter_seed
$ YOUR_VIRTUALENV_PYTHON_PATH/bin/python setup.py develop
```

Yeoman initialization

Go to the folder where it lives our Yeoman project and initialize it.

These are the standard commands (but, wait a moment, see the "Notes and known issues" subsection):

```
$ cd pyramid_starter_seed/webapp
$ bower install
$ npm install
```

Known issues

You'll need to perform these additional steps in order to get a working environment (the generator-webapp's version used by pyramid_starter_seed has a couple of known issues).

Avoid imagemin errors on build:

```
$ npm cache clean
$ npm install grunt-contrib-imagemin
```

Avoid Mocha/PhantomJS issue (see issues #446):

```
$ cd test
$ bower install
```

Build

Run:

```
$ grunt build
```

Run pyramid_starter_seed

Now can choose to run Pyramid in development or production mode.

Go to the root of your project directory, where the files *development.ini* and *production.ini* are located.

```
cd ../../..
```

Just type:

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

or:

```
$ YOUR_VIRTUALENV_PYTHON_PATH/bin/pserve production.ini
```

1.15.3 How it works pyramid_starter_seed

Note well that if you want to integrate a Pyramid application with the Yeoman workflow you can choose different strategies. So the pyramid_starter_seed's way is just one of the possible implementations.

.ini configurations

Production vs development .ini configurations.

Production:

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

PRODUCTION = true
minify = dist

...
```

Development:

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

PRODUCTION = false
```

```
minify = app
...
```

View callables

The view callable gets a different renderer depending on the production vs development settings:

```python
from pyramid.view import view_config

@view_config(route_name='home', renderer='webapp/%s/index.html')
def my_view(request):
    return {'project': 'pyramid_starter_seed'}
```

Since there is no .html renderer, pyramid_starter_seed register a custom Pyramid renderer based on ZPT/Chameleon. See .html renderer

Templates

Css and javascript

```
<tal:production tal:condition="production">
    <script src="${request.static_url('pyramid_starter_seed:webapp/%s/scripts/plugins.js' % minify)}
    </script>
</tal:production>
<tal:not_production tal:condition="not:production">
    <script src="${request.static_url('pyramid_starter_seed:webapp/%s/bower_components/bootstrap/js/a
    </script>
    <script src="${request.static_url('pyramid_starter_seed:webapp/%s/bower_components/bootstrap/js/d
    </script>
</tal:not_production>
<!-- build:js scripts/plugins.js -->
<tal:comment replace="nothing">
    <!-- DO NOT REMOVE this block (minifier) -->
    <script src="./bower_components/bootstrap/js/alert.js"></script>
    <script src="./bower_components/bootstrap/js/dropdown.js"></script>
</tal:comment>
<!-- endbuild -->
```

Note: the above verbose syntax could be avoided hacking with the grunt-bridge task. See grunt-bridge.

Images

```
<img class="logo img-responsive"
     src="${request.static_url('pyramid_starter_seed:webapp/%s/images/pyramid.png' % minify)}"
     alt="pyramid web framework" />
```

How to fork pyramid_starter_seed

Fetch pyramid_starter_seed, personalize it and then clone it!

Pyramid starter seed can be fetched, personalized and released with another name. So other developer can bootstrap, build, release and distribute their own starter templates without having to write a new package template generator.

For example you could create a more opinionated starter seed based on SQLAlchemy, ZODB nosql or powered by a javascript framework like AngularJS and so on.

The clone method should speed up the process of creation of new more evoluted packages based on Pyramid, also people that are not keen on writing their own reusable scaffold templates.

So if you want to release your own customized template based on pyramid_starter_seed you'll have to call a console script named pyramid_starter_seed_clone with the following syntax (obviously you'll have to call this command outside the root directory of pyramid_starter_seed):

```
$ YOUR_VIRTUALENV_PYTHON_PATH/bin/pyramid_starter_seed_clone new_template
```

and you'll get as a result a perfect renamed clone new_template.

The clone console script it might not work in some corner cases just in case you choose a new package name that contains reserved words or the name of a dependency of your plugin, but it should be quite easy to fix by hand or improving the console script. But if you provide tests you can check immediately if something went wrong during the cloning process and fix.

How pyramid_starter_seed works under the hood

More details explained on the original article (part 3):

- How pyramid_starter_seed works under the hood

Based on Davide Moro articles (how to integrate the Yeoman workflow with Pyramid):

- Pyramid starter seed template powered by Yeoman (part 1)
- Pyramid starter seed template powered by Yeoman (part 2)
- Pyramid starter seed template powered by Yeoman (part 3)

1.16 Miscellaneous

1.16.1 Interfaces

This chapter contains information about using `zope.interface` with Pyramid.

Dynamically Compute the Interfaces Provided by an Object

(Via Marius Gedminas)

When persisting the interfaces that are provided by an object in a pickle or in ZODB is not reasonable for your application, you can use this trick to dynamically return the set of interfaces provided by an object based on other data in an instance of the object:

```
from zope.interface.declarations import Provides

from mypackage import interfaces

class MyClass(object):

    color = None

    @property
    def __provides__(self):
        # black magic happens here: we claim to provide the right IFrob
        # subinterface depending on the value of the ``color`` attribute.
        iface = getattr(interfaces, 'I%sFrob' % self.color.title(),
                        interfaces.IFrob))
        return Provides(self.__class__, iface)
```

If you need the object to implement more than one interface, use `Provides(self.__class__, iface1, iface2, ...)`.

1.16.2 Using Object Events in Pyramid

Warning: This code works only in Pyramid 1.1a4+. It will also make your brain explode.

Zope's Component Architecture supports the concept of "object events", which are events which call a subscriber with an context object *and* the event object.

Here's an example of using an object event subscriber via the `@subscriber` decorator:

```
from zope.component.event import objectEventNotify
from zope.component.interfaces import ObjectEvent

from pyramid.events import subscriber
from pyramid.view import view_config

class ObjectThrownEvent(ObjectEvent):
    pass

class Foo(object):
  pass

@subscriber([Foo, ObjectThrownEvent])
def objectevent_listener(object, event):
    print object, event

@view_config(renderer='string')
def theview(request):
    objectEventNotify(ObjectThrownEvent(Foo()))
    objectEventNotify(ObjectThrownEvent(None))
    objectEventNotify(ObjectEvent(Foo()))
    return 'OK'

if __name__ == '__main__':
    from pyramid.config import Configurator
    from paste.httpserver import serve
    config = Configurator(autocommit=True)
    config.hook_zca()
    config.scan('__main__')
    serve(config.make_wsgi_app())
```

The `objectevent_listener` listener defined above will only be called when the `object` of the ObjectThrownEvent is of class `Foo`. We can tell that's the case because only the first call to objectEventNotify actually invokes the subscriber. The second and third calls to objectEventNotify do not call the subscriber. The second call doesn't invoke the subscriber because its object type is `None` (and not `Foo`). The third call doesn't invoke the subscriber because its objectevent type is ObjectEvent (and not `ObjectThrownEvent`). Clear as mud?

1.16.3 How to Install Pyramid on Mac OS X

This guide will walk you through the steps of installing Pyramid on Mac OS X from the very beginning. This guide is targeted toward the complete newbie to Pyramid and Python, and aims to simplify an often confusing process fraught with frustrating pitfalls.

Pre-Requisites

- A Macintosh computer running Mac OS X (10.5.x or greater recommended).
- Xcode (recommended).

  At some point, you may need Xcode tools to "make" or compile applications from source code. Experienced developers on the Mac know this, and by installing Xcode once, they "set it and forget it".

  - For 10.5.x, use Xcode 3.1.4.
  - For 10.6.x, use Xcode 3.2.x or later.

    Visit the Apple Developer Connection website to download a massive installer, or use the Mac OS X installation DVD that came with your computer.

    Optionally, 10.6.x and later can run Xcode 4.x. However Apple charges a nominal fee to download Xcode 4 which is $4.99 at the time of this writing. Visit the Apple Developer Tools website for more information.

- Python 2.7.x or greater, but less than Python 3.x.

  Python comes pre-installed on Mac OS X, but due to Apple's release cycle, it's often one or even two years old. The overwhelming recommendation of the "MacPython" community is to upgrade your Python by downloading and installing a newer version from the Python standard release page. [2]. Note that using the pre-installed (aka "system" Python) is almost never a good idea on Mac OS X, even if the Python version happens to be fairly recent, due to heavy customizations made to the Python environment by Apple themselves. Do yourself a favor and download and install a custom Python instead.

  Python Releases

  At the time of this writing, Pyramid does not yet run under Python 3.x.

Configure Your Python Development Environment for Pyramid

Next we need to configure a Python environment for developing Pyramid applications by installing a few tools.

Install ez_setup.py for setuptools

Python projects often rely on third party modules to obtain added functionality. These modules are packaged using `distutils`, and they contain a Python script named `setup.py`. By executing this script, the developer can quickly and easily install the modules from source. Often it is as easy as navigating to the directory containing the script, and executing the following command:

```
python setup.py install
```

and the module will install itself.

However, not all setup.py scripts are well-written and they may blindly attempt to import the `setuptools` bootstrap module `ez_setup.py` even though `ez_setup.py` is usually not installed on the user's machine. This causes much trouble. [3]

[2] Python on the Mac
[3] ez_setup (Sridhar Ratnakumar)

To remedy the situation, we need to install the missing module `ez_setup.py` on our system. Execute the following commands in Terminal:

```
cd ~
curl -O http://peak.telecommunity.com/dist/ez_setup.py
sudo python ez_setup.py
```

Install virtualenv Using virtualenv-burrito

virtualenv is a tool that creates clean sandboxes for development. This removes the potential for conflicts from existing system libraries and allows you to isolate dependencies. [4]

virtualenvwrapper is a tool that makes working with virtualenv easier [5], and virtualenv-burrito is a single-command tool that installs and sets up virtualenvwrapper for you. [6]

Execute the following commands in Terminal:

```
cd ~
curl -s https://raw.github.com/brainsik/virtualenv-burrito/master/virtualenv-burrito.sh | bash
```

If successful, you will see this:

```
Fin.

Done with setup!

The virtualenvwrapper environment will be available when you login.

To start it now, run this:
source /Users/<username>/.venvburrito/startup.sh
```

Next run this command, substituting your username for `<username>`:

```
source /Users/<username>/.venvburrito/startup.sh
```

And you will see this:

```
To create a virtualenv, run:
mkvirtualenv <cool-name>
```

That statement is a little incomplete. Read on for an explanation.

Create a New virtualenv for Pyramid

When making a new virtualenv, we do not want to include any dependencies from outside the virtualenv, so we want to use the –no-site-packages flag. Run the command:

```
mkvirtualenv --no-site-packages pyramid
```

Now activate the new virtualenv:

```
workon pyramid
```

Now cd to the virtualenv directory:

[4] virtualenv (Ian Bicking)
[5] virtualenvwrapper (Doug Hellmann)
[6] virtualenv-burrito (Jeremy Avnet)

```
cd ~/.virtualenvs/pyramid
```

And now the moment you've been waiting for. . .

Install Pyramid

Run the command:

```
bin/easy_install pyramid
```

Pyramid should now be installed.

What Next?

Try the Pyramid Quick Tutorial.

Read Pyramid Documentation.

Contribute to the Pylons Project Documentation

The Pylons Project documentation uses Sphinx. [7] It is recommended to install Sphinx into the current virtualenv using easy_install:

```
easy_install -U Sphinx
```

Visit the Sphinx website.

The Pylons Project documentation has several components.

- Pylons Project documentation
- Pylons Project repository including documentation
- Pyramid Cookbook
- Pyramid Cookbook repository

Footnotes

1.16.4 Pyramid Tutorial/Informational Videos

- Six Feet Up's Intro to Basic Pyramid.
- Daniel Nouri's "Writing A Pyramid Application" (long, 3+ hours), from EuroPython 2012:
  - Part 1
  - Part 2

  See also the related blog post.
- Carlos de la Guardia's Writing a Pyramid Application tutorial from PyCon 2012 (long, 3+ hours).
- Dylan Jay's Pyramid: Lighter, faster, better web apps from PyCon AU 2011 (~37 mins).

[7] Sphinx

- Carlos de la Guardia's Patterns for building large Pyramid applications" (~25 minutes).
- Eric Bieschke's Pyramid Turbo Start Tutorial" (very short, 2 mins, 2011).
- Chris McDonough presentation to Helsinki Python User's Group about Pyramid (2012), about 30 mins.
- Chris McDonough at DjangoCon 2012, About Django from the Pyramid Guy (about 30 mins).
- Chris McDonough and Mark Ramm: FLOSS Weekly 151: The Pylons Project (about 40 mins, 2010).
- Kevin Gill's What is Pyramid and where is it with respect to Django" (~43 mins) via Python Ireland May 2011.
- Saiju M's Create Pyramid Application With SQLAlchemy (~ 17 mins).
- George Dubus' Pyramid advanced configuration tactics for nice apps and libs from EuroPython 2013 (~34 mins).
- Chris McDonough at PyCon 2013, Pyramid Auth Is Hard, Let's Ride Bikes (~30 mins).
- Dylan Jay's DjangoCon AU 2013 Keynote, The myth of goldilocks and the three frameworks, Pyramid, Django and Plone (~45 mins).
- Paul Everitt: Python 3 Web Development with Pyramid and PyCharm (~1 hr).

1.17 TODO

- Provide an example of using a newrequest subscriber to mutate the request, providing additional user data from a database based on the current authenticated userid.
- Provide an example of adding a database connection to `settings` in __init__ and using it from a view.
- Provide an example of a catchall 500 error view.
- Redirecting to a URL with Parameters:

```
[22:04] <AGreatJewel> How do I redirect to a url and set some GET params?
some thing like return HTTPFound(location="whatever", params={ params here })
[22:05] <mcdonc> return HTTPFound(location="whatever?a=1&b=2")
[22:06] <AGreatJewel> ok. and I would need to urlencode the entire string?
[22:06] <AGreatJewel> or is that handled automatically
[22:07] <mcdonc> its a url
[22:07] <mcdonc> like you'd type into the browser
```

- Add an example of using a cascade to serve static assets from the root.
- Explore static file return from handler action using wsgiapp2 + fileapp.
- http://blog.dannynavarro.net/2011/01/14/async-web-apps-with-pyramid/
- http://pylonshq.com/pasties/e4b933da7f577c541cc2f2489f825fdd (facebook)
- http://pylonshq.com/pasties/c549d2be586797da17c7fad5ae875372 (twitter)
- http://alexmarandon.com/articles/zodb_bfg_pyramid_notes/
- http://www.serverzen.net/2010/11/20/pyramid-setting-up-debug-werkzeug
- http://groups.google.com/group/pylons-devel/msg/ab58353594b135c9 (pyramid_jinja2 i18n), also https://github.com/Pylons/pyramid_jinja2/pull/14
- Simple asynchronous task queue: http://blog.dannynavarro.net/2011/01/23/async-pyramid-example-done-right/
- Installing Pyramid on Red Hat Enterprise Linux 6 (John Fulton).
- Chameleon main template injected via BeforeRender.

- Hybrid authorization: https://github.com/mmerickel/pyramid_auth_demo
- http://whippleit.blogspot.com/2011/04/pyramid-on-google-app-engine-take-1-for.html
- Custom events: http://blog.dannynavarro.net/2011/06/12/using-custom-events-in-pyramid/
- TicTacToe and Long Polling With Pyramid: http://www.merickel.org/2011/6/21/tictactoe-and-long-polling-with-pyramid/
- Jim Penny's rolodex tutorial: http://jpenny.im/
- Thorsten Lockert's formhelpers/Pyramid example: https://github.com/tholo/formhelpers2
- The Python Ecosystem, an Introduction: http://mirnazim.org/writings/python-ecosystem-introduction/
- Outgrowing Pyramid Handlers: http://michael.merickel.org/2011/8/23/outgrowing-pyramid-handlers/
- Incorporate Google Analytics into a Pyramid Application: http://russell.ballestrini.net/incorporate-google-analytics-into-a-pyramid-application/
- Cookbook docs reorg
  - Move tutorials/overviews to tutorial project and replace with links

Pyramid Glossary

CHAPTER 2

Indices and tables

- genindex
- search

Index