# Pecan Documentation

***Release 1.2.1***

**Jonathan LaCour**

September 27, 2016

Welcome to Pecan, a lean Python web framework inspired by CherryPy, TurboGears, and Pylons. Pecan was originally created by the developers of ShootQ while working at Pictage.

Pecan was created to fill a void in the Python web-framework world – a very lightweight framework that provides object-dispatch style routing. Pecan does not aim to be a "full stack" framework, and therefore includes no out of the box support for things like sessions or databases (although tutorials are included for integrating these yourself in just a few lines of code). Pecan instead focuses on HTTP itself.

Although it is lightweight, Pecan does offer an extensive feature set for building HTTP-based applications, including:

- Object-dispatch for easy routing
- Full support for REST-style controllers
- Extensible security framework
- Extensible template language support
- Extensible JSON support
- Easy Python-based configuration

CHAPTER 1

# Narrative Documentation

## 1.1 Installation

### 1.1.1 Stable Version

We recommend installing Pecan with pip, but you can also try with **easy_install**. Creating a spot in your environment where Pecan can be isolated from other packages is best practice.

To get started with an environment for Pecan, we recommend creating a new virtual environment using virtualenv:

```
$ virtualenv pecan-env
$ cd pecan-env
$ source bin/activate
```

The above commands create a virtual environment and *activate* it. This will isolate Pecan's dependency installations from your system packages, making it easier to debug problems if needed.

Next, let's install Pecan:

```
$ pip install pecan
```

### 1.1.2 Development (Unstable) Version

If you want to run the latest development version of Pecan you will need to install git and clone the repo from GitHub:

```
$ git clone https://github.com/pecan/pecan.git
```

Assuming your virtual environment is still activated, call `setup.py` to install the development version.:

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

**Note:** The `master` development branch is volatile and is generally not recommended for production use.

Alternatively, you can also install from GitHub directly with `pip`.:

```
$ pip install -e git://github.com/pecan/pecan.git#egg=pecan
```

## 1.2 Creating Your First Pecan Application

Let's create a small sample project with Pecan.

**Note:** This guide does not cover the installation of Pecan. If you need instructions for installing Pecan, refer to *Installation*.

### 1.2.1 Base Application Template

Pecan includes a basic template for starting a new project. From your shell, type:

```
$ pecan create test_project
```

This example uses *test_project* as your project name, but you can replace it with any valid Python package name you like.

Go ahead and change into your newly created project directory.:

```
$ cd test_project
```

You'll want to deploy it in "development mode", such that it's available on `sys.path`, yet can still be edited directly from its source distribution:

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

Your new project contain these files:

```
$ ls

-- MANIFEST.in
-- config.py
-- public
|   -- css
|   |   -- style.css
|   -- images
-- setup.cfg
-- setup.py
-- test_project
    -- __init__.py
    -- app.py
    -- controllers
    |   -- __init__.py
    |   -- root.py
    -- model
    |   -- __init__.py
    -- templates
    |   -- error.html
    |   -- index.html
    |   -- layout.html
    -- tests
        -- __init__.py
        -- config.py
        -- test_functional.py
        -- test_units.py
```

The number of files and directories may vary based on the version of Pecan, but the above structure should give you an idea of what to expect.

Let's review the files created by the template.

**public** All your static files (like CSS, Javascript, and images) live here. Pecan comes with a simple file server that serves these static files as you develop.

Pecan application structure generally follows the MVC pattern. The directories under `test_project` encompass your models, controllers and templates.

**test_project/controllers** The container directory for your controller files.

**test_project/templates** All your templates go in here.

**test_project/model** Container for your model files.

Finally, a directory to house unit and integration tests:

**test_project/tests** All of the tests for your application.

The `test_project/app.py` file controls how the Pecan application will be created. This file must contain a `setup_app()` function which returns the WSGI application object. Generally you will not need to modify the `app.py` file provided by the base application template unless you need to customize your app in a way that cannot be accomplished using config. See *Python-Based Configuration* below.

To avoid unneeded dependencies and to remain as flexible as possible, Pecan doesn't impose any database or ORM (Object Relational Mapper). If your project will interact with a database, you can add code to `model/__init__.py` to load database bindings from your configuration file and define tables and ORM definitions.

## 1.2.2 Running the Application

The base project template creates the configuration file with the basic settings you need to run your Pecan application in `config.py`. This file includes the host and port to run the server on, the location where your controllers and templates are stored on disk, and the name of the directory containing any static files.

If you just run **`pecan serve`**, passing `config.py` as the configuration file, it will bring up the development server and serve the app:

```
$ pecan serve config.py
Starting server in PID 000.
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
```

The location for the configuration file and the argument itself are very flexible - you can pass an absolute or relative path to the file.

## 1.2.3 Python-Based Configuration

For ease of use, Pecan configuration files are pure Python–they're even saved as `.py` files.

This is how your default (generated) configuration file should look:

```
# Server Specific Configurations
server = {
    'port': '8080',
    'host': '0.0.0.0'
}

# Pecan Application Configurations
app = {
    'root': '${package}.controllers.root.RootController',
    'modules': ['${package}'],
    'static_root': '%(confdir)s/public',
    'template_path': '%(confdir)s/${package}/templates',
    'debug': True,
    'errors': {
        '404': '/error/404',
        '__force_dict__': True
    }
}

logging = {
    'loggers': {
        'root' : {'level': 'INFO', 'handlers': ['console']},
        '${package}': {'level': 'DEBUG', 'handlers': ['console']}
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        }
    },
    'formatters': {
        'simple': {
            'format': ('%(asctime)s %(levelname)-5.5s [%(name)s]'
                       '[%(threadName)s] %(message)s')
        }
    }
}

# Custom Configurations must be in Python dictionary format::
#
# foo = {'bar':'baz'}
#
# All configurations are accessible at::
# pecan.conf
```

You can also add your own configuration as Python dictionaries.

There's a lot to cover here, so we'll come back to configuration files in a later chapter (*Configuring Pecan Applications*).

### 1.2.4 The Application Root

The **Root Controller** is the entry point for your application. You can think of it as being analogous to your application's root URL path (in our case, `http://localhost:8080/`).

This is how it looks in the project template (`test_project.controllers.root.RootController`):

```
from pecan import expose
from webob.exc import status_map


class RootController(object):

    @expose(generic=True, template='index.html')
    def index(self):
        return dict()

    @index.when(method='POST')
    def index_post(self, q):
        redirect('https://pecan.readthedocs.io/en/latest/search.html?q=%s' % q)

    @expose('error.html')
    def error(self, status):
        try:
            status = int(status)
        except ValueError:
            status = 0
        message = getattr(status_map.get(status), 'explanation', '')
        return dict(status=status, message=message)
```

You can specify additional classes and methods if you need to do so, but for now, let's examine the sample project, controller by controller:

```
@expose(generic=True, template='index.html')
def index(self):
    return dict()
```

The `index()` method is marked as *publicly available* via the `expose()` decorator (which in turn uses the `index.html` template) at the root of the application (http://127.0.0.1:8080/), so any HTTP `GET` that hits the root of your application (`/`) will be routed to this method.

Notice that the `index()` method returns a Python dictionary. This dictionary is used as a namespace to render the specified template (`index.html`) into HTML, and is the primary mechanism by which data is passed from controller to template.

```
@index.when(method='POST')
def index_post(self, q):
    redirect('https://pecan.readthedocs.io/en/latest/search.html?q=%s' % q)
```

The `index_post()` method receives one HTTP `POST` argument (`q`). Because the argument `method` to `@index.when()` has been set to `'POST'`, any HTTP `POST` to the application root (in the example project, a form submission) will be routed to this method.

```
@expose('error.html')
def error(self, status):
    try:
        status = int(status)
    except ValueError:
        status = 0
    message = getattr(status_map.get(status), 'explanation', '')
    return dict(status=status, message=message)
```

Finally, we have the `error()` method, which allows the application to display custom pages for certain HTTP errors (`404`, etc...).

### 1.2.5 Running the Tests For Your Application

Your application comes with a few example tests that you can run, replace, and add to. To run them:

```
$ python setup.py test -q
running test
running egg_info
writing requirements to sam.egg-info/requires.txt
writing sam.egg-info/PKG-INFO
writing top-level names to sam.egg-info/top_level.txt
writing dependency_links to sam.egg-info/dependency_links.txt
reading manifest file 'sam.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
writing manifest file 'sam.egg-info/SOURCES.txt'
running build_ext
....
----------------------------------------------------------------------
Ran 4 tests in 0.009s

OK
```

The tests themselves can be found in the `tests` module in your project.

### 1.2.6 Deploying to a Web Server

Ready to deploy your new Pecan app? Take a look at *Deploying Pecan in Production*.

## 1.3 Controllers and Routing

Pecan uses a routing strategy known as **object-dispatch** to map an HTTP request to a controller, and then the method to call. Object-dispatch begins by splitting the path into a list of components and then walking an object path, starting at the root controller. You can imagine your application's controllers as a tree of objects (branches of the object tree map directly to URL paths).

Let's look at a simple bookstore application:

```python
from pecan import expose

class BooksController(object):
    @expose()
    def index(self):
        return "Welcome to book section."

    @expose()
    def bestsellers(self):
        return "We have 5 books in the top 10."

class CatalogController(object):
    @expose()
    def index(self):
        return "Welcome to the catalog."

    books = BooksController()

class RootController(object):
    @expose()
    def index(self):
        return "Welcome to store.example.com!"

    @expose()
    def hours(self):
        return "Open 24/7 on the web."
    catalog = CatalogController()
```

A request for `/catalog/books/bestsellers` from the online store would begin with Pecan breaking the request up into `catalog`, `books`, and `bestsellers`. Next, Pecan would lookup `catalog` on the root controller. Using the `catalog` object, Pecan would then lookup `books`, followed by `bestsellers`. What if the URL ends in a slash? Pecan will check for an `index` method on the last controller object.

To illustrate further, the following paths:

```
-- /
    -- /hours
    -- /catalog
         -- /catalog/books
            -- /catalog/books/bestsellers
```

route to the following controller methods:

```
-- RootController.index
    -- RootController.hours
    -- CatalogController.index
         -- BooksController.index
            -- BooksController.bestsellers
```

## 1.3.1 Exposing Controllers

You tell Pecan which methods in a class are publically-visible via *`expose()`*. If a method is *not* decorated with *`expose()`*, Pecan will never route a request to it.

*`expose()`* can be used in a variety of ways. The simplest case involves passing no arguments. In this scenario, the controller returns a string representing the HTML response body.

```
from pecan import expose

class RootController(object):
    @expose()
    def hello(self):
        return 'Hello World'
```

A more common use case is to *specify a template and a namespace*:

```
from pecan import expose

class RootController(object):
    @expose('html_template.mako')
    def hello(self):
        return {'msg': 'Hello!'}
```

```
<!-- html_template.mako -->
<html>
    <body>${msg}</body>
</html>
```

Pecan also has built-in support for a special *JSON renderer*, which translates template namespaces into rendered JSON text:

```
from pecan import expose

class RootController(object):
    @expose('json')
    def hello(self):
        return {'msg': 'Hello!'}
```

`expose()` calls can also be stacked, which allows you to serialize content differently depending on how the content is requested:

```
from pecan import expose

class RootController(object):
    @expose('json')
    @expose('text_template.mako', content_type='text/plain')
    @expose('html_template.mako')
    def hello(self):
        return {'msg': 'Hello!'}
```

You'll notice that we called `expose()` three times, with different arguments.

```
@expose('json')
```

The first tells Pecan to serialize the response namespace using JSON serialization when the client requests `/hello.json` or if an `Accept: application/json` header is present.

```
@expose('text_template.mako', content_type='text/plain')
```

The second tells Pecan to use the `text_template.mako` template file when the client requests `/hello.txt` or asks for text/plain via an `Accept` header.

```
@expose('html_template.mako')
```

The third tells Pecan to use the `html_template.mako` template file when the client requests `/hello.html`. If the client requests `/hello`, Pecan will use the `text/html` content type by default; in the absense of an explicit content type, Pecan assumes the client wants HTML.

**See also:**

- *pecan.decorators – Pecan Decorators*

### 1.3.2 Specifying Explicit Path Segments

Occasionally, you may want to use a path segment in your routing that doesn't work with Pecan's declarative approach to routing because of restrictions in Python's syntax. For example, if you wanted to route for a path that includes dashes, such as `/some-path/`, the following is *not* valid Python:

```
class RootController(object):

    @pecan.expose()
    def some-path(self):
        return dict()
```

To work around this, pecan allows you to specify an explicit path segment in the `expose()` decorator:

```
class RootController(object):

    @pecan.expose(route='some-path')
    def some_path(self):
        return dict()
```

In this example, the pecan application will reply with an `HTTP 200` for requests made to `/some-path/`, but requests made to `/some_path/` will yield an `HTTP 404`.

*`route()`* can also be used explicitly as an alternative to the `route` argument in *`expose()`*:

```
class RootController(object):

    @pecan.expose()
    def some_path(self):
        return dict()

pecan.route('some-path', RootController.some_path)
```

Routing to child controllers can be handled simliarly by utilizing *`route()`*:

```
class ChildController(object):

    @pecan.expose()
    def child(self):
        return dict()

class RootController(object):
    pass

pecan.route(RootController, 'child-path', ChildController())
```

In this example, the pecan application will reply with an `HTTP 200` for requests made to `/child-path/child/`.

### 1.3.3 Routing Based on Request Method

The `generic` argument to *`expose()`* provides support for overloading URLs based on the request method. In the following example, the same URL can be serviced by two different methods (one for handling HTTP `GET`, another for HTTP `POST`) using *generic controllers*:

```
from pecan import expose


class RootController(object):

    # HTTP GET /
    @expose(generic=True, template='json')
    def index(self):
        return dict()

    # HTTP POST /
    @index.when(method='POST', template='json')
    def index_POST(self, **kw):
        uuid = create_something()
        return dict(uuid=uuid)
```

### 1.3.4 Pecan's Routing Algorithm

Sometimes, the standard object-dispatch routing isn't adequate to properly route a URL to a controller. Pecan provides several ways to short-circuit the object-dispatch system to process URLs with more control, including the special `_lookup()`, `_default()`, and `_route()` methods. Defining these methods on your controller objects provides additional flexibility for processing all or part of a URL.

### 1.3.5 Routing to Subcontrollers with `_lookup`

The `_lookup()` special method provides a way to process a portion of a URL, and then return a new controller object to route to for the remainder.

A `_lookup()` method may accept one or more arguments, segments of the URL path to be processed (split on `/`). `_lookup()` should also take variable positional arguments representing the rest of the path, and it should include any portion of the path it does not process in its return value. The example below uses a `*remainder` list which will be passed to the returned controller when the object-dispatch algorithm continues.

In addition to being used for creating controllers dynamically, `_lookup()` is called as a last resort, when no other controller method matches the URL and there is no `_default()` method.

```python
from pecan import expose, abort
from somelib import get_student_by_name

class StudentController(object):
    def __init__(self, student):
        self.student = student

    @expose()
    def name(self):
        return self.student.name

class RootController(object):
    @expose()
    def _lookup(self, primary_key, *remainder):
        student = get_student_by_primary_key(primary_key)
        if student:
            return StudentController(student), remainder
        else:
            abort(404)
```

An HTTP GET request to `/8/name` would return the name of the student where `primary_key == 8`.

### 1.3.6 Falling Back with `_default`

The `_default()` method is called as a last resort when no other controller methods match the URL via standard object-dispatch.

```python
from pecan import expose

class RootController(object):
    @expose()
    def english(self):
        return 'hello'

    @expose()
    def french(self):
        return 'bonjour'

    @expose()
    def _default(self):
        return 'I cannot say hello in that language'
```

In the example above, a request to `/spanish` would route to `RootController._default()`.

### 1.3.7 Defining Customized Routing with `_route`

The `_route()` method allows a controller to completely override the routing mechanism of Pecan. Pecan itself uses the `_route()` method to implement its *`RestController`*. If you want to design an alternative routing system on top of Pecan, defining a base controller class that defines a `_route()` method will enable you to have total control.

## 1.4 Interacting with the Request and Response Object

For every HTTP request, Pecan maintains a *thread-local reference* to the request and response object, `pecan.request` and `pecan.response`. These are instances of `pecan.Request` and `pecan.Response`, respectively, and can be interacted with from within Pecan controller code:

```
@pecan.expose()
def login(self):
    assert pecan.request.path == '/login'
    username = pecan.request.POST.get('username')
    password = pecan.request.POST.get('password')

    pecan.response.status = 403
    pecan.response.text = 'Bad Login!'
```

While Pecan abstracts away much of the need to interact with these objects directly, there may be situations where you want to access them, such as:

- Inspecting components of the URI
- Determining aspects of the request, such as the user’s IP address, or the referer header
- Setting specific response headers
- Manually rendering a response body

### 1.4.1 Specifying a Custom Response

Set a specific HTTP response code (such as `203 Non-Authoritative Information`) by modifying the `status` attribute of the response object.

```
from pecan import expose, response

class RootController(object):

    @expose('json')
    def hello(self):
        response.status = 203
        return {'foo': 'bar'}
```

Use the utility function *`abort()`* to raise HTTP errors.

```
from pecan import expose, abort

class RootController(object):

    @expose('json')
    def hello(self):
        abort(404)
```

`abort()` raises an instance of `WSGIHTTPException` which is used by Pecan to render default response bodies for HTTP errors. This exception is stored in the WSGI request environ at `pecan.original_exception`, where it can be accessed later in the request cycle (by, for example, other middleware or *Custom Error Documents*).

If you'd like to return an explicit response, you can do so using `Response`:

```
from pecan import expose, Response

class RootController(object):

    @expose()
    def hello(self):
        return Response('Hello, World!', 202)
```

## 1.4.2 Extending Pecan's Request and Response Object

The request and response implementations provided by WebOb are powerful, but at times, it may be useful to extend application-specific behavior onto your request and response (such as specialized parsing of request headers or customized response body serialization). To do so, define custom classes that inherit from `pecan.Request` and `pecan.Response`, respectively:

```
class MyRequest(pecan.Request):
    pass

class MyResponse(pecan.Response):
    pass
```

and modify your application configuration to use them:

```
from myproject import MyRequest, MyResponse

app = {
    'root' : 'project.controllers.root.RootController',
    'modules' : ['project'],
    'static_root'   : '%(confdir)s/public',
    'template_path' : '%(confdir)s/project/templates',
    'request_cls': MyRequest,
    'response_cls': MyResponse
}
```

## 1.4.3 Mapping Controller Arguments

In Pecan, HTTP `GET` and `POST` variables that are not consumed during the routing process can be passed onto the controller method as arguments.

Depending on the signature of the method, these arguments can be mapped explicitly to arguments:

```
from pecan import expose

class RootController(object):
    @expose()
    def index(self, arg):
        return arg

    @expose()
    def kwargs(self, **kwargs):
        return str(kwargs)
```

```
$ curl http://localhost:8080/?arg=foo
foo
$ curl http://localhost:8080/kwargs?a=1&b=2&c=3
{u'a': u'1', u'c': u'3', u'b': u'2'}
```

or can be consumed positionally:

```
from pecan import expose

class RootController(object):
    @expose()
    def args(self, *args):
        return ','.join(args)
```

```
$ curl http://localhost:8080/args/one/two/three
one,two,three
```

The same effect can be achieved with HTTP `POST` body variables:

```
from pecan import expose

class RootController(object):
    @expose()
    def index(self, arg):
        return arg
```

```
$ curl -X POST "http://localhost:8080/" -H "Content-Type: application/x-www-form-urlencoded" -d "arg=
foo
```

## 1.4.4 Static File Serving

Because Pecan gives you direct access to the underlying `Request`, serving a static file download is as simple as setting the WSGI `app_iter` and specifying the content type:

```
import os
from random import choice

from webob.static import FileIter

from pecan import expose, response


class RootController(object):

    @expose(content_type='image/gif')
    def gifs(self):
        filepath = choice((
            "/path/to/funny/gifs/catdance.gif",
            "/path/to/funny/gifs/babydance.gif",
            "/path/to/funny/gifs/putindance.gif"
        ))
        f = open(filepath, 'rb')
        response.app_iter = FileIter(f)
        response.headers[
            'Content-Disposition'
        ] = 'attachment; filename="%s"' % os.path.basename(f.name)
```

If you don't know the content type ahead of time (for example, if you're retrieving files and their content types from a data store), you can specify it via `response.headers` rather than in the *expose()* decorator:

```
import os
from mimetypes import guess_type

from webob.static import FileIter

from pecan import expose, response


class RootController(object):

    @expose()
    def download(self):
        f = open('/path/to/some/file', 'rb')
        response.app_iter = FileIter(f)
        response.headers['Content-Type'] = guess_type(f.name)
        response.headers[
            'Content-Disposition'
        ] = 'attachment; filename="%s"' % os.path.basename(f.name)
```

### 1.4.5 Handling File Uploads

Pecan makes it easy to handle file uploads via standard multipart forms. Simply define your form with a file input:

```
<form action="/upload" method="POST" enctype="multipart/form-data">
  <input type="file" name="file" />
  <button type="submit">Upload</button>
</form>
```

You can then read the uploaded file off of the request object in your application's controller:

```
from pecan import expose, request

class RootController(object):
    @expose()
    def upload(self):
        assert isinstance(request.POST['file'], cgi.FieldStorage)
        data = request.POST['file'].file.read()
```

### 1.4.6 Thread-Safe Per-Request Storage

For convenience, Pecan provides a Python dictionary on every request which can be accessed and modified in a thread-safe manner throughout the life-cycle of an individual request:

```
pecan.request.context['current_user'] = some_user
print pecan.request.context.items()
```

This is particularly useful in situations where you want to store metadata/context about a request (e.g., in middleware, or per-routing hooks) and access it later (e.g., in controller code).

For more fine-grained control of the request, the underlying WSGI environ for a given Pecan request can be accessed and modified via `pecan.request.environ`.

### 1.4.7 Helper Functions

Pecan also provides several useful helper functions for moving between different routes. The `redirect()` function allows you to issue internal or `HTTP 302` redirects.

**See also:**

The `redirect()` utility, along with several other useful helpers, are documented in *pecan.core – Pecan Core*.

### 1.4.8 Determining the URL for a Controller

Given the ability for routing to be drastically changed at runtime, it is not always possible to correctly determine a mapping between a controller method and a URL.

For example, in the following code that makes use of `_lookup()` to alter the routing depending on a condition:

```
from pecan import expose, abort
from somelib import get_user_region


class DefaultRegionController(object):

    @expose()
    def name(self):
        return "Default Region"

class USRegionController(object):

    @expose()
    def name(self):
        return "US Region"

class RootController(object):
    @expose()
    def _lookup(self, user_id, *remainder):
        if get_user_region(user_id) == 'us':
            return USRegionController(), remainder
        else:
            return DefaultRegionController(), remainder
```

This logic depends on the geolocation of a given user and returning a completely different class given the condition. A helper to determine what URL `USRegionController.name` belongs to would fail to do it correctly.

## 1.5 Templating in Pecan

Pecan includes support for a variety of templating engines and also makes it easy to add support for new template engines. Currently, Pecan supports:

| Template System | Renderer Name |
|---|---|
| Mako | mako |
| Genshi | genshi |
| Kajiki | kajiki |
| Jinja2 | jinja |
| JSON | json |

The default template system is `mako`, but that can be changed by passing the `default_renderer` key in your application's configuration:

```
app = {
    'default_renderer' : 'kajiki',
    # ...
}
```

### 1.5.1 Using Template Renderers

*`pecan.decorators`* defines a decorator called *`expose()`*, which is used to flag a method as a public controller. The *`expose()`* decorator takes a `template` argument, which can be used to specify the path to the template file to use for the controller method being exposed.

```
class MyController(object):
    @expose('path/to/mako/template.html')
    def index(self):
        return dict(message='I am a mako template')
```

*`expose()`* will use the default template engine unless the path is prefixed by another renderer name.

```
@expose('kajiki:path/to/kajiki/template.html')
def my_controller(self):
    return dict(message='I am a kajiki template')
```

**See also:**

- *pecan.decorators – Pecan Decorators*
- *pecan.core – Pecan Core*
- *Controllers and Routing*

### 1.5.2 Overriding Templates

*`override_template()`* allows you to override the template set for a controller method when it is exposed. When *`override_template()`* is called within the body of the controller method, it changes the template that will be used for that invocation of the method.

```
class MyController(object):
    @expose('template_one.html')
    def index(self):
        # ...
        override_template('template_two.html')
        return dict(message='I will now render with template_two.html')
```

### 1.5.3 Manual Rendering

*`render()`* allows you to manually render output using the Pecan templating framework. Pass the template path and values to go into the template, and *`render()`* returns the rendered output as text.

```
@expose()
def controller(self):
    return render('my_template.html', dict(message='I am the namespace'))
```

### 1.5.4 The JSON Renderer

Pecan also provides a JSON renderer, which you can use by exposing a controller method with `@expose('json')`.

**See also:**

- *JSON Serialization*
- *pecan.jsonify – Pecan JSON Support*

### 1.5.5 Defining Custom Renderers

To define a custom renderer, you can create a class that follows the renderer protocol:

```
class MyRenderer(object):
    def __init__(self, path, extra_vars):
        '''
        Your renderer is provided with a path to templates,
        as configured by your application, and any extra
        template variables, also as configured
        '''
        pass

    def render(self, template_path, namespace):
        '''
        Lookup the template based on the path, and render
        your output based upon the supplied namespace
        dictionary, as returned from the controller.
        '''
        return str(namespace)
```

To enable your custom renderer, define a `custom_renderers` key in your application's configuration:

```
app = {
    'custom_renderers' : {
        'my_renderer' : MyRenderer
    },
    # ...
}
```

...and specify the renderer in the *expose()* method:

```
class RootController(object):

    @expose('my_renderer:template.html')
    def index(self):
        return dict(name='Bob')
```

## 1.6 Writing RESTful Web Services with Generic Controllers

Pecan simplifies RESTful web services by providing a way to overload URLs based on the request method. For most API's, the use of *generic controller* definitions give you everything you need to build out robust RESTful interfaces (and is the *recommended* approach to writing RESTful web services in pecan):

```
from pecan import abort, expose

# Note: this is *not* thread-safe.  In real life, use a persistent data store.
```

```
BOOKS = {
    '0': 'The Last of the Mohicans',
    '1': 'Catch-22'
}


class BookController(object):

    def __init__(self, id_):
        self.id_ = id_
        assert self.book

    @property
    def book(self):
        if self.id_ in BOOKS:
            return dict(id=self.id_, name=BOOKS[self.id_])
        abort(404)

    # HTTP GET /<id>/
    @expose(generic=True, template='json')
    def index(self):
        return self.book

    # HTTP PUT /<id>/
    @index.when(method='PUT', template='json')
    def index_PUT(self, **kw):
        BOOKS[self.id_] = kw['name']
        return self.book

    # HTTP DELETE /<id>/
    @index.when(method='DELETE', template='json')
    def index_DELETE(self):
        del BOOKS[self.id_]
        return dict()


class RootController(object):

    @expose()
    def _lookup(self, id_, *remainder):
        return BookController(id_), remainder

    # HTTP GET /
    @expose(generic=True, template='json')
    def index(self):
        return [dict(id=k, name=v) for k, v in BOOKS.items()]

    # HTTP POST /
    @index.when(method='POST', template='json')
    def index_POST(self, **kw):
        id_ = str(len(BOOKS))
        BOOKS[id_] = kw['name']
        return dict(id=id_, name=kw['name'])
```

## 1.7 Writing RESTful Web Services with RestController

For compatability with the TurboGears2 library, Pecan also provides a class-based solution to RESTful routing, *RestController*:

```
from pecan import expose
from pecan.rest import RestController

from mymodel import Book

class BooksController(RestController):

    @expose()
    def get(self, id):
        book = Book.get(id)
        if not book:
            abort(404)
        return book.title
```

### 1.7.1 URL Mapping

By default, *RestController* routes as follows:

| Method | Description | Example Method(s) / URL(s) |
|---|---|---|
| get_one | Display one record. | GET /books/1 |
| get_all | Display all records in a resource. | GET /books/ |
| get | A combo of get_one and get_all. | GET /books/ |
| | | GET /books/1 |
| new | Display a page to create a new resource. | GET /books/new |
| edit | Display a page to edit an existing resource. | GET /books/1/edit |
| post | Create a new record. | POST /books/ |
| put | Update an existing record. | POST /books/1?_method=put |
| | | PUT /books/1 |
| get_delete | Display a delete confirmation page. | GET /books/1/delete |
| delete | Delete an existing record. | POST /books/1?_method=delete |
| | | DELETE /books/1 |

Pecan's *RestController* uses the `?_method=` query string to work around the lack of support for the PUT and DELETE verbs when submitting forms in most current browsers.

In addition to handling REST, the *RestController* also supports the `index()`, `_default()`, and `_lookup()` routing overrides.

**Warning:** If you need to override `_route()`, make sure to call `RestController._route()` at the end of your custom method so that the REST routing described above still occurs.

### 1.7.2 Nesting `RestController`

*RestController* instances can be nested so that child resources receive the parameters necessary to look up parent resources.

For example:

```
from pecan import expose
from pecan.rest import RestController

from mymodel import Author, Book

class BooksController(RestController):

    @expose()
    def get(self, author_id, id):
        author = Author.get(author_id)
        if not author_id:
            abort(404)
        book = author.get_book(id)
        if not book:
            abort(404)
        return book.title

class AuthorsController(RestController):

    books = BooksController()

    @expose()
    def get(self, id):
        author = Author.get(id)
        if not author:
            abort(404)
        return author.name

class RootController(object):

    authors = AuthorsController()
```

Accessing `/authors/1/books/2` invokes `BooksController.get()` with `author_id` set to `1` and `id` set to `2`.

To determine which arguments are associated with the parent resource, Pecan looks at the `get_one()` then `get()` method signatures, in that order, in the parent controller. If the parent resource takes a variable number of arguments, Pecan will pass it everything up to the child resource controller name (e.g., `books` in the above example).

### 1.7.3 Defining Custom Actions

In addition to the default methods defined above, you can add additional behaviors to a *RestController* by defining a special `_custom_actions` dictionary.

For example:

```
from pecan import expose
from pecan.rest import RestController

from mymodel import Book

class BooksController(RestController):

    _custom_actions = {
        'checkout': ['POST']
    }

    @expose()
    def checkout(self, id):
        book = Book.get(id)
        if not book:
            abort(404)
        book.checkout()
```

`_custom_actions` maps method names to the list of valid HTTP verbs for those custom actions. In this case `checkout()` supports `POST`.

## 1.8 Configuring Pecan Applications

Pecan is very easy to configure. As long as you follow certain conventions, using, setting and dealing with configuration should be very intuitive.

Pecan configuration files are pure Python. Each "section" of the configuration is a dictionary assigned to a variable name in the configuration module.

### 1.8.1 Default Values

Below is the complete list of default values the framework uses:

```
server = {
    'port' : '8080',
    'host' : '0.0.0.0'
}

app = {
    'root' : None,
    'modules' : [],
    'static_root' : 'public',
    'template_path' : ''
}
```

### 1.8.2 Application Configuration

The `app` configuration values are used by Pecan to wrap your application into a valid WSGI app. The `app` configuration is specific to your application, and includes values like the root controller class location.

A typical application configuration might look like this:

```
app = {
    'root' : 'project.controllers.root.RootController',
    'modules' : ['project'],
    'static_root'   : '%(confdir)s/public',
    'template_path' : '%(confdir)s/project/templates',
    'debug' : True
}
```

Let's look at each value and what it means:

**modules** A list of modules where pecan will search for applications. Generally this should contain a single item, the name of your project's python package. At least one of the listed modules must contain an `app.setup_app` function which is called to create the WSGI app. In other words, this package should be where your `app.py` file is located, and this file should contain a `setup_app` function.

**root** The root controller of your application. Remember to provide a string representing a Python path to some callable (e.g., `"yourapp.controllers.root.RootController"`).

**static_root** The directory where your static files can be found (relative to the project root). Pecan comes with middleware that can be used to serve static files (like CSS and Javascript files) during development.

**template_path** Points to the directory where your template files live (relative to the project root).

**debug** Enables the ability to display tracebacks in the browser and interactively debug during development.

**Warning:** `app` is a reserved variable name for that section of the configuration, so make sure you don't override it.

**Warning:** Make sure **debug** is *always* set to `False` in production environments.

**See also:**

- *Base Application Template*

### 1.8.3 Server Configuration

Pecan provides some sane defaults. Change these to alter the host and port your WSGI app is served on.

```
server = {
    'port' : '8080',
    'host' : '0.0.0.0'
}
```

### 1.8.4 Additional Configuration

Your application may need access to other configuration values at runtime (like third-party API credentials). Put these settings in their own blocks in your configuration file.

```
twitter = {
    'api_key' : 'FOO',
    'api_secret' : 'SECRET'
}
```

### 1.8.5 Accessing Configuration at Runtime

You can access any configuration value at runtime via `pecan.conf`. This includes custom, application, and server-specific values.

For example, if you needed to specify a global administrator, you could do so like this within the configuration file.

```
administrator = 'foo_bar_user'
```

And it would be accessible in `pecan.conf` as:

```
>>> from pecan import conf
>>> conf.administrator
'foo_bar_user'
```

### 1.8.6 Dictionary Conversion

In certain situations you might want to deal with keys and values, but in strict dictionary form. The *Config* object has a helper method for this purpose that will return a dictionary representation of the configuration, including nested values.

Below is a representation of how you can access the *to_dict()* method and what it returns as a result (shortened for brevity):

```
>>> from pecan import conf
>>> conf
Config({'app': Config({'errors': {}, 'template_path': '', 'static_root': 'public', [...]
>>> conf.to_dict()
{'app': {'errors': {}, 'template_path': '', 'static_root': 'public', [...]
```

### 1.8.7 Prefixing Dictionary Keys

*to_dict()* allows you to pass an optional string argument if you need to prefix the keys in the returned dictionary.

```
>>> from pecan import conf
>>> conf
Config({'app': Config({'errors': {}, 'template_path': '', 'static_root': 'public', [...]
>>> conf.to_dict('prefixed_')
{'prefixed_app': {'prefixed_errors': {}, 'prefixed_template_path': '', 'prefixed_static_root': 'pref
```

### 1.8.8 Dotted Keys, Non-Python Idenfitiers, and Native Dictionaries

Sometimes you want to specify a configuration option that includes dotted keys or is not a valid Python idenfitier, such as `()`. These situations are especially common when configuring Python logging. By passing a special key, `__force_dict__`, individual configuration blocks can be treated as native dictionaries.

```
logging = {
    'root': {'level': 'INFO', 'handlers': ['console']},
    'loggers': {
        'sqlalchemy.engine': {'level': 'INFO', 'handlers': ['console']},
        '__force_dict__': True
    },
    'formatters': {
        'custom': {
            '()': 'my.package.customFormatter'
        }
    }
}

from myapp import conf
assert isinstance(conf.logging.loggers, dict)
assert isinstance(conf.logging.loggers['sqlalchemy.engine'], dict)
```

## 1.9 Security and Authentication

Pecan provides no out-of-the-box support for authentication, but it does give you the necessary tools to handle authentication and authorization as you see fit.

## 1.9.1 secure Decorator Basics

You can wrap entire controller subtrees *or* individual method calls with access controls using the `secure()` decorator.

To decorate a method, use one argument:

```
secure('<check_permissions_method_name>')
```

To secure a class, invoke with two arguments:

```
secure(object_instance, '<check_permissions_method_name>')
```

```
from pecan import expose
from pecan.secure import secure

class HighlyClassifiedController(object):
    pass

class UnclassifiedController(object):
    pass

class RootController(object):

    @classmethod
    def check_permissions(cls):
        if user_is_admin():
            return True
        return False

    @expose()
    def index(self):
      #
      # This controller is unlocked to everyone,
      # and will not run any security checks.
      #
      return dict()

    @secure('check_permissions')
    @expose()
    def topsecret(self):
        #
        # This controller is top-secret, and should
        # only be reachable by administrators.
        #
        return dict()

    highly_classified = secure(HighlyClassifiedController(), 'check_permissions')
    unclassified = UnclassifiedController()
```

## 1.9.2 SecureController

Alternatively, the same functionality can also be accomplished by subclassing Pecan's *SecureController*. Implementations of *SecureController* should extend the *check_permissions()* class method to return `True` if the user has permissions to the controller branch and `False` if they do not.

```
from pecan import expose
from pecan.secure import SecureController, unlocked

```

```
class HighlyClassifiedController(object):
    pass

class UnclassifiedController(object):
    pass

class RootController(SecureController):

    @classmethod
    def check_permissions(cls):
        if user_is_admin():
            return True
        return False

    @expose()
    @unlocked
    def index(self):
      #
      # This controller is unlocked to everyone,
      # and will not run any security checks.
      #
      return dict()

    @expose()
    def topsecret(self):
        #
        # This controller is top-secret, and should
        # only be reachable by administrators.
        #
        return dict()

    highly_classified = HighlyClassifiedController()
    unclassified = unlocked(UnclassifiedController())
```

Also note the use of the `unlocked()` decorator in the above example, which can be used similarly to explicitly unlock a controller for public access without any security checks.

### 1.9.3 Writing Authentication/Authorization Methods

The `check_permissions()` method should be used to determine user authentication and authorization. The code you implement here could range from simple session assertions (the existing user is authenticated as an administrator) to connecting to an LDAP service.

### 1.9.4 More on `secure`

The `secure()` method has several advanced uses that allow you to create robust security policies for your application.

First, you can pass via a string the name of either a class method or an instance method of the controller to use as the `check_permissions()` method. Instance methods are particularly useful if you wish to authorize access to attributes of a model instance. Consider the following example of a basic virtual filesystem.

```
from pecan import expose
from pecan.secure import secure

from myapp.session import get_current_user
from myapp.model import FileObject

class FileController(object):
    def __init__(self, name):
        self.file_object = FileObject(name)

    def read_access(self):
        self.file_object.read_access(get_current_user())

    def write_access(self):
        self.file_object.write_access(get_current_user())

    @secure('write_access')
    @expose()
    def upload_file(self):
        pass

    @secure('read_access')
    @expose()
    def download_file(self):
        pass

class RootController(object):
    @expose()
    def _lookup(self, name, *remainder):
        return FileController(name), remainder
```

The *secure()* method also accepts a function argument. When passing a function, make sure that the function is imported from another file or defined in the same file before the class definition, otherwise you will likely get error during module import.

```
from pecan import expose
from pecan.secure import secure

from myapp.auth import user_authenitcated

class RootController(object):
    @secure(user_authenticated)
    @expose()
    def index(self):
        return 'Logged in'
```

You can also use the *secure()* method to change the behavior of a *SecureController*. Decorating a method or wrapping a subcontroller tells Pecan to use another security function other than the default controller method. This is useful for situations where you want a different level or type of security.

```
from pecan import expose
from pecan.secure import SecureController, secure

from myapp.auth import user_authenticated, admin_user

class ApiController(object):
    pass

class RootController(SecureController):
    @classmethod
    def check_permissions(cls):
        return user_authenticated()
    @classmethod
    def check_api_permissions(cls):
        return admin_user()

    @expose()
    def index(self):
        return 'logged in user'

    api = secure(ApiController(), 'check_api_permissions')
```

In the example above, pecan will *only* call `admin_user()` when a request is made for `/api/`.

### 1.9.5 Multiple Secure Controllers

Secure controllers can be nested to provide increasing levels of security on subcontrollers. In the example below, when a request is made for `/admin/index/`, Pecan first calls *`check_permissions()`* on the `RootController` and then calls *`check_permissions()`* on the `AdminController`.

```
from pecan import expose
from pecan.secure import SecureController

from myapp.auth import user_logged_in, is_admin

class AdminController(SecureController):
    @classmethod
    def check_permissions(cls):
        return is_admin()

    @expose()
    def index(self):
        return 'admin dashboard'

class RootController(SecureController):
    @classmethod
    def check_permissions(cls):
        return user_logged_in

    @expose()
    def index(self):
        return 'user dashboard'
```

## 1.10 Pecan Hooks

Although it is easy to use WSGI middleware with Pecan, it can be hard (sometimes impossible) to have access to Pecan's internals from within middleware. Pecan Hooks are a way to interact with the framework, without having to write separate middleware.

Hooks allow you to execute code at key points throughout the life cycle of your request:

- *`on_route()`*: called before Pecan attempts to route a request to a controller
- *`before()`*: called after routing, but before controller code is run
- *`after()`*: called after controller code has been run
- *`on_error()`*: called when a request generates an exception

### 1.10.1 Implementating a Pecan Hook

In the below example, a simple hook will gather some information about the request and print it to `stdout`.

Your hook implementation needs to import *PecanHook* so it can be used as a base class. From there, you'll want to override the *on_route()*, *before()*, *after()*, or *on_error()* methods to define behavior.

```
from pecan.hooks import PecanHook

class SimpleHook(PecanHook):

    def before(self, state):
        print "\nabout to enter the controller..."

    def after(self, state):
        print "\nmethod: \t %s" % state.request.method
        print "\nresponse: \t %s" % state.response.status
```

*on_route()*, *before()*, and *after()* are each passed a shared state object which includes useful information, such as the request and response objects, and which controller was selected by Pecan's routing:

```
class SimpleHook(PecanHook):

    def on_route(self, state):
        print "\nabout to map the URL to a Python method (controller)..."
        assert state.controller is None  # Routing hasn't occurred yet
        assert isinstance(state.request, webob.Request)
        assert isinstance(state.response, webob.Response)
        assert isinstance(state.hooks, list)  # A list of hooks to apply

    def before(self, state):
        print "\nabout to enter the controller..."
        if state.request.path == '/':
            #
            # `state.controller` is a reference to the actual
            # `@pecan.expose()`-ed controller that will be routed to
            # and used to generate the response body
            #
            assert state.controller.__func__ is RootController.index.__func__
            assert isinstance(state.arguments, inspect.Arguments)
            print state.arguments.args
            print state.arguments.varargs
            print state.arguments.keywords
        assert isinstance(state.request, webob.Request)
        assert isinstance(state.response, webob.Response)
        assert isinstance(state.hooks, list)
```

*on_error()* is passed a shared state object **and** the original exception. If an *on_error()* handler returns a Response object, this response will be returned to the end user and no furthur *on_error()* hooks will be executed:

```
class CustomErrorHook(PecanHook):

    def on_error(self, state, exc):
        if isinstance(exc, SomeExceptionType):
            return webob.Response('Custom Error!', status=500)
```

### 1.10.2 Attaching Hooks

Hooks can be attached in a project-wide manner by specifying a list of hooks in your project's configuration file.

```
app = {
    'root' : '...'
    # ...
    'hooks': lambda: [SimpleHook()]
}
```

Hooks can also be applied selectively to controllers and their sub-controllers using the `__hooks__` attribute on one or more controllers and subclassing *`HookController`*.

```
from pecan import expose
from pecan.hooks import HookController
from my_hooks import SimpleHook

class SimpleController(HookController):

    __hooks__ = [SimpleHook()]

    @expose('json')
    def index(self):
        print "DO SOMETHING!"
        return dict()
```

Now that `SimpleHook` is included, let's see what happens when we run the app and browse the application from our web browser.

```
pecan serve config.py
serving on 0.0.0.0:8080 view at http://127.0.0.1:8080

about to enter the controller...
DO SOMETHING!
method:      GET
response:    200 OK
```

Hooks can be inherited from parent class or mixins. Just make sure to subclass from *`HookController`*.

```
from pecan import expose
from pecan.hooks import PecanHook, HookController

class ParentHook(PecanHook):

    priority = 1

    def before(self, state):
        print "\nabout to enter the parent controller..."

class CommonHook(PecanHook):

    priority = 2

    def before(self, state):
        print "\njust a common hook..."

class SubHook(PecanHook):

    def before(self, state):
        print "\nabout to enter the subcontroller..."

class SubMixin(object):
    __hooks__ = [SubHook()]

# We'll use the same instance for both controllers,
# to avoid double calls
common = CommonHook()

class SubController(HookController, SubMixin):

    __hooks__ = [common]

    @expose('json')
    def index(self):
        print "\nI AM THE SUB!"
        return dict()

class RootController(HookController):

    __hooks__ = [common, ParentHook()]

    @expose('json')
    def index(self):
        print "\nI AM THE ROOT!"
        return dict()

    sub = SubController()
```

Let's see what happens when we run the app. First loading the root controller:

```
pecan serve config.py
serving on 0.0.0.0:8080 view at http://127.0.0.1:8080

GET / HTTP/1.1" 200

about to enter the parent controller...

just a common hook

I AM THE ROOT!
```

Then loading the sub controller:

```
pecan serve config.py
serving on 0.0.0.0:8080 view at http://127.0.0.1:8080

GET /sub HTTP/1.1" 200

about to enter the parent controller...

just a common hook

about to enter the subcontroller...

I AM THE SUB!
```

**Note:** Make sure to set proper priority values for nested hooks in order to get them executed in the desired order.

**Warning:** Two hooks of the same type will be added/executed twice, if passed as different instances to a parent and a child controller. If passed as one instance variable - will be invoked once for both controllers.

## 1.10.3 Hooks That Come with Pecan

Pecan includes some hooks in its core. This section will describe their different uses, how to configure them, and examples of common scenarios.

### RequestViewerHook

This hook is useful for debugging purposes. It has access to every attribute the `response` object has plus a few others that are specific to the framework.

There are two main ways that this hook can provide information about a request:

1. Terminal or logging output (via an file-like stream like `stdout`)
2. Custom header keys in the actual response.

By default, both outputs are enabled.

**See also:**

- *pecan.hooks – Pecan Hooks*

#### Configuring RequestViewerHook

There are a few ways to get this hook properly configured and running. However, it is useful to know that no actual configuration is needed to have it up and running.

By default it will output information about these items:

- path : Displays the url that was used to generate this response
- status : The response from the server (e.g. '200 OK')
- method : The method for the request (e.g. 'GET', 'POST', 'PUT or 'DELETE')
- controller : The actual controller method in Pecan responsible for the response
- params : A list of tuples for the params passed in at request time
- hooks : Any hooks that are used in the app will be listed here.

The default configuration will show those values in the terminal via `stdout` and it will also add them to the response headers (in the form of `X-Pecan-item_name`).

This is how the terminal output might look for a */favicon.ico* request:

```
path          - /favicon.ico
status        - 404 Not Found
method        - GET
controller    - The resource could not be found.
params        - []
hooks         - ['RequestViewerHook']
```

In the above case, the file was not found, and the information was printed to *stdout*. Additionally, the following headers would be present in the HTTP response:

```
X-Pecan-path         /favicon.ico
X-Pecan-status       404 Not Found
X-Pecan-method       GET
X-Pecan-controller   The resource could not be found.
X-Pecan-params       []
X-Pecan-hooks        ['RequestViewerHook']
```

The configuration dictionary is flexible (none of the keys are required) and can hold two keys: `items` and `blacklist`.

This is how the hook would look if configured directly (shortened for brevity):

```
...
'hooks': lambda: [
    RequestViewerHook({'items':['path']})
]
```

### Modifying Output Format

The `items` list specify the information that the hook will return. Sometimes you will need a specific piece of information or a certain bunch of them according to the development need so the defaults will need to be changed and a list of items specified.

**Note:** When specifying a list of items, this list overrides completely the defaults, so if a single item is listed, only that item will be returned by the hook.

The hook has access to every single attribute the request object has and not only to the default ones that are displayed, so you can fine tune the information displayed.

These is a list containing all the possible attributes the hook has access to (directly from *webob*):

| | |
|---|---|
| accept | make_tempfile |
| accept_charset | max_forwards |
| accept_encoding | method |
| accept_language | params |
| application_url | path |
| as_string | path_info |
| authorization | path_info_peek |
| blank | path_info_pop |
| body | path_qs |
| body_file | path_url |
| body_file_raw | postvars |
| body_file_seekable | pragma |
| cache_control | query_string |
| call_application | queryvars |
| charset | range |
| content_length | referer |
| content_type | referrer |
| cookies | relative_url |

Continued on next page

Table 1.1 – continued from previous page

| | |
|---|---|
| copy | remote_addr |
| copy_body | remote_user |
| copy_get | remove_conditional_headers |
| date | request_body_tempfile_limit |
| decode_param_names | scheme |
| environ | script_name |
| from_file | server_name |
| from_string | server_port |
| get_response | str_GET |
| headers | str_POST |
| host | str_cookies |
| host_url | str_params |
| http_version | str_postvars |
| if_match | str_queryvars |
| if_modified_since | unicode_errors |
| if_none_match | upath_info |
| if_range | url |
| if_unmodified_since | urlargs |
| is_body_readable | urlvars |
| is_body_seekable | uscript_name |
| is_xhr | user_agent |
| make_body_seekable | |

And these are the specific ones from Pecan and the hook:

- controller
- hooks
- params (params is actually available from *webob* but it is parsed by the hook for redability)

### Blacklisting Certain Paths

Sometimes it's annoying to get information about *every* single request. To limit the output, pass the list of URL paths for which you do not want data as the `blacklist`.

The matching is done at the start of the URL path, so be careful when using this feature. For example, if you pass a configuration like this one:

```
{ 'blacklist': ['/f'] }
```

It would not show *any* url that starts with `f`, effectively behaving like a globbing regular expression (but not quite as powerful).

For any number of blocking you may need, just add as many items as wanted:

```
{ 'blacklist' : ['/favicon.ico', '/javascript', '/images'] }
```

Again, the `blacklist` key can be used along with the `items` key or not (it is not required).

## 1.11 JSON Serialization

Pecan includes a simple, easy-to-use system for generating and serving JSON. To get started, create a file in your project called `json.py` and import it in your project's `app.py`.

Your `json` module will contain a series of rules for generating JSON from objects you return in your controller.

Let's say that we have a controller in our Pecan application which we want to use to return JSON output for a `User` object:

```
from myproject.lib import get_current_user

class UsersController(object):
    @expose('json')
    def current_user(self):
        '''
        return an instance of myproject.model.User which represents
        the current authenticated user
        '''
        return get_current_user()
```

In order for this controller to function, Pecan will need to know how to convert the `User` object into data types compatible with JSON. One way to tell Pecan how to convert an object into JSON is to define a rule in your `json.py`:

```
from pecan.jsonify import jsonify
from myproject import model

@jsonify.register(model.User)
def jsonify_user(user):
    return dict(
        name = user.name,
        email = user.email,
        birthday = user.birthday.isoformat()
    )
```

In this example, when an instance of the `model.User` class is returned from a controller which is configured to return JSON, the `jsonify_user()` rule will be called to convert the object to JSON-compatible data. Note that the rule does not generate a JSON string, but rather generates a Python dictionary which contains only JSON friendly data types.

Alternatively, the rule can be specified on the object itself, by specifying a `__json__()` method in the class:

```
class User(object):
    def __init__(self, name, email, birthday):
        self.name = name
        self.email = email
        self.birthday = birthday

    def __json__(self):
        return dict(
            name = self.name,
            email = self.email,
            birthday = self.birthday.isoformat()
        )
```

The benefit of using a `json.py` module is having all of your JSON rules defined in a central location, but some projects prefer the simplicity of keeping the JSON rules attached directly to their model objects.

## 1.12 Context/Thread-Locals vs. Explicit Argument Passing

In any pecan application, the module-level `pecan.request` and `pecan.response` are proxy objects that always refer to the request and response being handled in the current thread.

This *thread locality* ensures that you can safely access a global reference to the current request and response in a multi-threaded environment without constantly having to pass object references around in your code; it's a feature of pecan that makes writing traditional web applications easier and less verbose.

Some people feel thread-locals are too implicit or magical, and that explicit reference passing is much clearer and more maintainable in the long run. Additionally, the default implementation provided by pecan uses `threading.local()` to associate these context-local proxy objects with the *thread identifier* of the current server thread. In asynchronous server models - where lots of tasks run for short amounts of time on a *single* shared thread - supporting this mechanism involves monkeypatching `threading.local()` to behave in a greenlet-local manner.

### 1.12.1 Disabling Thread-Local Proxies

If you're certain that you *do not* want to utilize context/thread-locals in your project, you can do so by passing the argument `use_context_locals=False` in your application's configuration file:

```
app = {
    'root': 'project.controllers.root.RootController',
    'modules': ['project'],
    'static_root': '%(confdir)s/public',
    'template_path': '%(confdir)s/project/templates',
    'debug': True,
    'use_context_locals': False
}
```

Additionally, you'll need to update **all** of your pecan controllers to accept positional arguments for the current request and response:

```
class RootController(object):

    @pecan.expose('json')
    def index(self, req, resp):
        return dict(method=req.method) # path: /

    @pecan.expose()
    def greet(self, req, resp, name):
        return name  # path: /greet/joe
```

It is *imperative* that the request and response arguments come **after** `self` and before any positional form arguments.

## 1.13 Command Line Pecan

Any Pecan application can be controlled and inspected from the command line using the built-in **`pecan`** command. The usage examples of **`pecan`** in this document are intended to be invoked from your project's root directory.

### 1.13.1 Serving a Pecan App For Development

Pecan comes bundled with a lightweight WSGI development server based on Python's `wsgiref.simple_server` module.

Serving your Pecan app is as simple as invoking the `pecan serve` command:

```
$ pecan serve config.py
Starting server in PID 000.
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
```

and then visiting it in your browser.

The server `host` and `port` in your configuration file can be changed as described in *Server Configuration*.

### 1.13.2 Reloading Automatically as Files Change

Pausing to restart your development server as you work can be interruptive, so **`pecan serve`** provides a `--reload` flag to make life easier.

To provide this functionality, Pecan makes use of the Python watchdog library. You'll need to install it for development use before continuing:

```
$ pip install watchdog
Downloading/unpacking watchdog
...
Successfully installed watchdog
```

```
$ pecan serve --reload config.py
Monitoring for changes...
Starting server in PID 000.
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
```

As you work, Pecan will listen for any file or directory modification events in your project and silently restart your server process in the background.

### 1.13.3 The Interactive Shell

Pecan applications also come with an interactive Python shell which can be used to execute expressions in an environment very similar to the one your application runs in. To invoke an interactive shell, use the `pecan shell` command:

```
$ pecan shell config.py
Pecan Interactive Shell
Python 2.7.1 (r271:86832, Jul 31 2011, 19:30:53)
[GCC 4.2.1 (Based on Apple Inc. build 5658)

  The following objects are available:
  wsgiapp    - This project's WSGI App instance
  conf       - The current configuration
  app        - webtest.TestApp wrapped around wsgiapp

>>> conf
Config({
    'app': Config({
        'root': 'myapp.controllers.root.RootController',
        'modules': ['myapp'],
        'static_root': '/Users/somebody/myapp/public',
        'template_path': '/Users/somebody/myapp/project/templates',
        'errors': {'404': '/error/404'},
        'debug': True
    }),
    'server': Config({
        'host': '0.0.0.0',
        'port': '8080'
    })
})
>>> app
<webtest.app.TestApp object at 0x101a830>
>>> app.get('/')
<200 OK text/html body='<html>\n ...\n\n'/936>
```

Press `Ctrl-D` to exit the interactive shell (or `Ctrl-Z` on Windows).

#### Using an Alternative Shell

`pecan shell` has optional support for the IPython and bpython alternative shells, each of which can be specified with the `--shell` flag (or its abbreviated alias, `-s`), e.g.,

```
$ pecan shell --shell=ipython config.py
$ pecan shell -s bpython config.py
```

### 1.13.4 Configuration from an environment variable

In all the examples shown, you will see that the **pecan** commands accepted a file path to the configuration file. An alternative to this is to specify the configuration file in an environment variable (`PECAN_CONFIG`).

This is completely optional; if a file path is passed in explicitly, Pecan will honor that before looking for an environment variable.

For example, to serve a Pecan application, a variable could be exported and subsequently be re-used when no path is passed in.

```
$ export PECAN_CONFIG=/path/to/app/config.py
$ pecan serve
Starting server in PID 000.
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
```

Note that the path needs to reference a valid pecan configuration file, otherwise the command will error out with a message indicating that the path is invalid (for example, if a directory is passed in).

If `PECAN_CONFIG` is not set and no configuration is passed in, the command will error out because it will not be able to locate a configuration file.

### 1.13.5 Extending `pecan` with Custom Commands

While the commands packaged with Pecan are useful, the real utility of its command line toolset lies in its extensibility. It's convenient to be able to write a Python script that can work "in a Pecan environment" with access to things like your application's parsed configuration file or a simulated instance of your application itself (like the one provided in the `pecan shell` command).

#### Writing a Custom Pecan Command

As an example, let's create a command that can be used to issue a simulated HTTP GET to your application and print the result. Its invocation from the command line might look something like this:

```
$ pecan wget config.py /path/to/some/resource
```

Let's say you have a distribution with a package in it named `myapp`, and that within this package is a `wget.py` module:

```
# myapp/myapp/wget.py
import pecan
from webtest import TestApp

class GetCommand(pecan.commands.BaseCommand):
    '''
    Issues a (simulated) HTTP GET and returns the request body.
    '''

    arguments = pecan.commands.BaseCommand.arguments + ({
        'name': 'path',
        'help': 'the URI path of the resource to request'
    },)

    def run(self, args):
        super(GetCommand, self).run(args)
        app = TestApp(self.load_app())
        print app.get(args.path).body
```

Let's analyze this piece-by-piece.

### Overriding the `run` Method

First, we're subclassing *`BaseCommand`* and extending the *`run()`* method to:

- Load a Pecan application - *`load_app()`*
- Wrap it in a fake WGSI environment - `TestApp`
- Issue an HTTP GET request against it - `get()`

### Defining Custom Arguments

The `arguments` class attribute is used to define command line arguments specific to your custom command. You'll notice in this example that we're *adding* to the arguments list provided by *`BaseCommand`* (which already provides an argument for the `config_file`), rather than overriding it entirely.

The format of the `arguments` class attribute is a `tuple` of dictionaries, with each dictionary representing an argument definition in the same format accepted by Python's `argparse` module (more specifically, `add_argument()`). By providing a list of arguments in this format, the **pecan** command can include your custom commands in the help and usage output it provides.

```
$ pecan -h
usage: pecan [-h] command ...

positional arguments:
  command
    wget        Issues a (simulated) HTTP GET and returns the request body
    serve       Open an interactive shell with the Pecan app loaded
    ...

$ pecan wget -h
```

```
usage: pecan wget [-h] config_file path
$ pecan wget config.py /path/to/some/resource
```

Additionally, you'll notice that the first line of the docstring from `GetCommand` – `Issues a (simulated) HTTP GET and returns the request body` – is automatically used to describe the **wget** command in the output for `$ pecan -h`. Following this convention allows you to easily integrate a summary for your command into the Pecan command line tool.

### Registering a Custom Command

Now that you've written your custom command, you'll need to tell your distribution's `setup.py` about its existence and reinstall. Within your distribution's `setup.py` file, you'll find a call to `setup()`.

```
# myapp/setup.py
...
setup(
    name='myapp',
    version='0.1',
    author='Joe Somebody',
    ...
)
```

Assuming it doesn't exist already, we'll add the `entry_points` argument to the `setup()` call, and define a `[pecan.command]` definition for your custom command:

```
# myapp/setup.py
...
setup(
    name='myapp',
    version='0.1',
    author='Joe Somebody',
    ...
    entry_points="""
    [pecan.command]
    wget = myapp.wget:GetCommand
    """
)
```

Once you've done this, reinstall your project in development to register the new entry point.

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

Then give it a try.

```
$ pecan wget config.py /path/to/some/resource
```

## 1.14 Developing Pecan Applications Locally

### 1.14.1 Reloading Automatically as Files Change

Pausing to restart your development server as you work can be interruptive, so **pecan serve** provides a `--reload` flag to make life easier.

To provide this functionality, Pecan makes use of the Python watchdog library. You'll need to install it for development use before continuing:

```
$ pip install watchdog
Downloading/unpacking watchdog
...
Successfully installed watchdog
```

```
$ pecan serve --reload config.py
Monitoring for changes...
Starting server in PID 000.
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
```

As you work, Pecan will listen for any file or directory modification events in your project and silently restart your server process in the background.

### 1.14.2 Debugging Pecan Applications

Pecan comes with simple debugging middleware for helping diagnose problems in your applications. To enable the debugging middleware, simply set the `debug` flag to `True` in your configuration file:

```
app = {
    ...
    'debug': True,
    ...
}
```

Once enabled, the middleware will automatically catch exceptions raised by your application and display the Python stack trace and WSGI environment in your browser when runtime exceptions are raised.

To improve debugging, including support for an interactive browser-based console, Pecan makes use of the Python *backlash <https://pypi.python.org/pypi/backlash>* library. You'll need to install it for development use before continuing:

```
$ pip install backlash
Downloading/unpacking backlash
...
Successfully installed backlash
```

### 1.14.3 Serving Static Files

Pecan comes with simple file serving middleware for serving CSS, Javascript, images, and other static files. You can configure it by ensuring that the following options are specified in your configuration file:

```
app = {
    ...
    'debug': True,
    'static_root': '%(confdir)/public
}
```

where `static_root` is an absolute pathname to the directory in which your static files live. For convenience, the path may include the `%(confdir)` variable, which Pecan will substitute with the absolute path of your configuration file at runtime.

**Note:** In production, `app.debug` should *never* be set to `True`, so you'll need to serve your static files via your production web server.

## 1.15 Deploying Pecan in Production

There are a variety of ways to deploy a Pecan project to a production environment. The following examples are meant to provide *direction*, not explicit instruction; deployment is usually heavily dependent upon the needs and goals of individual applications, so your mileage will probably vary.

**Note:** While Pecan comes packaged with a simple server *for development use* (**`pecan serve`**), using a *production-ready* server similar to the ones described in this document is **very highly encouraged**.

### 1.15.1 Installing Pecan

A few popular options are available for installing Pecan in production environments:

- Using setuptools. Manage Pecan as a dependency in your project's `setup.py` file so that it's installed alongside your project (e.g., `python /path/to/project/setup.py install`). The default Pecan project described in *Creating Your First Pecan Application* facilitates this by including Pecan as a dependency for your project.
- Using pip. Use `pip freeze` and `pip install` to create and install from a `requirements.txt` file for your project.
- Via the manual instructions found in *Installation*.

**Note:** Regardless of the route you choose, it's highly recommended that all deployment installations be done in a Python virtual environment.

### 1.15.2 Disabling Debug Mode

**Warning:** One of the most important steps to take before deploying a Pecan app into production is to ensure that you have disabled **Debug Mode**, which provides a development-oriented debugging environment for tracebacks encountered at runtime. Failure to disable this development tool in your production environment *will* result in serious security issues. In your production configuration file, ensure that `debug` is set to `False`.

```
# myapp/production_config.py
app = {
    ...
    'debug': False
}
```

### 1.15.3 Pecan and WSGI

WSGI is a Python standard that describes a standard interface between servers and an application. Any Pecan application is also known as a "WSGI application" because it implements the WSGI interface, so any server that is "WSGI compatible" may be used to serve your application. A few popular examples are:

- mod_wsgi
- uWSGI

- Gunicorn
- waitress
- CherryPy

Generally speaking, the WSGI entry point to any Pecan application can be generated using *deploy()*:

```
from pecan.deploy import deploy
application = deploy('/path/to/some/app/config.py')
```

### 1.15.4 Considerations for Static Files

Pecan comes with static file serving (e.g., CSS, Javascript, images) middleware which is **not** recommended for use in production.

In production, Pecan doesn't serve media files itself; it leaves that job to whichever web server you choose.

For serving static files in production, it's best to separate your concerns by serving static files separately from your WSGI application (primarily for performance reasons). There are several popular ways to accomplish this. Here are two:

1. Set up a proxy server (such as nginx, cherokee, *CherryPy*, or lighttpd) to serve static files and proxy application requests through to your WSGI application:

```
     <HTTP Client> --- <Production/Proxy Server>, e.g., Apache, nginx, cherokee (0.0.0.0:80) --- <Sta
                        |
                        -- <WSGI Server> Instance e.g., mod_wsgi, Gunicorn, uWSGI (127.0.0.1:5000 or
                        -- <WSGI Server> Instance e.g., mod_wsgi, Gunicorn, uWSGI (127.0.0.1:5001 or
                        -- <WSGI Server> Instance e.g., mod_wsgi, Gunicorn, uWSGI (127.0.0.1:5002 or
                        -- <WSGI Server> Instance e.g., mod_wsgi, Gunicorn, uWSGI (127.0.0.1:5003 or
```

2. Serve static files via a separate service, virtual host, or CDN.

### 1.15.5 Common Recipes

#### Apache + mod_wsgi

mod_wsgi is a popular Apache module which can be used to host any WSGI-compatible Python application (including your Pecan application).

To get started, check out the installation and configuration documentation for mod_wsgi.

For the sake of example, let's say that our project, `simpleapp`, lives at `/var/www/simpleapp`, and that a virtualenv has been created at `/var/www/venv` with any necessary dependencies installed (including Pecan). Additionally, for security purposes, we've created a user, `user1`, and a group, `group1` to execute our application under.

The first step is to create a `.wsgi` file which mod_wsgi will use as an entry point for your application:

```
# /var/www/simpleapp/app.wsgi
from pecan.deploy import deploy
application = deploy('/var/www/simpleapp/config.py')
```

Next, add Apache configuration for your application. Here's a simple example:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess simpleapp user=user1 group=group1 threads=5 python-path=/var/www/venv/lib/pytho
    WSGIScriptAlias / /var/www/simpleapp/app.wsgi

    <Directory /var/www/simpleapp/>
        WSGIProcessGroup simpleapp
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

For more instructions and examples of mounting WSGI applications using mod_wsgi, consult the mod_wsgi Documentation.

Finally, restart Apache and give it a try.

### uWSGI

uWSGI is a fast, self-healing and developer/sysadmin-friendly application container server coded in pure C. It uses the uwsgi protocol, but can speak other protocols as well (http, fastcgi...).

Running Pecan applications with uWSGI is a snap:

```
$ pip install uwsgi
$ pecan create simpleapp && cd simpleapp
$ python setup.py develop
$ uwsgi --http-socket :8080 --venv /path/to/virtualenv --pecan config.py
```

or using a Unix socket (that nginx, for example, could be configured to proxy to):

```
$ uwsgi -s /tmp/uwsgi.sock --venv /path/to/virtualenv --pecan config.py
```

### Gunicorn

Gunicorn, or "Green Unicorn", is a WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project. It supports both eventlet and greenlet.

Running a Pecan application on Gunicorn is simple. Let's walk through it with Pecan's default project:

```
$ pip install gunicorn
$ pecan create simpleapp && cd simpleapp
$ python setup.py develop
$ gunicorn_pecan config.py
```

### CherryPy

CherryPy offers a pure Python HTTP/1.1-compliant WSGI thread-pooled web server. It can support Pecan applications easily and even serve static files like a production server would do.

The examples that follow are geared towards using CherryPy as the server in charge of handling a Pecan app along with serving static files.

```
$ pip install cherrypy
$ pecan create simpleapp && cd simpleapp
$ python setup.py develop
```

To run with CherryPy, the easiest approach is to create a script in the root of the project (alongside `setup.py`), so that we can describe how our example application should be served. This is how the script (named `run.py`) looks:

```python
import os
import cherrypy
from cherrypy import wsgiserver

from pecan import deploy

simpleapp_wsgi_app = deploy('/path/to/production_config.py')

public_path = os.path.abspath(os.path.join(os.path.dirname(__file__), 'public'))

# A dummy class for our Root object
# necessary for some CherryPy machinery
class Root(object):
    pass

def make_static_config(static_dir_name):
    """
    All custom static configurations are set here, since most are common, it
    makes sense to generate them just once.
    """
    static_path = os.path.join('/', static_dir_name)
    path = os.path.join(public_path, static_dir_name)
    configuration = {
        static_path: {
            'tools.staticdir.on': True,
            'tools.staticdir.dir': path
        }
    }
    return cherrypy.tree.mount(Root(), '/', config=configuration)

# Assuming your app has media on different paths, like 'css', and 'images'
application = wsgiserver.WSGIPathInfoDispatcher({
    '/': simpleapp_wsgi_app,
    '/css': make_static_config('css'),
    '/images': make_static_config('images')
    }
)

server = wsgiserver.CherryPyWSGIServer(('0.0.0.0', 8080), application,
server_name='simpleapp')

try:
    server.start()
except KeyboardInterrupt:
    print "Terminating server..."
    server.stop()
```

To start the server, simply call it with the Python executable:

```
$ python run.py
```

## 1.16 Logging

Pecan uses the Python standard library's `logging` module by passing logging configuration options into the logging.config.dictConfig function. The full documentation for the `dictConfig()` format is the best source of information for logging configuration, but to get you started, this chapter will provide you with a few simple examples.

### 1.16.1 Configuring Logging

Sample logging configuration is provided with the quickstart project introduced in *Creating Your First Pecan Application*:

```
$ pecan create myapp
```

The default configuration defines one handler and two loggers.

```
# myapp/config.py

app = { ... }
server = { ... }

logging = {
    'root' : {'level': 'INFO', 'handlers': ['console']},
    'loggers': {
        'myapp': {'level': 'DEBUG', 'handlers': ['console']}
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        }
    },
    'formatters': {
        'simple': {
            'format': ('%(asctime)s %(levelname)-5.5s [%(name)s]'
                       '[%(threadName)s] %(message)s')
        }
    }
}
```

- `console` logs messages to `stderr` using the `simple` formatter.
- `myapp` logs messages sent at a level above or equal to `DEBUG` to the `console` handler
- `root` logs messages at a level above or equal to the `INFO` level to the `console` handler

### 1.16.2 Writing Log Messages in Your Application

The logger named `myapp` is reserved for your usage in your Pecan application.

Once you have configured your logging, you can place logging calls in your code. Using the logging framework is very simple.

```
# myapp/myapp/controllers/root.py
from pecan import expose
import logging
```

```
logger = logging.getLogger(__name__)

class RootController(object):

    @expose()
    def index(self):
        if bad_stuff():
            logger.error('Uh-oh!')
        return dict()
```

### 1.16.3 Logging to Files and Other Locations

Python's `logging` library defines a variety of handlers that assist in writing logs to file. A few interesting ones are:

- `FileHandler` - used to log messages to a file on the filesystem
- `RotatingFileHandler` - similar to `FileHandler`, but also rotates logs periodically
- `SysLogHandler` - used to log messages to a UNIX syslog
- `SMTPHandler` - used to log messages to an email address via SMTP

Using any of them is as simple as defining a new handler in your application's `logging` block and assigning it to one of more loggers.

### 1.16.4 Logging Requests with Paste Translogger

Paste (which is not included with Pecan) includes the `TransLogger` middleware for logging requests in Apache Combined Log Format. Combined with file-based logging, TransLogger can be used to create an `access.log` file similar to `Apache`.

To add this middleware, modify your the `setup_app` method in your project's `app.py` as follows:

```
# myapp/myapp/app.py
from pecan import make_app
from paste.translogger import TransLogger

def setup_app(config):
    # ...
    app = make_app(
        config.app.root
        # ...
    )
    app = TransLogger(app, setup_console_handler=False)
    return app
```

By default, `TransLogger` creates a logger named `wsgi`, so you'll need to specify a new (file-based) handler for this logger in our Pecan configuration file:

```
# myapp/config.py

app = { ... }
server = { ... }

logging = {
    'loggers': {
        # ...
        'wsgi': {'level': 'INFO', 'handlers': ['logfile'], 'qualname': 'wsgi'}
    },
    'handlers': {
        # ...
        'logfile': {
            'class': 'logging.FileHandler',
            'filename': '/etc/access.log',
            'level': 'INFO',
            'formatter': 'messageonly'
        }
    },
    'formatters': {
        # ...
        'messageonly': {'format': '%(message)s'}
    }
}
```

## 1.17 Testing Pecan Applications

Tests can live anywhere in your Pecan project as long as the test runner can discover them. Traditionally, they exist in a package named `myapp.tests`.

The suggested mechanism for unit and integration testing of a Pecan application is the `unittest` module.

### 1.17.1 Test Discovery and Other Tools

Tests for a Pecan project can be invoked as simply as `python setup.py test`, though it's possible to run your tests with different discovery and automation tools. In particular, Pecan projects are known to work well with nose, pytest, and tox.

### 1.17.2 Writing Functional Tests with WebTest

A **unit test** typically relies on "mock" or "fake" objects to give the code under test enough context to run. In this way, only an individual unit of source code is tested.

A healthy suite of tests combines **unit tests** with **functional tests**. In the context of a Pecan application, functional tests can be written with the help of the `webtest` library. In this way, it is possible to write tests that verify the behavior of an HTTP request life cycle from the controller routing down to the HTTP response. The following is an example that is similar to the one included with Pecan's quickstart project.

```
# myapp/myapp/tests/__init__.py

import os
from unittest import TestCase
from pecan import set_config
from pecan.testing import load_test_app

class FunctionalTest(TestCase):
    """
    Used for functional tests where you need to test your
    literal application and its integration with the framework.
    """

    def setUp(self):
        self.app = load_test_app(os.path.join(
            os.path.dirname(__file__),
            'config.py'
        ))

    def tearDown(self):
        set_config({}, overwrite=True)
```

The testing utility included with Pecan, *pecan.testing.load_test_app()*, can be passed a file path representing a Pecan configuration file, and will return an instance of the application, wrapped in a `TestApp` environment.

From here, it's possible to extend the `FunctionalTest` base class and write tests that issue simulated HTTP requests.

```
class TestIndex(FunctionalTest):

    def test_index(self):
        resp = self.app.get('/')
        assert resp.status_int == 200
        assert 'Hello, World' in resp.body
```

**See also:**

See the `webtest` documentation for further information about the methods available to a `TestApp` instance.

## 1.17.3 Special Testing Variables

Sometimes it's not enough to make assertions about the response body of certain requests. To aid in inspection, Pecan applications provide a special set of "testing variables" to any `TestResponse` object.

Let's suppose that your Pecan applicaton had some controller which took a `name` as an optional argument in the URL.

```
# myapp/myapp/controllers/root.py
from pecan import expose

class RootController(object):

    @expose('index.html')
    def index(self, name='Joe'):
        """A request to / will access this controller"""
        return dict(name=name)
```

and rendered that name in it's template (and thus, the response body).

```
# myapp/myapp/templates/index.html
Hello, ${name}!
```

A functional test for this controller might look something like

```
class TestIndex(FunctionalTest):

    def test_index(self):
        resp = self.app.get('/')
        assert resp.status_int == 200
        assert 'Hello, Joe!' in resp.body
```

In addition to `webtest.TestResponse.body`, Pecan also provides `webtest.TestResponse.namespace`, which represents the template namespace returned from the controller, and `webtest.TestResponse.template_name`, which contains the name of the template used.

```
class TestIndex(FunctionalTest):

    def test_index(self):
        resp = self.app.get('/')
        assert resp.status_int == 200
        assert resp.namespace == {'name': 'Joe'}
        assert resp.template_name == 'index.html'
```

In this way, it's possible to test the return value and rendered template of individual controllers.

CHAPTER 2

# Cookbook and Common Patterns

## 2.1 Generating and Validating Forms

Pecan provides no opinionated support for working with form generation and validation libraries, but it's easy to import your library of choice with minimal effort.

This article details best practices for integrating the popular forms library, WTForms, into your Pecan project.

### 2.1.1 Defining a Form Definition

Let's start by building a basic form with a required `first_name` field and an optional `last_name` field.

```
from wtforms import Form, TextField, validators

class MyForm(Form):
    first_name = TextField(u'First Name', validators=[validators.required()])
    last_name = TextField(u'Last Name', validators=[validators.optional()])

class SomeController(object):
    pass
```

### 2.1.2 Rendering a Form in a Template

Next, let's add a controller, and pass a form instance to the template.

```
from pecan import expose
from wtforms import Form, TextField, validators

class MyForm(Form):
    first_name = TextField(u'First Name', validators=[validators.required()])
    last_name = TextField(u'Last Name', validators=[validators.optional()])

class SomeController(object):

    @expose(template='index.html')
    def index(self):
        return dict(form=MyForm())
```

Here's the Mako template file:

```
<form method="post" action="/">
    <div>
        ${form.first_name.label}:
        ${form.first_name}
    </div>
    <div>
        ${form.last_name.label}:
        ${form.last_name}
    </div>
    <input type="submit" value="submit">
</form>
```

### 2.1.3 Validating POST Values

Using the same `MyForm` definition, let's redirect the user if the form is validated, otherwise, render the form again.

```
from pecan import expose, request
from wtforms import Form, TextField, validators

class MyForm(Form):
    first_name = TextField(u'First Name', validators=[validators.required()])
    last_name = TextField(u'Last Name', validators=[validators.optional()])

class SomeController(object):

    @expose(template='index.html')
    def index(self):
        my_form = MyForm(request.POST)
        if request.method == 'POST' and my_form.validate():
            # save_values()
            redirect('/success')
        else:
            return dict(form=my_form)
```

## 2.2 Working with Sessions and User Authentication

Pecan provides no opinionated support for managing user sessions, but it's easy to hook into your session framework of choice with minimal effort.

This article details best practices for integrating the popular session framework, Beaker, into your Pecan project.

### 2.2.1 Setting up Session Management

There are several approaches that can be taken to set up session management. One approach is WSGI middleware. Another is Pecan *Pecan Hooks*.

Here's an example of wrapping your WSGI application with Beaker's `SessionMiddleware` in your project's `app.py`.

```
from pecan import conf, make_app
from beaker.middleware import SessionMiddleware
from test_project import model

app = make_app(
    ...
)
app = SessionMiddleware(app, conf.beaker)
```

And a corresponding dictionary in your configuration file.

```
beaker = {
    'session.key'           : 'sessionkey',
    'session.type'          : 'cookie',
    'session.validate_key'  : '05d2175d1090e31f42fa36e63b8d2aad',
    '__force_dict__'        : True
}
```

## 2.3 Working with Databases, Transactions, and ORM's

Pecan provides no opinionated support for working with databases, but it's easy to hook into your ORM of choice. This article details best practices for integrating the popular Python ORM, SQLAlchemy, into your Pecan project.

### 2.3.1 `init_model` and Preparing Your Model

Pecan's default quickstart project includes an empty stub directory for implementing your model as you see fit.

```
.
-- test_project
    -- app.py
    -- __init__.py
    -- controllers
    -- model
    |   -- __init__.py
    -- templates
```

By default, this module contains a special method, `init_model()`.

```
from pecan import conf

def init_model():
    """
    This is a stub method which is called at application startup time.

    If you need to bind to a parsed database configuration, set up tables
    or ORM classes, or perform any database initialization, this is the
    recommended place to do it.

    For more information working with databases, and some common recipes,
    see https://pecan.readthedocs.io/en/latest/databases.html
    """
    pass
```

The purpose of this method is to determine bindings from your configuration file and create necessary engines, pools, etc. according to your ORM or database toolkit of choice.

Additionally, your project's `model` module can be used to define functions for common binding operations, such as starting transactions, committing or rolling back work, and clearing a session. This is also the location in your project where object and relation definitions should be defined. Here's what a sample Pecan configuration file with database bindings might look like.

```
# Server Specific Configurations
server = {
    ...
}

# Pecan Application Configurations
app = {
    ...
}

# Bindings and options to pass to SQLAlchemy's ``create_engine``
sqlalchemy = {
    'url'           : 'mysql://root:@localhost/dbname?charset=utf8&use_unicode=0',
    'echo'          : False,
    'echo_pool'     : False,
    'pool_recycle'  : 3600,
    'encoding'      : 'utf-8'
}
```

And a basic model implementation that can be used to configure and bind using SQLAlchemy.

```
from pecan                  import conf
from sqlalchemy             import create_engine, MetaData
from sqlalchemy.orm         import scoped_session, sessionmaker

Session = scoped_session(sessionmaker())
metadata = MetaData()

def _engine_from_config(configuration):
    configuration = dict(configuration)
    url = configuration.pop('url')
    return create_engine(url, **configuration)

def init_model():
    conf.sqlalchemy.engine = _engine_from_config(conf.sqlalchemy)

def start():
    Session.bind = conf.sqlalchemy.engine
    metadata.bind = Session.bind

def commit():
    Session.commit()

def rollback():
    Session.rollback()

def clear():
    Session.remove()
```

### 2.3.2 Binding Within the Application

There are several approaches to wrapping your application's requests with calls to appropriate model function calls. One approach is WSGI middleware. We also recommend Pecan *Pecan Hooks*. Pecan comes with `TransactionHook`, a hook which can be used to wrap requests in database transactions for you. To use it, simply include it in your project's `app.py` file and pass it a set of functions related to database binding.

```
from pecan import conf, make_app
from pecan.hooks import TransactionHook
from test_project import model

app = make_app(
    conf.app.root,
    static_root     = conf.app.static_root,
    template_path   = conf.app.template_path,
    debug           = conf.app.debug,
    hooks           = [
        TransactionHook(
            model.start,
            model.start_read_only,
            model.commit,
            model.rollback,
            model.clear
        )
    ]
)
```

In the above example, on HTTP `POST`, `PUT`, and `DELETE` requests, *`TransactionHook`* takes care of the transaction automatically by following these rules:

1. Before controller routing has been determined, `model.start()` is called. This function should bind to the appropriate SQLAlchemy engine and start a transaction.
2. Controller code is run and returns.
3. If your controller or template rendering fails and raises an exception, `model.rollback()` is called and the original exception is re-raised. This allows you to rollback your database transaction to avoid committing work when exceptions occur in your application code.
4. If the controller returns successfully, `model.commit()` and `model.clear()` are called.

On idempotent operations (like HTTP `GET` and `HEAD` requests), *`TransactionHook`* handles transactions following different rules.

1. `model.start_read_only()` is called. This function should bind to your SQLAlchemy engine.
2. Controller code is run and returns.
3. If the controller returns successfully, `model.clear()` is called.

Also note that there is a useful *`after_commit()`* decorator provided in *pecan.decorators – Pecan Decorators*.

### 2.3.3 Splitting Reads and Writes

Employing the strategy above with *`TransactionHook`* makes it very simple to split database reads and writes based upon HTTP methods (i.e., GET/HEAD requests are read-only and would potentially be routed to a read-only database slave, while POST/PUT/DELETE requests require writing, and would always bind to a master database with read/write privileges). It's also possible to extend *`TransactionHook`* or write your own hook implementation for more refined control over where and when database bindings are called.

Assuming a master/standby setup, where the master accepts write requests and the standby can only get read requests, a Pecan configuration for sqlalchemy could be:

```
# Server Specific Configurations
server = {
    ...
}
```

```
# Pecan Application Configurations
app = {
    ...
}

# Master database
sqlalchemy_w = {
    'url': 'postgresql+psycopg2://root:@master_host/dbname',
    'pool_recycle': 3600,
    'encoding': 'utf-8'
}

# Read Only database
sqlalchemy_ro = {
    'url': 'postgresql+psycopg2://root:@standby_host/dbname',
    'pool_recycle': 3600,
    'encoding': 'utf-8'
}
```

Given the unique configuration settings for each database, the bindings would need to change from what Pecan's default quickstart provides (see *init_model and Preparing Your Model* section) to accommodate for both write and read only requests:

```
from pecan import conf
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import scoped_session, sessionmaker

Session = scoped_session(sessionmaker())
metadata = MetaData()


def init_model():
    conf.sqlalchemy_w.engine = _engine_from_config(conf.sqlalchemy_w)
    conf.sqlalchemy_ro.engine = _engine_from_config(conf.sqlalchemy_ro)

def _engine_from_config(configuration):
    configuration = dict(configuration)
    url = configuration.pop('url')
    return create_engine(url, **configuration)


def start():
    Session.bind = conf.sqlalchemy_w.engine
    metadata.bind = conf.sqlalchemy_w.engine


def start_read_only():
    Session.bind = conf.sqlalchemy_ro.engine
    metadata.bind = conf.sqlalchemy_ro.engine


def commit():
    Session.commit()


def rollback():
    Session.rollback()
```

```
def clear():
    Session.close()


def flush():
    Session.flush()
```

## 2.4 Custom Error Documents

In this article we will configure a Pecan application to display a custom error page whenever the server returns a `404 Page Not Found` status.

This article assumes that you have already created a test application as described in *Creating Your First Pecan Application*.

**Note:** While this example focuses on the `HTTP 404` message, the same technique may be applied to define custom actions for any of the `HTTP` status response codes in the 400 and 500 range. You are well advised to use this power judiciously.

### 2.4.1 Overview

Pecan makes it simple to customize error documents in two simple steps:

- *Configure Routing* of the HTTP status messages you want to handle in your application's `config.py`
- *Write Custom Controllers* to handle the status messages you have configured

### 2.4.2 Configure Routing

Let's configure our application `test_project` to route `HTTP 404 Page Not Found` messages to a custom controller.

First, let's update `test_project/config.py` to specify a new error-handler.

```
# Pecan Application Configurations
app = {
    'root'              : 'test_project.controllers.root.RootController',
    'modules'           : ['test_project'],
    'static_root'       : '%(confdir)s/public',
    'template_path'     : '%(confdir)s/test_project/templates',
    'reload'            : True,
    'debug'             : True,

    # modify the 'errors' key to direct HTTP status codes to a custom
    # controller
    'errors'            : {
        #404             : '/error/404',
        404             : '/notfound',
        '__force_dict__' : True
    }
}
```

Instead of the default error page, Pecan will now route 404 messages to the controller method `notfound`.

### 2.4.3 Write Custom Controllers

The easiest way to implement the error handler is to add it to `test_project.root.RootController` class (typically in `test_project/controllers/root.py`).

```
from pecan import expose
from webob.exc import status_map


class RootController(object):

    @expose(generic=True, template='index.html')
    def index(self):
        return dict()

    @index.when(method='POST')
    def index_post(self, q):
        redirect('https://pecan.readthedocs.io/en/latest/search.html?q=%s' % q)


    ## custom handling of '404 Page Not Found' messages
    @expose('error.html')
    def notfound(self):
        return dict(status=404, message="test_project does not have this page")


    @expose('error.html')
    def error(self, status):
        try:
            status = int(status)
        except ValueError:
            status = 0
        message = getattr(status_map.get(status), 'explanation', '')
        return dict(status=status, message=message)
```

And that's it!

Notice that the only bit of code we added to our `RootController` was:

```
## custom handling of '404 Page Not Found' messages
@expose('error.html')
def notfound(self):
    return dict(status=404, message="test_project does not have this page")
```

We simply *expose()* the `notfound` controller with the `error.html` template (which was conveniently generated for us and placed under `test_project/templates/` when we created `test_project`). As with any Pecan controller, we return a dictionary of variables for interpolation by the template renderer.

Now we can modify the error template, or write a brand new one to make the 404 error status page of `test_project` as pretty or fancy as we want.

## 2.5 Example Application: Simple Forms Processing

This guide will walk you through building a simple Pecan web application that will do some simple forms processing.

### 2.5.1 Project Setup

First, you'll need to install Pecan:

```
$ pip install pecan
```

Use Pecan's basic template support to start a new project:

```
$ pecan create mywebsite
$ cd mywebsite
```

Install the new project in development mode:

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

With the project ready, go into the `templates` folder and edit the `index.html` file. Modify it so that it resembles this:

```
<%inherit file="layout.html" />

<%def name="title()">
    Welcome to Pecan!
</%def>
    <header>
        <h1><img src="/images/logo.png" /></h1>
    </header>
    <div id="content">
        <form method="POST" action="/">
            <fieldset>
                <p>Enter a message: <input name="message" /></p>
                <p>Enter your first name: <input name="first_name" /></p>
                <input type="submit" value="Submit" />
            </fieldset>
        </form>
        % if not form_post_data is UNDEFINED:
            <p>${form_post_data['first_name']}, your message is: ${form_post_data['message']}</p>
        % endif
    </div>
```

**What did we just do?**

1. Modified the contents of the `form` tag to have two `input` tags. The first is named `message` and the second is named `first_name`
2. Added a check if `form_post_data` has not been defined so we don't show the message or wording
3. Added code to display the message from the user's `POST` action

Go into the `controllers` folder now and edit the `root.py` file. There will be two functions inside of the `RootController` class which will display the `index.html` file when your web browser hits the `'/'` endpoint. If the user puts some data into the textbox and hits the submit button then they will see the personalized message displayed back at them.

Modify the `root.py` to look like this:

```
from pecan import expose


class RootController(object):

    @expose(generic=True, template='index.html')
    def index(self):
        return dict()

    @index.when(method='POST', template='index.html')
    def index_post(self, **kwargs):
        return dict(form_post_data=kwargs)
```

**What did we just do?**

1. Modified the `index` function to render the initial `index.html` webpage
2. Modified the `index_post` function to return the posted data via keyword arguments

Run the application:

```
$ pecan serve config.py
```

Open a web browser: http://127.0.0.1:8080/

### 2.5.2 Adding Validation

Enter a message into the textbox along with a name in the second textbox and press the submit button. You should see a personalized message displayed below the form once the page posts back.

One problem you might have noticed is if you don't enter a message or a first name then you simply see no value entered for that part of the message. Let's add a little validation to make sure a message and a first name was actually entered. For this, we will use WTForms but you can substitute anything else for your projects.

Add support for the WTForms library:

```
$ pip install wtforms
```

**Note:** Keep in mind that Pecan is not opinionated when it comes to a particular library when working with form generation, validation, etc. Choose which libraries you prefer and integrate those with Pecan. This is one way of doing this, there are many more ways so feel free to handle this however you want in your own projects.

Go back to the `root.py` files and modify it like this:

```
from pecan import expose, request
from wtforms import Form, TextField, validators


class PersonalizedMessageForm(Form):
    message = TextField(u'Enter a message',
                        validators=[validators.required()])
    first_name = TextField(u'Enter your first name',
                           validators=[validators.required()])


class RootController(object):

    @expose(generic=True, template='index.html')
    def index(self):
        return dict(form=PersonalizedMessageForm())

    @index.when(method='POST', template='index.html')
    def index_post(self):
        form = PersonalizedMessageForm(request.POST)
        if form.validate():
            return dict(message=form.message.data,
                        first_name=form.first_name.data)
        else:
            return dict(form=form)
```

**What did we just do?**

1. Added the `PersonalizedMessageForm` with two textfields and a required field validator for each
2. Modified the `index` function to create a new instance of the `PersonalizedMessageForm` class and return it
3. In the `index_post` function modify it to gather the posted data and validate it. If its valid, then set the returned data to be displayed on the webpage. If not valid, send the form which will contain the data plus the error message(s)

Modify the `index.html` like this:

```
<%inherit file="layout.html" />

## provide definitions for blocks we want to redefine
<%def name="title()">
    Welcome to Pecan!
</%def>
    <header>
        <h1><img src="/images/logo.png" /></h1>
    </header>
    <div id="content">
        % if not form:
            <p>${first_name}, your message is: ${message}</p>
        % else:
            <form method="POST" action="/">
                <div>
                    ${form.message.label}:
                    ${form.message}
                    % if form.message.errors:
                        <strong>${form.message.errors[0]}</strong>
                    % endif
                </div>
               <div>
                    ${form.first_name.label}:
                    ${form.first_name}
                    % if form.first_name.errors:
                        <strong>${form.first_name.errors[0]}</strong>
                    % endif
                </div>
                <input type="submit" value="Submit">
            </form>
        % endif
    </div>
```

**Note:** Keep in mind when using the WTForms library you can customize the error messages and more. Also, you have multiple validation rules so make sure to catch all the errors which will mean you need a loop rather than the simple example above which grabs the first error item in the list. See the documentation for more information.

Run the application:

```
$ pecan serve config.py
```

Open a web browser: http://127.0.0.1:8080/

Try the form with valid data and with no data entered.

## 2.6 Example Application: Simple AJAX

This guide will walk you through building a simple Pecan web application that uses AJAX to fetch JSON data from a server.

### 2.6.1 Project Setup

First, you'll need to install Pecan:

```
$ pip install pecan
```

Use Pecan's basic template support to start a new project:

```
$ pecan create myajax
$ cd myajax
```

Install the new project in development mode:

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

### 2.6.2 Adding JavaScript AJAX Support

For this project we will need to add jQuery support. To add jQuery go into the `templates` folder and edit the `layout.html` file.

Adding jQuery support is easy, we actually only need one line of code:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
```

The JavaScript to make the AJAX call is a little more in depth but shouldn't be unfamiliar if you've ever worked with jQuery before.

The `layout.html` file will look like this:

```
<html>
    <head>
        <title>${self.title()}</title>
        ${self.style()}
        ${self.javascript()}
    </head>
    <body>
        ${self.body()}
    </body>
</html>

<%def name="title()">
    Default Title
</%def>
```

```
<%def name="style()">
    <link rel="stylesheet" type="text/css" media="screen" href="/css/style.css" />
</%def>

<%def name="javascript()">
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script language="text/javascript" src="/javascript/shared.js"></script>

    <script>
        function onSuccess(data, status, jqXHR) {
            // Use a template or something here instead
            // Just for demo purposes
            $("#result").html("<div>" +
                    "<p></p><strong>Project Name: " + data.name + "</strong></p>" +
                    "<p>Project License: " + data.licensing + "</p>" +
                    "<p><a href='" + data.repository + "'>Project Repository: " + data.repository + '
                    "<p><a href='" + data.documentation + "'>Project Documentation: " + data.document
                    "</div>");
        }

        function onError(jqXHR, textStatus, errorThrown) {
            alert('HTTP Status Code: ' + jqXHR.status + ', ' + errorThrown);
        }

        $(document).ready(function () {
            $("#submit").click(function () {
                $.ajax({
                    url: "/projects/",
                    data: "id=" + $("#projects").val(),
                    contentType: 'application/json',
                    dataType: 'json',
                    success: onSuccess,
                    error: onError
                });

                return false;
            });
        });
    </script>
</%def>
```

**What did we just do?**

1. In the `head` section we added jQuery support via the Google CDN
2. Added JavaScript to make an AJAX call to the server via an HTTP `GET` passing in the `id` of the project to fetch more information on
3. Once the `onSuccess` event is triggered by the returning data we take that and display it on the web page below the controls

### 2.6.3 Adding Additional HTML

Let's edit the `index.html` file next. We will add HTML to support the AJAX interaction between the web page and Pecan. Modify `index.html` to look like this:

```
<%inherit file="layout.html" />
```

```
<%def name="title()">
Welcome to Pecan!
</%def>

<header>
    <h1><img src="/images/logo.png"/></h1>
</header>

<div id="content">
    <p>Select a project to get details:</p>
    <select id="projects">
        <option value="0">OpenStack</option>
        <option value="1">Pecan</option>
        <option value="2">Stevedore</option>
    </select>
    <button id="submit" type="submit">Submit</button>

    <div id="result"></div>

</div>
```

**What did we just do?**

1. Added a dropdown control and submit button for the user to interact with. Users can pick an open source project and get more details on it

### 2.6.4 Building the Model with JSON Support

The HTML and JavaScript work is now taken care of. At this point we can add a model to our project inside of the `model` folder. Create a file in there called `projects.py` and add the following to it:

```python
class Project(object):
    def __init__(self, name, licensing, repository, documentation):
        self.name = name
        self.licensing = licensing
        self.repository = repository
        self.documentation = documentation

    def __json__(self):
        return dict(
            name=self.name,
            licensing=self.licensing,
            repository=self.repository,
            documentation=self.documentation
        )
```

**What did we just do?**

1. Created a model called `Project` that can hold project specific data
2. Added a `__json__` method so an instance of the `Project class` can be easily represented as JSON. The controller we will soon build will make use of that JSON capability

**Note:** There are other ways to return JSON with Pecan, check out *JSON Serialization* for more information.

### 2.6.5 Working with the Controllers

We don't need to do anything major to the `root.py` file in the `controllers` folder except to add support for a new controller we will call `ProjectsController`. Modify the `root.py` like this:

```
from pecan import expose

from myajax.controllers.projects import ProjectsController


class RootController(object):

    projects = ProjectsController()

    @expose(generic=True, template='index.html')
    def index(self):
        return dict()
```

**What did we just do?**

1. Removed some of the initial boilerplate code since we won't be using it
2. Add support for the upcoming `ProjectsController`

The final piece is to add a file called `projects.py` to the `controllers` folder. This new file will host the `ProjectsController` which will listen for incoming AJAX `GET` calls (in our case) and return the appropriate JSON response.

Add the following code to the `projects.py` file:

```
from pecan import expose, response
from pecan.rest import RestController

from myajax.model.projects import Project


class ProjectsController(RestController):

    # Note: You would probably store this information in a database
    # This is just for simplicity and demonstration purposes
    def __init__(self):
        self.projects = [
            Project(name='OpenStack',
                    licensing='Apache 2',
                    repository='http://github.com/openstack',
                    documentation='http://docs.openstack.org'),
            Project(name='Pecan',
                    licensing='BSD',
                    repository='http://github.com/pecan/pecan',
                    documentation='https://pecan.readthedocs.io'),
            Project(name='stevedore',
                    licensing='Apache 2',
                    repository='http://github.com/dreamhost/pecan',
                    documentation='http://docs.openstack.org/developer/stevedore/')
        ]


    @expose('json', content_type='application/json')
    def get(self, id):
        response.status = 200
        return self.projects[int(id)]
```

**What did we just do?**

1. Created a local class variable called `projects` that holds three open source projects and their details. Typically this kind of information would probably reside in a database
2. Added code for the new controller that will listen on the `projects` endpoint and serve back JSON based on the `id` passed in from the web page

Run the application:

```
$ pecan serve config.py
```

Open a web browser: http://127.0.0.1:8080/

There is something else we could add. What if an `id` is passed that is not found? A proper `HTTP 404` should be sent back. For this we will modify the `ProjectsController`.

Change the `get` function to look like this:

```
@expose('json', content_type='application/json')
def get(self, id):
    try:
        response.status = 200
        return self.projects[int(id)]
    except (IndexError, ValueError) as ex:
        abort(404)
```

To test this out we need to pass an invalid `id` to the `ProjectsController`. This can be done by going into the `index.html` and adding an additional `option` tag with an `id` value that is outside of 0-2.

```
<p>Select a project to get details:</p>
<select id="projects">
    <option value="0">OpenStack</option>
    <option value="1">Pecan</option>
    <option value="2">Stevedore</option>
    <option value="3">WSME</option>
</select>
```

You can see that we added `WSME` to the list and the value is 3.

Run the application:

```
$ pecan serve config.py
```

Open a web browser: http://127.0.0.1:8080/

Select `WSME` from the list. You should see the error dialog box triggered.

CHAPTER 3

# API Documentation

Pecan's source code is well documented using Python docstrings and comments. In addition, we have generated API documentation from the docstrings here:

## 3.1 `pecan.core` – Pecan Core

The *`pecan.core`* module is the base module for creating and extending Pecan. The core logic for processing HTTP requests and responses lives here.

**class** `pecan.core.`**`Pecan`**`(`**args*, ***kw*`)`

Bases: `pecan.core.PecanBase`

Pecan application object. Generally created using `pecan.make_app`, rather than being created manually.

Creates a Pecan application instance, which is a WSGI application.

**Parameters**

- **`root`** – A string representing a root controller object (e.g., "myapp.controller.root.RootController")
- **`default_renderer`** – The default template rendering engine to use. Defaults to mako.
- **`template_path`** – A relative file system path (from the project root) where template files live. Defaults to 'templates'.
- **`hooks`** – A callable which returns a list of *`pecan.hooks.PecanHook`*
- **`custom_renderers`** – Custom renderer objects, as a dictionary keyed by engine name.
- **`extra_template_vars`** – Any variables to inject into the template namespace automatically.
- **`force_canonical`** – A boolean indicating if this project should require canonical URLs.
- **`guess_content_type_from_ext`** – A boolean indicating if this project should use the extension in the URL for guessing the content type to return.
- **`use_context_locals`** – When *True*, *pecan.request* and *pecan.response* will be available as thread-local references.
- **`request_cls`** – Can be used to specify a custom *pecan.request* object. Defaults to *pecan.Request*.
- **`response_cls`** – Can be used to specify a custom *pecan.response* object. Defaults to *pecan.Response*.

`pecan.core.`**`abort`**(*status_code*, *detail=''*, *headers=None*, *comment=None*, ***kw*)

Raise an HTTP status code, as specified. Useful for returning status codes like 401 Unauthorized or 403 Forbidden.

**Parameters**

- **`status_code`** – The HTTP status code as an integer.
- **`detail`** – The message to send along, as a string.
- **`headers`** – A dictionary of headers to send along with the response.
- **`comment`** – A comment to include in the response.

`pecan.core.`**`load_app`**(*config*, ***kwargs*)

Used to load a `Pecan` application and its environment based on passed configuration.

**Parameters** **`config`** – Can be a dictionary containing configuration, a string which represents a (relative) configuration filename

returns a pecan.Pecan object

`pecan.core.`**`override_template`**(*template*, *content_type=None*)

Call within a controller to override the template that is used in your response.

**Parameters**

- **`template`** – a valid path to a template file, just as you would specify in an `@expose`.
- **`content_type`** – a valid MIME type to use for the response.func_closure

`pecan.core.`**`redirect`**(*location=None*, *internal=False*, *code=None*, *headers={}*, *add_slash=False*, *request=None*)

Perform a redirect, either internal or external. An internal redirect performs the redirect server-side, while the external redirect utilizes an HTTP 302 status code.

**Parameters**

- **`location`** – The HTTP location to redirect to.
- **`internal`** – A boolean indicating whether the redirect should be internal.
- **`code`** – The HTTP status code to use for the redirect. Defaults to 302.
- **`headers`** – Any HTTP headers to send with the response, as a dictionary.
- **`request`** – The `pecan.Request` instance to use.

`pecan.core.`**`render`**(*template*, *namespace*, *app=None*)

Render the specified template using the Pecan rendering framework with the specified template namespace as a dictionary. Useful in a controller where you have no template specified in the `@expose`.

**Parameters**

- **`template`** – The path to your template, as you would specify in `@expose`.
- **`namespace`** – The namespace to use for rendering the template, as a dictionary.
- **`app`** – The instance of `pecan.Pecan` to use

## 3.2 `pecan.commands` – Pecan Commands

The `pecan.commands` module implements the `pecan` console script used to provide (for example) `pecan serve` and `pecan shell` command line utilities.

**class** `pecan.commands.base.`**`BaseCommand`**

Bases: *`pecan.commands.base.BaseCommandParent`*

A base interface for Pecan commands.

Can be extended to support `pecan` command extensions in individual Pecan projects, e.g.,

$ `pecan my-custom-command config.py`

```
# myapp/myapp/custom_command.py
class CustomCommand(pecan.commands.base.BaseCommand):
    '''
    (First) line of the docstring is used to summarize the command.
    '''

    arguments = ({
        'name': '--extra_arg',
        'help': 'an extra command line argument',
        'optional': True
    })

    def run(self, args):
        super(SomeCommand, self).run(args)
        if args.extra_arg:
            pass
```

**class** `pecan.commands.base.`**`BaseCommandParent`**

Bases: `object`

A base interface for Pecan commands.

Can be extended to support `pecan` command extensions in individual Pecan projects, e.g.,

$ `pecan my-custom-command config.py`

```
# myapp/myapp/custom_command.py
class CustomCommand(pecan.commands.base.BaseCommand):
    '''
    (First) line of the docstring is used to summarize the command.
    '''

    arguments = ({
        'name': '--extra_arg',
        'help': 'an extra command line argument',
        'optional': True
    })

    def run(self, args):
        super(SomeCommand, self).run(args)
        if args.extra_arg:
            pass
```

**`run`**(*args*)

To be implemented by subclasses.

**class** `pecan.commands.base.`**`CommandManager`**

Bases: `object`

Used to discover *pecan.command* entry points.

**class** `pecan.commands.base.`**`CommandRunner`**

Bases: `object`

Dispatches *pecan* command execution requests.

## 3.2.1 `pecan.commands.server` – Pecan Development Server

Serve command for Pecan.

**class** `pecan.commands.serve.`**`PecanWSGIRequestHandler`**(**args*, ***kwargs*)

Bases: `wsgiref.simple_server.WSGIRequestHandler`, `object`

A wsgiref request handler class that allows actual log output depending on the application configuration.

**`log_message`**(*format*, **args*)

overrides the `log_message` method from the wsgiref server so that normal logging works with whatever configuration the application has been set to.

Levels are inferred from the HTTP status code, 4XX codes are treated as warnings, 5XX as errors and everything else as INFO level.

**class** `pecan.commands.serve.`**`ServeCommand`**

Bases: *`pecan.commands.base.BaseCommand`*

Serves a Pecan web application.

This command serves a Pecan web application using the provided configuration file for the server and application.

**`serve`**(*app*, *conf*)

A very simple approach for a WSGI server.

`pecan.commands.serve.`**`gunicorn_run`**()

The `gunicorn_pecan` command for launching `pecan` applications

## 3.2.2 `pecan.commands.shell` – Pecan Interactive Shell

Shell command for Pecan.

**class** `pecan.commands.shell.`**`BPythonShell`**

Bases: `object`

Open an interactive bpython shell with the Pecan app loaded.

**classmethod** **`invoke`**(*ns*, *banner*)

**Parameters**

- **`ns`** – local namespace
- **`banner`** – interactive shell startup banner

Embed an interactive bpython shell.

**class** `pecan.commands.shell.`**`IPythonShell`**

Bases: `object`

Open an interactive ipython shell with the Pecan app loaded.

**classmethod** **`invoke`**(*ns*, *banner*)

**Parameters**

- **`ns`** – local namespace
- **`banner`** – interactive shell startup banner

Embed an interactive ipython shell. Try the InteractiveShellEmbed API first, fall back on IPShellEmbed for older IPython versions.

**class** `pecan.commands.shell.`**`NativePythonShell`**

Bases: `object`

Open an interactive python shell with the Pecan app loaded.

**classmethod** **`invoke`**(*ns*, *banner*)

**Parameters**

- **`ns`** – local namespace
- **`banner`** – interactive shell startup banner

Embed an interactive native python shell.

**class** `pecan.commands.shell.`**`ShellCommand`**

Bases: *`pecan.commands.base.BaseCommand`*

Open an interactive shell with the Pecan app loaded. Attempt to invoke the specified python shell flavor (ipython, bpython, etc.). Fall back on the native python shell if the requested flavor variance is not installed.

**`invoke_shell`**(*locs*, *banner*)

Invokes the appropriate flavor of the python shell. Falls back on the native python shell if the requested flavor (ipython, bpython,etc) is not installed.

**`load_model`**(*config*)

Load the model extension module

**`run`**(*args*)

Load the pecan app, prepare the locals, sets the banner, and invokes the python shell.

## 3.3 `pecan.configuration` – Pecan Configuration Engine

The *`pecan.configuration`* module provides an implementation of a Python-based configuration engine for Pecan applications.

**class** `pecan.configuration.`**`Config`**(*conf_dict={}*, *filename=''*)

Bases: `object`

Base class for Pecan configurations.

Create a Pecan configuration object from a dictionary or a filename.

**Parameters**

- **`conf_dict`** – A python dictionary to use for the configuration.
- **`filename`** – A filename to use for the configuration.

**`to_dict`**(*prefix=None*)

Converts recursively the Config object into a valid dictionary.

**Parameters** **`prefix`** – A string to optionally prefix all key elements in the returned dictonary.

**`update`**(*conf_dict*)

Updates this configuration with a dictionary.

**Parameters** **`conf_dict`** – A python dictionary to update this configuration with.

`pecan.configuration.`**`conf_from_dict`**(*conf_dict*)

Creates a configuration dictionary from a dictionary.

**Parameters** **`conf_dict`** – The configuration dictionary.

`pecan.configuration.`**`conf_from_file`**`(`*filepath*`)`

Creates a configuration dictionary from a file.

**Parameters** **`filepath`** – The path to the file.

`pecan.configuration.`**`get_conf_path_from_env`**`()`

If the `PECAN_CONFIG` environment variable exists and it points to a valid path it will return that, otherwise it will raise a `RuntimeError`.

`pecan.configuration.`**`initconf`**`()`

Initializes the default configuration and exposes it at `pecan.configuration.conf`, which is also exposed at `pecan.conf`.

`pecan.configuration.`**`set_config`**`(`*config*, *overwrite=False*`)`

Updates the global configuration.

**Parameters** **`config`** – Can be a dictionary containing configuration, or a string which represents a (relative) configuration filename.

## 3.4 `pecan.decorators` – Pecan Decorators

The *`pecan.decorators`* module includes useful decorators for creating Pecan applications.

`pecan.decorators.`**`expose`**`(`*template=None*, *generic=False*, *route=None*, ***kw*`)`

Decorator used to flag controller methods as being "exposed" for access via HTTP, and to configure that access.

**Parameters**

- **`template`** – The path to a template, relative to the base template directory. Can also be passed a string representing a special or custom renderer, such as `'json'` for *The JSON Renderer*.
- **`content_type`** – The content-type to use for this template.
- **`generic`** – A boolean which flags this as a "generic" controller, which uses generic functions based upon `functools.singledispatch` generic functions. Allows you to split a single controller into multiple paths based upon HTTP method.
- **`route`** – The name of the path segment to match (excluding separator characters, like `/`). Defaults to the name of the function itself, but this can be used to resolve paths which are not valid Python function names, e.g., if you wanted to route a function to 'some-special-path'.

`pecan.decorators.`**`transactional`**`(`*ignore_redirects=True*`)`

If utilizing the *`pecan.hooks`* `TransactionHook`, allows you to flag a controller method or class as being wrapped in a transaction, regardless of HTTP method.

**Parameters** **`ignore_redirects`** – Indicates if the hook should ignore redirects for this controller or not.

`pecan.decorators.`**`accept_noncanonical`**`(`*func*`)`

Flags a controller method as accepting non-canoncial URLs.

`pecan.decorators.`**`after_commit`**`(`*action*`)`

If utilizing the *`pecan.hooks`* `TransactionHook`, allows you to flag a controller method to perform a callable action after the commit is successfully issued.

**Parameters** **`action`** – The callable to call after the commit is successfully issued.

`pecan.decorators.`**`after_rollback`**(*action*)

If utilizing the *`pecan.hooks`* `TransactionHook`, allows you to flag a controller method to perform a callable action after the rollback is successfully issued.

**Parameters** **`action`** – The callable to call after the rollback is successfully issued.

## 3.5 `pecan.deploy` – Pecan Deploy

The *`pecan.deploy`* module includes fixtures to help deploy Pecan applications.

`pecan.deploy.`**`deploy`**(*config*)

Given a config (dictionary of relative filename), returns a configured WSGI app.

## 3.6 `pecan.hooks` – Pecan Hooks

The *`pecan.hooks`* module includes support for creating Pecan `hooks` which are a simple way to hook into the request processing of requests coming into your application to perform cross-cutting tasks.

**class** `pecan.hooks.`**`PecanHook`**

Bases: `object`

A base class for Pecan hooks. Inherit from this class to create your own hooks. Set a priority on a hook by setting the `priority` attribute for the hook, which defaults to 100.

**`after`**(*state*)

Override this method to create a hook that gets called after the request has been handled by the controller.

**Parameters** **`state`** – The Pecan `state` object for the current request.

**`before`**(*state*)

Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

**Parameters** **`state`** – The Pecan `state` object for the current request.

**`on_error`**(*state*, *e*)

Override this method to create a hook that gets called upon an exception being raised in your controller.

**Parameters**

- **`state`** – The Pecan `state` object for the current request.
- **`e`** – The `Exception` object that was raised.

**`on_route`**(*state*)

Override this method to create a hook that gets called upon the start of routing.

**Parameters** **`state`** – The Pecan `state` object for the current request.

**class** `pecan.hooks.`**`TransactionHook`**(*start*, *start_ro*, *commit*, *rollback*, *clear*)

Bases: *`pecan.hooks.PecanHook`*

**Parameters**

- **`start`** – A callable that will bind to a writable database and start a transaction.
- **`start_ro`** – A callable that will bind to a readable database.
- **`commit`** – A callable that will commit the active transaction.
- **`rollback`** – A callable that will roll back the active transaction.

- **clear** – A callable that will clear your current context.

A basic framework hook for supporting wrapping requests in transactions. By default, it will wrap all but `GET` and `HEAD` requests in a transaction. Override the `is_transactional` method to define your own rules for what requests should be transactional.

**is_transactional**(*state*)
Decide if a request should be wrapped in a transaction, based upon the state of the request. By default, wraps all but `GET` and `HEAD` requests in a transaction, along with respecting the `transactional` decorator from :mod:pecan.decorators.

**Parameters** **state** – The Pecan state object for the current request.

**class** pecan.hooks.**HookController**
Bases: `object`

A base class for controllers that would like to specify hooks on their controller methods. Simply create a list of hook objects called __hooks__ as a class attribute of your controller.

**class** pecan.hooks.**RequestViewerHook**(*config=None*, *writer=<open file '<stdout>', mode 'w'>*, *terminal=True*, *headers=True*)
Bases: *pecan.hooks.PecanHook*

**Parameters**

- **config** – A (optional) dictionary that can hold `items` and/or `blacklist` keys.
- **writer** – The stream writer to use. Can redirect output to other streams as long as the passed in stream has a `write` callable method.
- **terminal** – Outputs to the chosen stream writer (usually the terminal)
- **headers** – Sets values to the X-HTTP headers

Returns some information about what is going on in a single request. It accepts specific items to report on but uses a default list of items when none are passed in. Based on the requested `url`, items can also be blacklisted. Configuration is flexible, can be passed in (or not) and can contain some or all the keys supported.

**items**

This key holds the items that this hook will display. When this key is passed only the items in the list will be used. Valid items are *any* item that the `request` object holds, by default it uses the following:

- path
- status
- method
- controller
- params
- hooks

**Note:** This key should always use a `list` of items to use.

**blacklist**

This key holds items that will be blacklisted based on `url`. If there is a need to omit urls that start with */javascript*, then this key would look like:

```
'blacklist': ['/javascript']
```

As many blacklisting items as needed can be contained in the list. The hook will verify that the url is not starting with items in this list to display results, otherwise it will get omitted.

**Note:** This key should always use a `list` of items to use.

For more detailed documentation about this hook, please see *RequestViewerHook*

**`format_hooks`**(*hooks*)
Tries to format the hook objects to be more readable Specific to Pecan (not available in the request object)

**`get_controller`**(*state*)
Retrieves the actual controller name from the application Specific to Pecan (not available in the request object)

## 3.7 `pecan.middleware.debug` – Pecan Debugging Middleware

## 3.8 `pecan.jsonify` – Pecan `JSON` Support

The *`pecan.jsonify`* module includes support for `JSON` rule creation using generic functions.

**class** `pecan.jsonify.`**`GenericJSON`**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)
Bases: `json.encoder.JSONEncoder`

Generic JSON encoder. Makes several attempts to correctly JSONify requested response objects.

**`default`**(*obj*)
Converts an object and returns a `JSON`-friendly structure.

**Parameters** **`obj`** – object or structure to be converted into a `JSON`-ifiable structure

Considers the following special cases in order:

- **object has a callable __json__() attribute defined** returns the result of the call to __json__()
- **date and datetime objects** returns the object cast to str
- **Decimal objects** returns the object cast to float
- **SQLAlchemy objects** returns a copy of the object.__dict__ with internal SQLAlchemy parameters removed
- **SQLAlchemy ResultProxy objects** Casts the iterable ResultProxy into a list of tuples containing the entire resultset data, returns the list in a dictionary along with the resultset "row" count.

  **Note:** {'count': 5, 'rows': [('Ed Jones',), ('Pete Jones',), ('Wendy Williams',), ('Mary Contrary',), ('Fred Smith',)]}

- **SQLAlchemy RowProxy objects** Casts the RowProxy cursor object into a dictionary, probably losing its ordered dictionary behavior in the process but making it JSON-friendly.
- **webob_dicts objects** returns webob_dicts.mixed() dictionary, which is guaranteed to be JSON-friendly.

## 3.9 `pecan.rest` – Pecan `REST` Controller

The *`pecan.rest`* module includes support for writing fully `RESTful` controllers in your Pecan application.

**class** `pecan.rest.`**`RestController`**

Bases: `object`

A base class for `REST` based controllers. Inherit from this class to implement a REST controller.

`RestController` implements a set of routing functions which override the default pecan routing with behavior consistent with RESTful routing. This functionality covers navigation to the requested resource controllers, and the appropriate handling of both the common (`GET`, `POST`, `PUT`, `DELETE`) as well as custom-defined REST action methods.

For more on developing **RESTful** web applications with Pecan, see *Writing RESTful Web Services with Generic Controllers*.

## 3.10 `pecan.routing` – Pecan Routing

The *`pecan.routing`* module is the basis for all object-dispatch routing in Pecan.

`pecan.routing.`**`lookup_controller`**(*obj*, *remainder*, *request=None*)

Traverses the requested url path and returns the appropriate controller object, including default routes.

Handles common errors gracefully.

`pecan.routing.`**`find_object`**(*obj*, *remainder*, *notfound_handlers*, *request*)

'Walks' the url path in search of an action for which a controller is implemented and returns that controller object along with what's left of the remainder.

`pecan.routing.`**`route`**(**args*)

This function is used to define an explicit route for a path segment.

You generally only want to use this in situations where your desired path segment is not a valid Python variable/function name.

For example, if you wanted to be able to route to:

/path/with-dashes/

...the following is invalid Python syntax:

```
class Controller(object):

    with-dashes = SubController()
```

...so you would instead define the route explicitly:

```
class Controller(object):
    pass

pecan.route(Controller, 'with-dashes', SubController())
```

## 3.11 `pecan.secure` – Pecan Secure Controllers

The *`pecan.secure`* module includes a basic framework for building security into your applications.

`pecan.secure.`**`unlocked`**(*func_or_obj*)
This method unlocks method or class attribute on a SecureController. Can be used to decorate or wrap an attribute

`pecan.secure.`**`secure`**(*func_or_obj*, *check_permissions_for_obj=None*)
This method secures a method or class depending on invocation.

**To decorate a method use one argument:** @secure(<check_permissions_method>)

**To secure a class, invoke with two arguments:** secure(<obj instance>, <check_permissions_method>)

**class** `pecan.secure.`**`SecureControllerBase`**
Bases: `object`

**classmethod** **`check_permissions`**()
Returns *True* or *False* to grant access. Implemented in subclasses of *`SecureController`*.

**class** `pecan.secure.`**`SecureController`**
Bases: *`pecan.secure.SecureControllerBase`*

Used to apply security to a controller. Implementations of SecureController should extend the *check_permissions* method to return a True or False value (depending on whether or not the user has permissions to the controller).

## 3.12 `pecan.templating` – Pecan Templating

The *`pecan.templating`* module includes support for a variety of templating engines, plus the ability to create your own template engines.

**class** `pecan.templating.`**`ExtraNamespace`**(*extras={}*)
Bases: `object`

Extra variables for the template namespace to pass to the renderer as named parameters.

**Parameters** **`extras`** – dictionary of extra parameters. Defaults to an empty dict.

**`make_ns`**(*ns*)
Returns the *lazily* created template namespace.

**`update`**(*d*)
Updates the extra variable dictionary for the namespace.

**class** `pecan.templating.`**`JinjaRenderer`**(*path*, *extra_vars*)
Bases: `object`

Defines the builtin `Jinja` renderer.

**`render`**(*template_path*, *namespace*)
Implements `Jinja` rendering.

**class** `pecan.templating.`**`JsonRenderer`**(*path*, *extra_vars*)
Bases: `object`

Defines the builtin `JSON` renderer.

**`render`**(*template_path*, *namespace*)
Implements `JSON` rendering.

**class** `pecan.templating.`**`MakoRenderer`**(*path*, *extra_vars*)
Bases: `object`

Defines the builtin `Mako` renderer.

**`render`** (*template_path*, *namespace*)
Implements `Mako` rendering.

**class** `pecan.templating.`**`RendererFactory`** (*custom_renderers={}*, *extra_vars={}*)
Bases: `object`

Manufactures known Renderer objects.

**Parameters**

- **`custom_renderers`** – custom-defined renderers to manufacture
- **`extra_vars`** – extra vars for the template namespace

**`add_renderers`** (*custom_dict*)
Adds a custom renderer.

**Parameters** **`custom_dict`** – a dictionary of custom renderers to add

**`available`** (*name*)
Returns true if queried renderer class is available.

**Parameters** **`name`** – renderer name

**`get`** (*name*, *template_path*)
Returns the renderer object.

**Parameters**

- **`name`** – name of the requested renderer
- **`template_path`** – path to the template

`pecan.templating.`**`format_jinja_error`** (*exc_value*)
Implements `Jinja` renderer error formatting.

`pecan.templating.`**`format_line_context`** (*filename*, *lineno*, *context=10*)
Formats the the line context for error rendering.

**Parameters**

- **`filename`** – the location of the file, within which the error occurred
- **`lineno`** – the offending line number
- **`context`** – number of lines of code to display before and after the offending line.

`pecan.templating.`**`format_mako_error`** (*exc_value*)
Implements `Mako` renderer error formatting.

## 3.13 `pecan.testing` – Pecan Testing

The *`pecan.testing`* module includes fixtures to help test Pecan applications.

`pecan.testing.`**`load_test_app`** (*config=None*, ***kwargs*)
Used for functional tests where you need to test your literal application and its integration with the framework.

**Parameters** **`config`** – Can be a dictionary containing configuration, a string which represents a (relative) configuration filename or `None` which will fallback to get the `PECAN_CONFIG` env variable.

returns a pecan.Pecan WSGI application wrapped in a webtest.TestApp instance.

**::** app = load_test_app('path/to/some/config.py')

resp = app.get('/path/to/some/resource').status_int assert resp.status_int == 200

resp = app.post('/path/to/some/resource', params={'param': 'value'}) assert resp.status_int == 302

Alternatively you could call `load_test_app` with no parameters if the environment variable is set

```
app = load_test_app()

resp = app.get('/path/to/some/resource').status_int
assert resp.status_int == 200
```

`pecan.testing.`**`reset_global_config`**`()`
When tests alter application configurations they can get sticky and pollute other tests that might rely on a pristine configuration. This helper will reset the config by overwriting it with `pecan.configuration.DEFAULT`.

## 3.14 `pecan.util` – Pecan Utils

The *`pecan.util`* module includes utility functions for Pecan.

`pecan.util.`**`getargspec`**`(`*method*`)`
Drill through layers of decorators attempting to locate the actual argspec for a method.

CHAPTER 4

# Change History

## 4.1 1.2.1

- Reverts a stable API change/regression (in the 1.2 release) (https://github.com/pecan/pecan/issues/72). This change will re-released in a future major version upgrade.

## 4.2 1.2

- Added a better error message when an invalid template renderer is specified in *pecan.expose()* (https://github.com/pecan/pecan/issues/81).
- Pecan controllers that return *None* are now treated as an *HTTP 204 No Content* (https://github.com/pecan/pecan/issues/72).
- The *method* argument to *pecan.expose()* for generic controllers is no longer optional (https://github.com/pecan/pecan/pull/77).

## 4.3 1.1.2

- Fixed a bug where JSON-formatted HTTP response bodies were not making use of pecan's JSON type registration functionality (http://pecan.readthedocs.io/en/latest/jsonify.html) (https://github.com/pecan/pecan/issues/68).
- Updated code and documentation examples to support readthedoc's move from *readthedocs.org* to *readthedocs.io*.

## 4.4 1.1.1

- Pecan now officially supports Python 3.5.
- Pecan now uses *inspect.signature* instead of *inspect.getargspec* in Python 3.5 and higher (because *inspect.getargspec* is deprecated in these versions of Python 3).
- Fixed a bug that caused "after" hooks to run multiple times when *pecan.redirect(..., internal=True)* was used (https://github.com/pecan/pecan/issues/58).

## 4.5 1.1.0

- *pecan.middleware.debug.DebugMiddleware* now logs exceptions at the ERROR level (https://github.com/pecan/pecan/pull/56).
- Fix a Javascript bug in the default project scaffold (https://github.com/pecan/pecan/pull/55).

## 4.6 1.0.5

- Fix a bug in controller argspec detection when class-based decorators are used (https://github.com/pecan/pecan/issues/47).

## 4.7 1.0.4

- Removed an open file handle leak when pecan renders errors for Jinja2 and Genshi templates (https://github.com/pecan/pecan/issues/30).
- Resolved a bug which caused log output to be duplicated in projects created with *pecan create* (https://github.com/pecan/pecan/issues/39).

## 4.8 1.0.3

- Fixed a bug in *pecan.hooks.HookController* for newer versions of Python3.4 (https://github.com/pecan/pecan/issues/19).

## 4.9 1.0.2

- Fixed an edge case in *pecan.util.getargspec* that caused the incorrect argspec to be returned in certain situations when using Python 2.6.
- Added a *threading.lock* to the file system monitoring in *pecan serve –reload* to avoid extraneous server reloads.

## 4.10 1.0.1

- Fixed a bug wherein the file extension for URLs with a trailing slash (*file.html* vs *file.html/*) were not correctly guessed, thus resulting in incorrect Content-Type headers.
- Fixed a subtle bug in *pecan.config.Configuration* attribute/item assignment that caused some types of configuration changes to silently fail.

## 4.11 1.0.0

- Replaced pecan's debugger middleware with an (optional) dependency on the *backlash* package. Developers who want to debug application-level tracebacks interactively should *pip install backlash* in their development environment.

- Fixed a Content-Type related bug: when an explicit content_type is specified as an argument to *pecan.expose()*, it is now given precedence over the application-level default renderer.
- Fixed a bug that prevented the usage of certain RFC3986-specified characters in path segments.
- Fixed a bug in *pecan.abort* which suppressed the original traceback (and prevented monitoring tools like NewRelic from working as effectively).

## 4.12 0.9.0

- Support for Python 3.2 has been dropped.
- Added a new feature which allows users to specify custom path segments for controllers. This is especially useful for path segments that are not valid Python identifiers (such as path segments that include certain punctuation characters, like */some/~path~/*).
- Added a new configuration option, *app.debugger*, which allows developers to specify an alternative debugger to *pdb* (e.g., *ipdb*) when performing interactive debugging with pecan's *DebugMiddleware*.
- Changed new quickstart pecan projects to default the *pecan* log level to *DEBUG* for development.
- Fixed a bug that prevented *staticmethods* from being used as controllers.
- Fixed a decoding bug in the way pecan handles certain quoted URL path segments and query strings.
- Fixed several bugs in the way pecan handles Unicode path segments (for example, now you can define pecan routes that contain emoji characters).
- Fixed several bugs in RestController that caused it to return *HTTP 404 Not Found* rather than *HTTP 405 Method Not Allowed*. Additionally, RestController now returns valid *Allow* headers when *HTTP 405 Method Not Allowed* is returned.
- Fixed a bug which allowed special pecan methods (*_route*, *_lookup*, *_default*) to be marked as generic REST methods.
- Added more emphasis in pecan's documentation to the need for *debug=False* in production deployments.

## 4.13 0.8.3

- Changed pecan to more gracefully handle a few odd request encoding edge cases. Now pecan applications respond with an HTTP 400 (rather than an uncaught UnicodeDecodeError, resulting in an HTTP 500) when:
  - HTTP POST requests are composed of non-Unicode data
  - Request paths contain invalid percent-encoded characters, e.g., `/some/path/%aa/`
- Improved verbosity for import-related errors in pecan configuration files, especially those involving relative imports.

## 4.14 0.8.2

- Fixes a bug that breaks support for multi-value query string variables (e.g., *?check=a&check=b*).

## 4.15 0.8.1

- Improved detection of infinite recursion for PecanHook and pypy. This fixes a bug discovered in pecan + pypy that could result in infinite recursion when using the PecanHook metaclass.
- Fixed a bug that prevented @exposed controllers from using @staticmethod.
- Fixed a minor bug in the controller argument calculation.

## 4.16 0.8.0

- For HTTP POSTs, map JSON request bodies to controller keyword arguments.
- Improved argspec detection and leniency for wrapped controllers.
- When path arguments are incorrect for RestController, return HTTP 404, not 400.
- When detecting non-content for HTTP 204, properly catch UnicodeDecodeError.
- Fixed a routing bug for generic subcontrollers.
- Fixed a bug in generic function handling when context locals are disabled.
- Fixed a bug that mixes up argument order for generic functions.
- Removed *assert* for flow control; it can be optimized away with *python -O*.

## 4.17 0.7.0

- Fixed an edge case in RestController routing which should have returned an HTTP 400 but was instead raising an exception (and thus, HTTP 500).
- Fixed an incorrect root logger configuration for quickstarted pecan projects.
- Added *pecan.state.arguments*, a new feature for inspecting controller call arguments.
- Fixed an infinite recursion error in PecanHook application. Subclassing both *rest.RestController* and *hooks.HookController* resulted in an infinite recursion error in hook application (which prevented applications from starting).
- Pecan's tests are now included in its source distribution.

## 4.18 0.6.1

- Fixed a bug which causes pecan to mistakenly return HTTP 204 for non-empty response bodies.

## 4.19 0.6.0

- Added support for disabling the *pecan.request* and *pecan.response* threadlocals at the WSGI application level in favor of explicit reference passing. For more information, see *Context/Thread-Locals vs. Explicit Argument Passing*.
- Added better support for hook composition via subclassing and mixins. For more information, see *Attaching Hooks*.

- Added support for specifying custom request and response implementations at the WSGI application level for people who want to extend the functionality provided by the base classes in *webob*.
- Pecan controllers may now return an explicit *webob.Response* instance to short-circuit Pecan's template rendering and serialization.
- For generic methods that return HTTP 405, pecan now generates an *Allow* header to communicate acceptable methods to the client.
- Fixed a bug in adherence to RFC2616: if an exposed method returns no response body (or namespace), pecan will now enforce an HTTP 204 response (instead of HTTP 200).
- Fixed a bug in adherence to RFC2616: when pecan responds with HTTP 204 or HTTP 304, the *Content-Type* header is automatically stripped (because these types of HTTP responses do not contain body content).
- Fixed a bug: now when clients request JSON via an *Accept* header, *webob* HTTP exceptions are serialized as JSON, not their native HTML representation.
- Fixed a bug that broke applications which specified *default_renderer = json*.

## 4.20 0.5.0

- This release adds formal support for pypy.
- Added colored request logging to the *pecan serve* command.
- Added a scaffold for easily generating a basic REST API.
- Added the ability to pass arbitrary keyword arguments to *pecan.testing.load_test_app*.
- Fixed a recursion-related bug in the error document middleware.
- Fixed a bug in the *gunicorn_pecan* command that caused *threading.local* data to leak between eventlet/gevent green threads.
- Improved documentation through fixes and narrative tutorials for sample pecan applications.

## 4.21 0.4.5

- Fixed a trailing slash bug for *RestController's that have a '_lookup* method.
- Cleaned up the WSGI app reference from the threadlocal state on every request (to avoid potential memory leaks, especially when testing).
- Improved pecan documentation and corrected intersphinx references.
- pecan supports Python 3.4.

## 4.22 0.4.4

- Removed memoization of certain controller attributes, which can lead to a memory leak in dynamic controller lookups.

## 4.23 0.4.3

- Fixed several bugs for RestController.
- Fixed a bug in security handling for generic controllers.
- Resolved a bug in *_default* handlers used in *RestController*.
- Persist *pecan.request.context* across internal redirects.

## 4.24 0.4.2

- Remove a routing optimization that breaks the WSME pecan plugin.

## 4.25 0.4.1

- Moved the project to StackForge infrastructure, including Gerrit code review, Jenkins continuous integration, and GitHub mirroring.
- Added a pecan plugin for the popular uwsgi server.
- Replaced the `simplegeneric` dependency with the new `functools.singledispatch` function in preparation for Python 3.4 support.
- Optimized pecan's core dispatch routing for notably faster response times.

## 4.26 0.3.2

- Made some changes to simplify how `pecan.conf.app` is passed to new apps.
- Fixed a routing bug for certain `_lookup` controller configurations.
- Improved documentation for handling file uploads.
- Deprecated the `pecan.conf.requestviewer` configuration option.

## 4.27 0.3.1

- `on_error` hooks can now return a Pecan Response objects.
- Minor documentation and release tooling updates.

## 4.28 0.3.0

- Pecan now supports Python 2.6, 2.7, 3.2, and 3.3.

## 4.29 0.2.4

- Add support for `_lookup` methods as a fallback in RestController.
- A variety of improvements to project documentation.

## 4.30 0.2.3

- Add a variety of optimizations to `pecan.core` that improve request handling time by approximately 30% for simple object dispatch routing.
- Store exceptions raised by `abort` in the WSGI environ so they can be accessed later in the request handling (e.g., by other middleware or pecan hooks).
- Make TransactionHook more robust so that it isn't as susceptible to failure when exceptions occur in *other* pecan hooks within a request.
- Rearrange quickstart verbiage so users don't miss a necessary step.

## 4.31 0.2.2

- Unobfuscate syntax highlighting JavaScript for debian packaging.
- Extract the scaffold-building tests into tox.
- Add support for specifying a pecan configuration file via the `PECAN_CONFIG` environment variable.
- Fix a bug in `DELETE` methods in two (or more) nested `RestControllers`.
- Add documentation for returning specific HTTP status codes.

## 4.32 0.2.1

- Include a license, readme, and `requirements.txt` in distributions.
- Improve inspection with `dir()` for `pecan.request` and `pecan.response`
- Fix a bug which prevented pecan applications from being mounted at WSGI virtual paths.

## 4.33 0.2.0

- Update base project scaffolding tests to be more repeatable.
- Add an application-level configuration option to disable content-type guessing by URL
- Fix the wrong test dependency on Jinja, it's Jinja2.
- Fix a routing-related bug in `RestController`. Fixes #156
- Add an explicit `CONTRIBUTING.rst` document.
- Improve visibility of deployment-related docs.
- Add support for a `gunicorn_pecan` console script.

- Remove and annotate a few unused (and py26 alternative) imports.
- Bug fix: don't strip a dotted extension from the path unless it has a matching mimetype.
- Add a test to the scaffold project buildout that ensures pep8 passes.
- Fix misleading output for `$ pecan --version`.

## 4.34 0.2.0b

- Fix a bug in `SecureController`. Resolves #131.
- Extract debug middleware static file dependencies into physical files.
- Improve a test that can fail due to a race condition.
- Improve documentation about configation format and `app.py`.
- Add support for content type detection via HTTP Accept headers.
- Correct source installation instructions in `README`.
- Fix an incorrect code example in the Hooks documentation.
- docs: Fix minor typo in `*args` Routing example.

## 4.35 License

The Pecan framework and the documentation is BSD Licensed:

# Python Module Index