
Pylons Reference Documentation

Release 1.0.2

Ben Bangert, Graham Higgins, James Gardner, Philip Jenvey

January 12, 2018

Contents

1	Getting Started	1
1.1	Requirements	1
1.2	Installing	1
1.3	Creating a Pylons Project	3
1.4	Running the application	4
1.5	Hello World	4
2	Concepts of Pylons	7
2.1	The ‘Why’ of a Pylons Project	7
2.2	WSGI Applications	8
2.3	WSGI Middleware	8
2.4	Controller Dispatch	10
2.5	Paster	10
2.6	Loading the Application	11
3	Controllers	13
3.1	Standard Controllers	14
3.2	Using the WSGI Controller to provide a WSGI service	16
3.3	Using the REST Controller with a RESTful API	17
3.4	Using the XML-RPC Controller for XML-RPC requests	20
4	Views	23
4.1	Templates	24
4.2	Passing Variables to Templates	24
4.3	Default Template Variables	25
4.4	Configuring Template Engines	26
4.5	Custom <code>render()</code> functions	27
4.6	Templating with Mako	28
5	Models	31
5.1	About the model	31
5.2	Model Basics	32
5.3	Organizing	34
5.4	Creating a Model	34
5.5	Adding a Relation	35
5.6	Creating the Database	36
5.7	A brief guide to using model objects in the Controller	36

5.8	Logging	41
5.9	About SQLAlchemy	41
6	Advanced Models	43
6.1	Advanced SQLAlchemy	43
6.2	Non-SQLAlchemy libraries	47
6.3	Object Databases	48
6.4	Popular No-SQL Databases	48
7	Configuration	49
7.1	Runtime Configuration	49
7.2	Environment	51
7.3	URL Configuration	51
7.4	Middleware	53
7.5	Application Setup	55
8	Logging	57
8.1	Logging messages	57
8.2	Basic Logging configuration	58
8.3	Filtering log messages	59
8.4	Advanced Configuration	60
8.5	Request logging with Paste’s TransLogger	60
8.6	Logging to wsgi.errors	61
9	Helpers	67
9.1	Pagination	67
9.2	Secure Form Tag Helpers	72
10	Forms	73
10.1	The basics	73
10.2	Getting Started	73
10.3	Using the Helpers	75
10.4	File Uploads	75
10.5	Validating user input with FormEncode	76
10.6	Other Form Tools	79
11	Internationalization and Localization	81
11.1	Introduction	81
11.2	Getting Started	82
11.3	Using Babel	83
11.4	Back To Work	85
11.5	Testing the Application	86
11.6	Fallback Languages	87
11.7	Translations Within Templates	88
11.8	Lazy Translations	88
11.9	Producing a Python Egg	89
11.10	Plural Forms	90
11.11	Summary	90
11.12	Further Reading	90
11.13	<code>babel.core</code> – Babel core classes	91
11.14	<code>babel.localedata</code> — Babel locale data	103
11.15	<code>babel.dates</code> – Babel date classes	103
11.16	<code>babel.numbers</code> – Babel number classes	105
12	Sessions	109

12.1	Sessions	109
12.2	The Session Object	109
12.3	Configuring the Session	110
12.4	Storing SQLAlchemy mapped objects in Beaker sessions	111
12.5	Custom and caching middleware	111
12.6	Using <i>Session</i> in Internationalization	111
12.7	Using <i>Session</i> in Secure Forms	112
12.8	Hacking the session for no cookies	112
12.9	Using middleware (Beaker) with a composite app	113
13	Caching	115
13.1	Types of Caching	115
13.2	Namespaces and Keys	116
13.3	Configuring	116
13.4	Browser-Side	117
13.5	Controller Actions	118
13.6	Templates	119
13.7	Arbitrary Functions	119
13.8	Fragments	120
14	Unit and functional testing	121
14.1	Unit Testing with <code>webtest</code>	121
14.2	Example: Testing a Controller	122
14.3	Testing Pylons Objects	124
14.4	Testing Your Own Objects	124
14.5	Unit Testing	125
14.6	Functional Testing	125
15	Errors, Troubleshooting, and Debugging	127
15.1	Error Middleware	127
15.2	Interactive Debugging	129
15.3	E-mailing Errors	130
15.4	Programmatically Handling Errors	130
16	Upgrading	133
16.1	1.0 -> 1.0.1	133
16.2	0.9.7 -> 1.0	133
17	Packaging and Deployment Overview	137
17.1	Egg Files	137
17.2	Installing as a Non-root User	138
17.3	Understanding the Setup Process	138
17.4	Deploying the Application	140
17.5	Advanced Usage	140
18	Running Pylons Apps with Other Web Servers	141
18.1	Using Fast-CGI	141
18.2	Apache Configuration	142
18.3	PrefixMiddleware	142
18.4	Using Java Web Servers with Jython	143
19	Documenting Your Application	145
19.1	Introduction	145
19.2	Tutorial	145
19.3	Learning ReStructuredText	146

19.4	Using Docstrings	146
19.5	Using doctest	147
19.6	Summary	147
20	Distributing Your Application	149
20.1	Running Your Application	150
21	Python 2.3 Installation Instructions	151
21.1	Advice of end of support for Python 2.3	151
21.2	Preparation	151
21.3	System-wide Install	151
22	Windows Notes	153
22.1	For Win2K or WinXP	153
22.2	For Windows 95, 98 and ME	154
22.3	Finally	154
23	Pylons on Jython	155
23.1	Installation	155
23.2	Deploying to Java Web servers	155
24	Security policy for bugs	157
24.1	Receiving Security Updates	157
24.2	Reporting Security Issues	157
24.3	Minimising Risk	158
25	WSGI support	159
25.1	Paste and WSGI	159
25.2	Using a WSGI Application as a Pylons 0.9 Controller	160
25.3	Running a WSGI Application From Within a Controller	160
25.4	Configuring Middleware Within a Pylons Application	161
25.5	The Cascade	162
25.6	Useful Resources	162
26	Advanced Pylons	163
26.1	WSGI, CLI scripts	163
26.2	Adding commands to Paster	165
26.3	Creating Paste templates	168
26.4	Using Entry Points to Write Plugins	172
27	Pylons Execution Analysis	175
27.1	The sample application	175
27.2	Pylons' dependencies	176
27.3	The analysis	177
28	Pylons Modules	187
28.1	<code>pylons.commands</code> – Command line functions	187
28.2	<code>pylons.configuration</code> – Configuration object and defaults setup	187
28.3	<code>pylons.controllers</code> – Controllers	187
28.4	<code>pylons.controllers.core</code> – WSGIController Class	189
28.5	<code>pylons.controllers.util</code> – Controller Utility functions	189
28.6	<code>pylons.controllers.xmlrpc</code> – XMLRPCController Class	189
28.7	<code>pylons.decorators</code> – Decorators	189
28.8	<code>pylons.decorators.cache</code> – Cache Decorators	189
28.9	<code>pylons.decorators.rest</code> – REST-ful Decorators	189

28.10	<code>pylons.decorators.secure</code> – Secure Decorators	189
28.11	<code>pylons.error</code> – Error handling support	189
28.12	<code>pylons.i18n.translation</code> – Translation/Localization functions	189
28.13	<code>pylons.log</code> – Logging for WSGI errors	189
28.14	<code>pylons.middleware</code> – WSGI Middleware	189
28.15	<code>pylons.templating</code> – Render functions and helpers	190
28.16	<code>pylons.test</code> – Test related functionality	190
28.17	<code>pylons.util</code> – Paste Template and Pylons utility functions	190
28.18	<code>pylons.wsgiapp</code> – PylonsWSGI App Creator	190
29	Third-party components	191
29.1	<code>FormEncode</code>	191
29.2	<code>weberror</code> – Weberror	192
29.3	<code>webtest</code> – WebTest	192
29.4	<code>webob</code> – Request/Response objects	192
30	Glossary	193
	Python Module Index	197

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

1.1 Requirements

- **Python 2 series above and including 2.4 (Python 3 or later not supported at this time)**

1.2 Installing

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

By the Way

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

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

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

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

Tip

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

```
$ curl https://raw.githubusercontent.com/Pylons/pylons/master/scripts/go-pylons.py |  
↪python - mydevenv
```

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

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

How it Works

The `go-pylons.py` script is little more than a basic [virtualenv](#) bootstrap script, that then does `easy_install Pylons==1.0`. You could do the equivalent steps by manually fetching the `virtualenv.py` script and then installing Pylons like so:

```
curl -O http://bitbucket.org/ianb/virtualenv/raw/8dd7663d9811/virtualenv.py  
python virtualenv.py mydevenv  
mydevenv/bin/easy_install Pylons==1.0
```

This will leave a functional `virtualenv` and Pylons installation.

Activate the virtual environment (scripts may also be run by specifying the full path to the `mydevenv/bin` dir):

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

Or on Windows to activate:

```
> mydevenv\Scripts\activate.bat
```

Note: If you get an error such as:

```
ImportError: No module named _md5
```

during the install. It is likely that your Python installation is missing standard libraries needed to run Pylons. Debian and other systems using debian packages most frequently encounter this, make sure to install the `python-dev` packages and `python-hashlib` packages.

1.2.1 Working Directly From the Source Code

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

Check out the latest code:

```
$ hg clone http://bitbucket.org/bbangert/pylons/
```

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

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

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

1.3 Creating a Pylons Project

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

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

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

Running this will prompt for two choices:

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

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

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

- **helloworld**
 - MANIFEST.in
 - README.txt
 - development.ini - *Runtime Configuration*
 - docs
 - ez_setup.py
 - helloworld (See the nested *helloworld directory*)
 - helloworld.egg-info
 - setup.cfg
 - setup.py - *Application Setup*
 - test.ini

The nested `helloworld` directory looks like this:

- **helloworld**
 - `__init__.py`
 - **config**
 - * `environment.py` - *Environment*
 - * `middleware.py` - *Middleware*
 - * `routing.py` - *URL Configuration*
 - `controllers` - *Controllers*

- lib
 - * `app_globals.py` - *app_globals*
 - * `base.py`
 - * `helpers.py` - *Helpers*
- `model` - *Models*
- `public`
- `templates` - *Templates*
- `tests` - *Unit and functional testing*
- `websetup.py` - *Runtime Configuration*

1.4 Running the application

Run the web application:

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

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

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

The `paster serve` command can be run anywhere, as long as the `development.ini` path is properly specified. Generally during development it's run in the root directory of the project.

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

1.5 Hello World

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

```
$ paster controller hello
```

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

```
import logging

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

from helloworld.lib.base import BaseController, render

log = logging.getLogger(__name__)

class HelloController(BaseController):
```

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

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

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



Tip

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

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

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

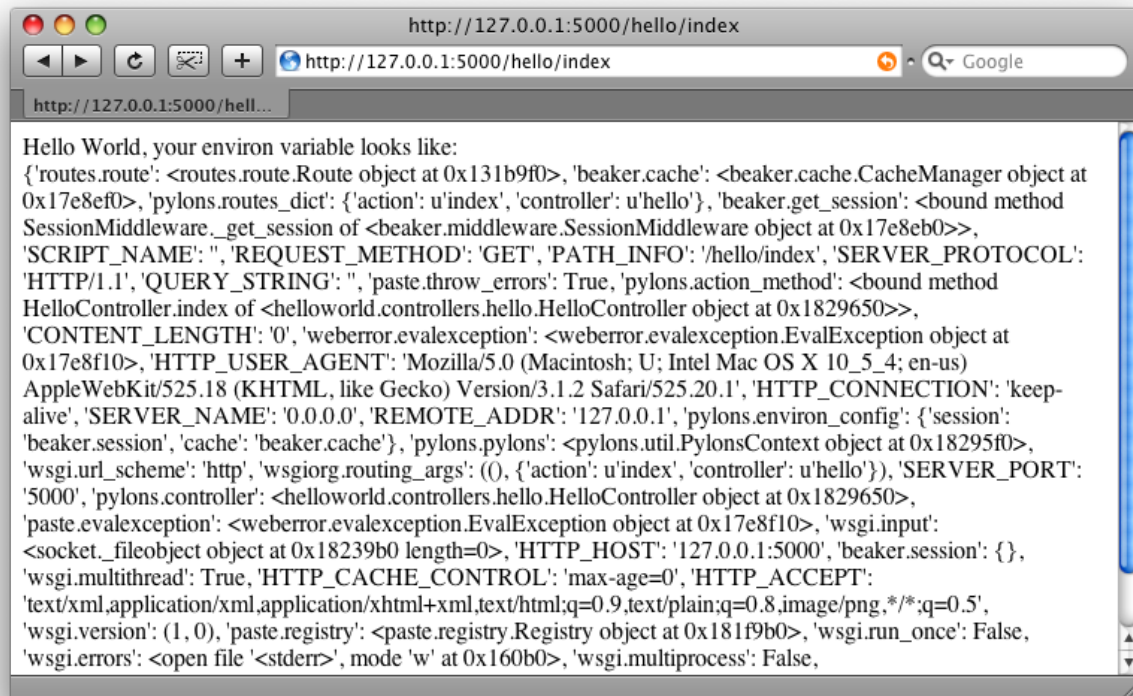
```
Hello World, the environ variable looks like: <br />  
  
${request.environ}
```

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

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

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

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



Concepts of Pylons

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

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

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

2.1 The ‘Why’ of a Pylons Project

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

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

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

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

2.2 WSGI Applications

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

A ‘Hello World’ WSGI Application:

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

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

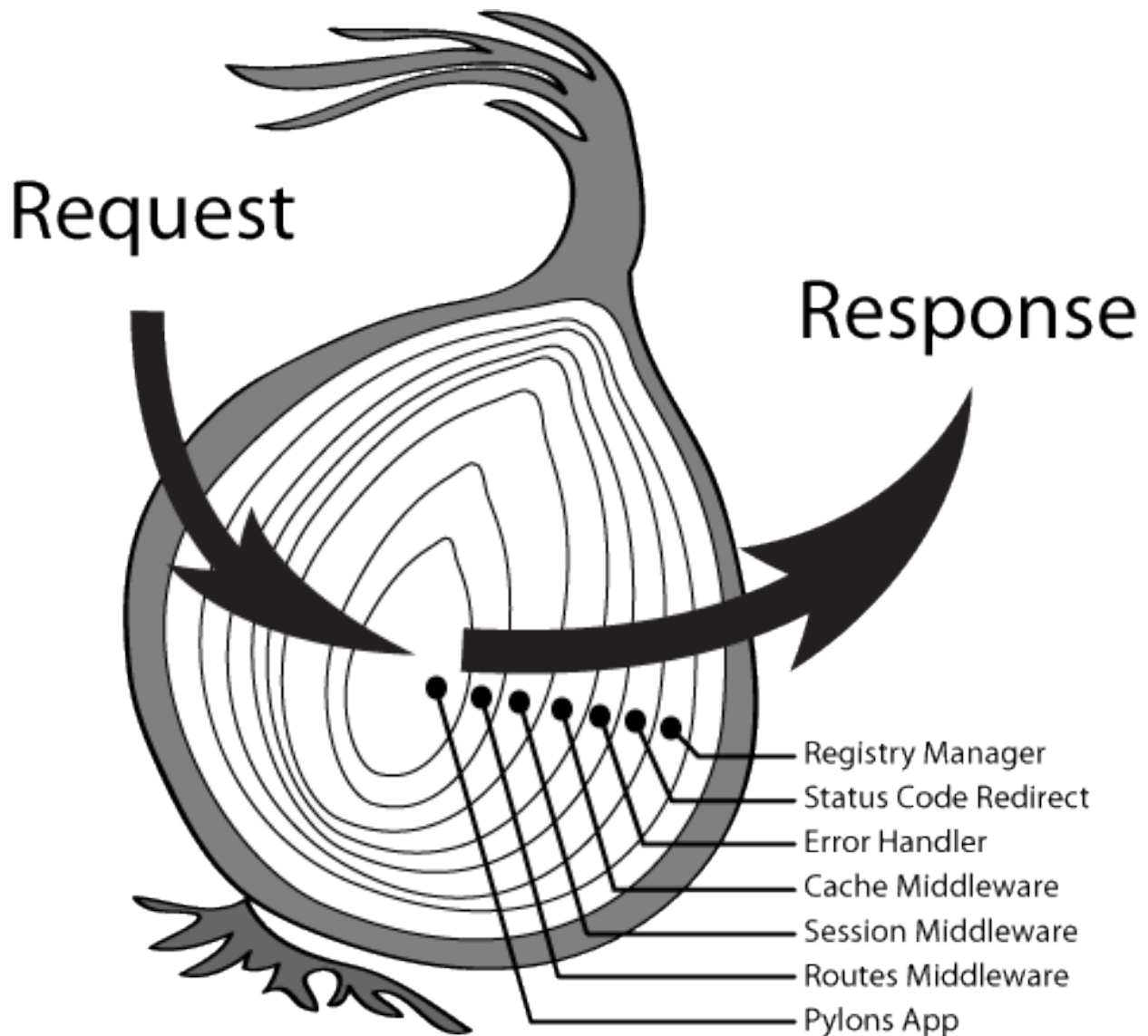
The WSGI specification lays out a [set of keys that will be set in the environ dict](#).

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

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

2.3 WSGI Middleware

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



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

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

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

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

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

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

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

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

2.4 Controller Dispatch

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

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

New in version 1.0: Controller name can also be a dotted path to the module / callable that should be imported and called. For example, to use a controller named 'Foo' that is in the 'bar.controllers' package, the controller name would be `bar.controllers:Foo`.

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

2.5 Paster

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

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

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

Commands:
  create            Create the file layout for a Python distribution
```

```

grep          Search project for symbol
help          Display help
make-config   Install a package and create a fresh config file/directory
points        Show information about entry points
post          Run a request for the described application
request       Run a request for the described application
serve         Serve the described application
setup-app     Setup an application, given a config file

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

```

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

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

2.6 Loading the Application

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

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

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

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

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

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

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

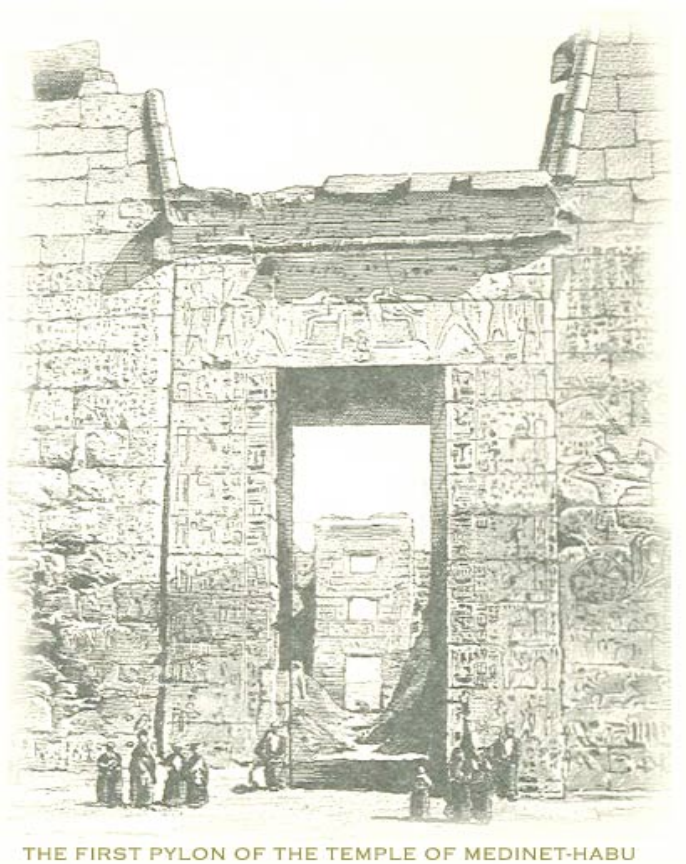
[paste.app_install]
main = pylons.util:PylonsInstaller
"""
```

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

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

CHAPTER 3

Controllers



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

template.

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

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

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

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

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

3.1 Standard Controllers

Standard Controllers intended for subclassing by web developers

3.1.1 Keeping methods private

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

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

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

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

3.1.2 Special methods

Special controller methods you may define:

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

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

3.1.3 Adding Controllers dynamically

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

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

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

```
from routes import request_config

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

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

3.1.4 Customizing the Controller Name

By default, Pylons looks for a controller named ‘Something’Controller. This naming scheme can be overridden by supplying an optional module-level variable called `__controller__` to indicate the desired controller class:

```
import logging

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

from helloworld.lib.base import BaseController, render

log = logging.getLogger(__name__)

__controller__ = 'Hello'

class Hello(BaseController):

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

3.1.5 Attaching WSGI apps

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

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

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

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

Create a new controller file in your Pylons project directory:

```
$ paster controller wsgiapp
```

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

Edit your controller so it looks like this:

```
import logging

from YOURPROJ.lib.base import *

log = logging.getLogger(__name__)

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

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

```
# CUSTOM ROUTES HERE

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

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

3.2 Using the WSGI Controller to provide a WSGI service

3.2.1 The Pylons WSGI Controller

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

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

3.2.2 WSGIController methods

Special `WSGIController` methods you may define:

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

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

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

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

3.3 Using the REST Controller with a RESTful API

3.3.1 Using the paster restcontroller template

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

Create a REST Controller and accompanying functional test

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

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

Example usage:

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

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

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

3.3.2 An Atom-Style REST Controller for Users

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

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

from restmarks.lib.base import *

log = logging.getLogger(__name__)

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

    def index(self, format='html'):
        """GET /users: All items in the collection.<br>
        @param format the format passed from the URI.
```

```
"""
#url('users')
users = model.User.select()
if format == 'json':
    data = []
    for user in users:
        d = user._state['original'].data
        del d['password']
        d['link'] = url('user', id=user.name)
        data.append(d)
    response.headers['content-type'] = 'text/javascript'
    return dumps(data)
else:
    c.users = users
    return render('/users/index_user.mako')

def create(self):
    """POST /users: Create a new item."""
    # url('users')
    user = model.User.get_by(name=request.params['name'])
    if user:
        # The client tried to create a user that already exists
        abort(409, '409 Conflict',
              headers=[('location', url('user', id=user.name))])
    else:
        try:
            # Validate the data that was sent to us
            params = model.forms.UserForm.to_python(request.params)
        except Invalid, e:
            # Something didn't validate correctly
            abort(400, '400 Bad Request -- %s' % e)
        user = model.User(**params)
        model.objectstore.flush()
        response.headers['location'] = url('user', id=user.name)
        response.status_code = 201
        c.user_name = user.name
        return render('/users/created_user.mako')

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

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

```

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

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

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

def edit(self, id, format='html'):
    """GET /users/id;edit: Form to edit an existing item.
    @param id the id (name) of the user to be updated.
    @param format the format of the URI requested.
    """
    # url('edit_user', id=ID)

```

```
user = model.User.get_by(name=id)
if not user:
    abort(404, '404 Not Found')
# Get the form values from the table
c.values = model.forms.UserForm.from_python(user.__dict__)
return render('/users/edit_user.mako')
```

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

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

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

3.4.1 A brief introduction to XML-RPC

XML-RPC is a specification that describes a Remote Procedure Call (RPC) interface by which an application can use the Internet to execute a specified procedure call on a remote XML-RPC server. The name of the procedure to be called and any required parameter values are “marshalled” into XML. The XML forms the body of a POST request which is dispatched via HTTP to the XML-RPC server. At the server, the procedure is executed, the returned value(s) is/are marshalled into XML and dispatched back to the application. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

3.4.2 XML-RPC Controller that speaks WSGI

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

This controller handles XML-RPC responses and complies with the [XML-RPC Specification](#) as well as the [XML-RPC Introspection](#) specification.

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

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

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

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

For example:

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

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

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

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

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

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

Note, requiring a signature is optional.

Also note that a convenient fault handler function is provided.

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

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

3.4.3 A simple XML-RPC service

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

```
import xmlrpclib

from pylons import request
from pylons.controllers import XMLRPCController

states = ['Delaware', 'Pennsylvania', 'New Jersey', 'Georgia',
          'Connecticut', 'Massachusetts', 'Maryland', 'South Carolina',
```

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

class RpctestController(XMLRPCController):

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

3.4.4 Testing the service

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

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

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

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

CHAPTER 4

Views



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

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

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

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

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

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

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

4.1 Templates

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

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

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

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

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

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

4.1.1 Directly-supported template engines

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

4.2 Passing Variables to Templates

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

```
import logging

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

from helloworld.lib.base import BaseController, render

log = logging.getLogger(__name__)

class HelloController(BaseController):
```



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

Using the variable in the template:

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

4.2.1 Strict vs Attribute-Safe `tmpl_context` objects

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

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

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

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

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

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

4.3 Default Template Variables

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

- `c` – Template context object (Alias for `tmpl_context`)
- `tmpl_context` – Template context object
- `config` – Pylons `PylonsConfig` object (acts as a dict)
- `g` – Project application globals object (Alias for `app_globals`)
- `app_globals` – Project application globals object
- `h` – Project helpers module reference
- `request` – Pylons `Request` object for this request
- `response` – Pylons `Response` object for this request
- `session` – Pylons session object (unless Sessions are removed)
- `translator` – Gettext translator object configured for current locale
- `ungettext()` – Unicode capable version of gettext's `ngettext` function (handles plural translations)
- `_()` – Unicode capable gettext translate function
- `N_()` – gettext no-op function to mark a string for translation, but doesn't actually translate
- `url` – An instance of the `routes.util.URLGenerator` configured for this request.

4.4 Configuring Template Engines

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

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

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

4.4.1 Using Multiple Template Engines

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

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

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

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

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

```
from pylons.templating import cached_template, pylons_globals

def render_mako_admin(template_name, extra_vars=None, cache_key=None,
                      cache_type=None, cache_expire=None):
    # Create a render callable for the cache function
    def render_template():
        # Pull in extra vars if needed
        globs = extra_vars or {}

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

```

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

    return template.render(**globs)

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

```

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

4.5 Custom render() functions

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

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

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

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

```

from pylons.templating import pylons_globals

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

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

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

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

```

Using the `pylons_globals()` function also makes it easy to get to the `app_globals` object which is where the template engine was attached in `config/environment.py`.

Changed in version 0.9.7: Prior to 0.9.7, all templating was handled through a layer called 'Buffer'. This layer frequently made customization of the template engine difficult as any customization required additional plugin modules being installed. Pylons 0.9.7 now deprecates use of the Buffet plug-in layer.

See also:

`pylons.templating` - Pylons templating API

4.6 Templating with Mako

4.6.1 Introduction

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

Static vs. dynamic

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

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

4.6.2 Making a template hierarchy

Create a base template

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

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

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

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

Create child templates

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

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

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

<h1>My Controller</h1>

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

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

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

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

Check that it works

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

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

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

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

<h1>My Controller</h1>

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

  </body>
</html>
```

See also:

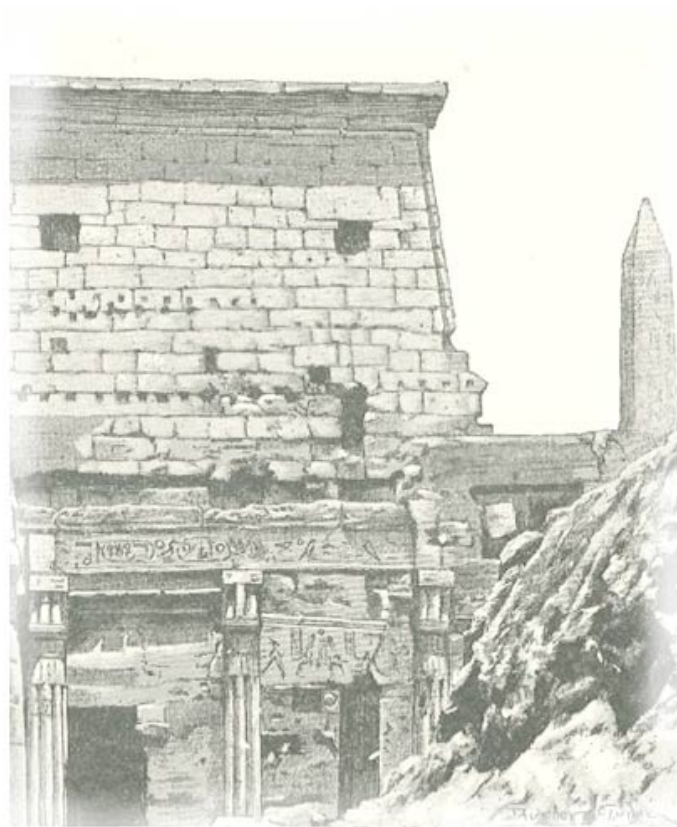
The [Mako documentation](#) Reasonably straightforward to follow

See the [Internationalization and Localization](#) Provides more help on making your application more worldly.

CHAPTER 5

Models

5.1 About the model



ONE OF THE PYLONS OF RAMESES II AT LUXOR

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

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

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

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

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

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

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

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

5.2 Model Basics

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

The recommended and most commonly-adopted approach used in Pylons applications is to use SQLAlchemy with the declarative configuration style and develop with a relational database (Postgres, MySQL, etc).

This is the documented and recommended approach for creating a Pylons project with a SQL database.

5.2.1 Install SQLAlchemy

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

```
easy_install SQLAlchemy
```

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

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


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

Tip: Checking Your Version

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

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
0.5.8
```

5.2.2 Create a Pylons Project with SQLAlchemy

When creating a Pylons project, one of the questions asked as part of the project creation dialogue is whether the project should be configured with SQLAlchemy. Before continuing, ensure that the project was created with this option, if it's missing the `model/meta.py` file, then the project should be re-created with this option.

Tip: The project doesn't need to be deleted to add this option, just re-run the *paster* command in the project's parent directory and answer "yes" to the SQLAlchemy prompt. The files will then be added and existing files will present a prompt on whether to replace them or leave the current file.

5.2.3 Configure SQLAlchemy

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

For SQLite

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

Where *mydatabasefilename.db* is the path to your SQLite database file. “%(here)s” represents the directory containing the *development.ini* file. If you're using an absolute path, use four slashes after the colon: “*sqlite:///var/lib/myapp/database.sqlite*”. Don't use a relative path (three slashes) because the current directory could be anything. The example has three slashes because the value of “%(here)s” always starts with a slash (or the platform equivalent; e.g., “C:\foo” on Windows).

For MySQL

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

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

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

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

For PostgreSQL

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

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

5.3 Organizing

When you answer "yes" to the SQLAlchemy question when creating a Pylons project, it configures a simple default model. The model consists of two files: `model/__init__.py` and `model/meta.py`.

5.3.1 `model/__init__.py`

The file `model/__init__.py` contains the table definitions, the ORM classes and an `init_model()` function. This `init_model()` function must be called at application startup. In the Pylons default project template this call is made in the `load_environment()` function (in the file `config/environment.py`).

5.3.2 `model/meta.py`

`model/meta.py` is merely a container for a few housekeeping objects required by SQLAlchemy such as `Session`, `metadata` and `engine` to avoid import issues. In the context of the default Pylons application, only the `Session` object is instantiated.

The objects are optional in the context of other applications that do not make use of them and so if you answer "no" to the SQLAlchemy question when creating a Pylons project, the creation of `model/meta.py` is simply skipped.

It is recommended that, for each model, a new module inside the `model/` directory should be created. This keeps the models tidy when they get larger as more domain specific code is added to each one.

5.4 Creating a Model

SQLAlchemy 0.5 has an optional *Declarative* syntax which offers the convenience of defining the table and the ORM class in one step. This is the recommended usage of SQLAlchemy.

Create a `model/person.py` module:

```
"""Person model"""
from sqlalchemy import Column
from sqlalchemy.types import Integer, String

from myapp.model.meta import Base

class Person(Base):
    __tablename__ = "person"

    id = Column(Integer, primary_key=True)
```

```

name = Column(String(100))
email = Column(String(100))

def __init__(self, name='', email=''):
    self.name = name
    self.email = email

def __repr__(self):
    return "<Person('%s') " % self.name

```

Note: Base is imported from `model/meta.py` to prevent recursive import problems when added to `model/__init__.py` in the next step.

Then for convenience when using the models, import it in `model/__init__.py`:

```

"""The application's model objects"""
from myapp.model.meta import Session, Base

from myapp.model.person import Person

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

```

5.5 Adding a Relation

Here's an example of a Person and an Address class with a one-to-many relationship on *person.addresses*.

First, add a `model/address.py` module:

```

"""Address model"""
from sqlalchemy import Column, ForeignKey
from sqlalchemy.types import Integer, String
from sqlalchemy.orm import relation, backref

from myapp.model.meta import Base

class Address(Base):
    __tablename__ = "address"

    id = Column(Integer, primary_key=True)
    address = Column(String(100))
    city = Column(String(100))
    state = Column(String(2))
    person_id = Column(Integer, ForeignKey('person.id'))

    person = relation('Person', backref=backref('addresses', order_by=id))

    def __repr__(self):
        return "<Person('%s') " % self.name

```

When models are created using the declarative Base, each one is added by name to a mapping. This allows the relation option above to locate the model it should be related to based on the text string 'Person'.

Then add the import to the `model/__init__.py` file:

```
"""The application's model objects"""
from myapp.model.meta import Session, Base

from myapp.model.address import Address
from myapp.model.person import Person

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

See also:

[Building a Relation and SQLAlchemy manual](#)

5.6 Creating the Database

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

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

Then run the following on the command line:

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

5.7 A brief guide to using model objects in the Controller

In which we: query a model, update a model entity, create a model entity and delete several model entities, all inside a Pylons controller.

To illustrate some typical ways of handling model objects in the Controller, we will draw from the example `PagesController` code of the QuickWiki Tutorial.

5.7.1 The Session

The SQLAlchemy-provided `Session` object is a crucially important facet when working with models and model object entities.

The SQLAlchemy documentation describes the `Session` thus: “In the most general sense, the `Session` establishes all conversations with the database and represents a “holding zone” for all the mapped instances which you’ve loaded or created during its lifespan.”

All of the model access that takes place in a Pylons controller is done in the context of a `Session` providing a database connection reference that is created at the start of the processing of each request and destroyed at the end of the processing of the request.

These creation and destruction operations are performed automatically by the `BaseController` instantiated in `MYAPP/lib/base.py` which is in turn subclassed for each standard Pylons controller, ensuring that subclassed controllers can access the database only in a request-specific context which, in turn, protects against data accidentally leaking across requests.

See also:

SQLAlchemy documentation for the [Session object](#)

The net effect of this is that a fully-instantiated `Session` object is available for import and immediate use in the controller for, e.g. querying the model.

5.7.2 Querying the model

The `Session` object provides a `query()` function that, when applied to a class of mapped model object, returns a SQLAlchemy `Query` object that can be passed around and repeatedly consulted.

See also:

SQLAlchemy documentation for the [Query object](#)

Standard usage is illustrated in this code for the `__before__()` function of the `QuickWiki PagesController` in which `self.page_q` is bound to the `Query` object returned by `Session.query(Page)` - where `Page` is the class of mapped model object that will be the subject of the queries.

```
from MYAPP.lib.base import Session
from MYAPP.model import Page

class PagesController(BaseController):

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

    # [ ... ]
```

The `Query` object that is bound to `self.page_q` is now specialised to perform queries of the `Page` declarative base entity / mapped model entity.

See also:

SQLAlchemy documentation for the [Querying the database](#)

Here, in the context of a controller's `index()` action, it is used in a very straightforward manner - `self.page_q.all()` - to fuel a list comprehension that returns a list containing the `title` of every `Page` object in the database:

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

and `self.page_q` is used in similarly direct manner for the `show()` action that retrieves a `Page` with a given value of `title` and then calls the `Page`'s `get_wiki_content()` class method.

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

Note: the `title` argument to the function is bound when the request is dispatched by the Routes map, typically of the form:

```
map.connect('show_page', '/page/show/{title}', controller='page', action='show')
```

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

5.7.3 Creating, updating and deleting model entities

When performing operations that change the state of the database, the recommended approach is for Pylons users to take full advantage of the abstraction provided by the SQLAlchemy ORM and simply treat the retrieved or created model entities as Python objects, make changes to them in a conventional Pythonic way, add them to or delete them from the `Session` “holding zone” and call `Session.commit()` to commit the changes to the database.

The three examples shown below are condensed illustrations of how these operations are typically performed in controller actions.

Creating a model entity

SQLAlchemy’s Declarative Base syntax allows model entity classes to act as constructors, accepting keyworded args and values. In this example, a new `Page` is created with the given title, the created model entity object is then added to the `Session` and then the change is committed.

```
def create(self, title):
    page = Page(title=title)
    Session.add(page)
    Session.commit()
    redirect_to('show_page', title=title)
```

Updating a model entity

Perhaps the most straightforward use - a model entity object is retrieved from the database, a field value is updated and the change committed.

(Note, this example is considerably abbreviated as a controller action - preliminary content checking has been omitted, as has exception handling for the database query.)

```
def save(self, title):
    page = self.page_q.filter_by(title=title).first()
    page.content=escape(request.POST.getone('content'))
    Session.commit()
    redirect_to('show_page', title=title)
```

Deleting a model entity

This example of shows the freedom that the Pylons user has to make repeated changes to the model (in this instance, repeatedly deleting entities from the database) before finally committing those changes by calling `Session`.

```
commit().
```

```
def delete(self):
    titles = request.POST.getall('title')
    pages = self.page_q.filter(Page.title.in_(titles))
    for page in pages:
        Session.delete(page)
    Session.commit()
    redirect_to('pages')
```

The [Object Relational tutorial](#) in the SQLAlchemy documentation covers a basic SQLAlchemy object-relational mapping scenario in much more detail and the [SQL Expression tutorial](#) covers the details of manipulating and marshalling the model entity objects.

5.7.4 Using multiple databases

In order to use multiple databases, in `MYAPP/model/meta.py` create as many instances of `Base` as there are databases to connect to:

```
"""SQLAlchemy Metadata and Session object"""
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker

__all__ = ['Base', 'Base2', 'Session']

# SQLAlchemy session manager. Updated by model.init_model()
Session = scoped_session(sessionmaker())

# The declarative Base
Base = declarative_base()
Base2 = declarative_base()
```

Declare the different database URLs in `development.ini`, appending an integer to the `sqlalchemy` keyword in order to differentiate between them.

```
sqlalchemy.url = sqlite:///%(here)s/database_one.sqlite
sqlalchemy.echo = true
sqlalchemy2.url = sqlite:///%(here)s/database_two.sqlite
sqlalchemy2.echo = false
```

In `MYAPP/config/environment.py`, pick up those db URL declarations by using the different keywords (in this example: `sqlalchemy` and `sqlalchemy2`). Create the engines and call `model.init_model()`, passing through both engines as parameters.

```
# Setup the SQLAlchemy database engine
# Engine 0
engine = engine_from_config(config, 'sqlalchemy.')
engine2 = engine_from_config(config, 'sqlalchemy2.')
model.init_model(engine, engine2)
```

Bind the engines appropriately to the `Base`-specific metadata in `MYAPP/model/__init__.py` - note `init_model()` is expecting both engines to be supplied as formal parameters.

```
def init_model(engine, engine2):
    meta.Base.metadata.bind = engine
    meta.Base2.metadata.bind = engine2
```

Then import Base and/or Base2

```
from MYAPP.model.meta import Base, Base2
```

and use as required, e.g.

```
class Author(Base2):
    __tablename__ = 'authors'
    id = Column(Integer, primary_key=True)
    keywords = relation("Keyword", secondary=keywords)
```

5.7.5 Avoiding the “circular imports” problem of model interdependency

Closely-interdependent models can sometimes cause “circular import” problems, where importing one model file causes a dependent model file to be imported, which then cause the first model file to be imported, and so on round and round in circles.

In order to break the circle, define the model entities as globals in MYAPP/model/meta.py

```
"""The application's model objects"""
import sqlalchemy as sa
from MYAPP.model import meta
from sqlalchemy.orm import scoped_session, sessionmaker

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

    import MYAPP.model.user
    User = MYAPP.model.user.User
    global User

    import MYAPP.model.newsletter
    Newsletter = MYAPP.model.newsletter.Newsletter
    global Newsletter

    import MYAPP.model.submission
    Submission = MYAPP.model.submission.Submission
    global Submission
```

5.7.6 Testing the Models

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

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

class TestModels(TestController):

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



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

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

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

5.8 Logging

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

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

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

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

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

SQLAlchemy has several other loggers you can configure in the same way. “`sqlalchemy.pool`” level `INFO` tells when connections are checked out from the engine’s connection pool and when they’re returned. “`sqlalchemy.orm`” and buddies log various ORM operations. See “Configuring Logging” in the [SQLAlchemy manual](#).

5.9 About SQLAlchemy

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

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

- The object-relational mapper (ORM) lets you interact with the database using your own object classes rather than writing SQL code.
- The SQL expression language has many methods to create customized SQL statements, and the result cursor is more friendly than DBAPI’s.
- The low-level execute methods accept literal SQL strings if you find something the SQL builder can’t do, such as adding a column to an existing table or modifying the column’s type. If they return results, you still get the benefit of SQLAlchemy’s result cursor.

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

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

5.9.1 SQLAlchemy add-ons

Most of these provide a higher-level ORM, either by combining the table definition and ORM class definition into one step, or supporting an “active record” style of access.

Please take the time to learn how to do things “the regular way” before using these shortcuts in a production application.

Understanding what these add-ons do behind the scenes will help if you have to troubleshoot a database error or work around a limitation in the add-on later.

[SQLSoup](#), an extension to SQLAlchemy, provides a quick way to generate ORM classes based on existing database tables.

If you're familiar with ActiveRecord, used in Ruby on Rails, then you may want to use the [Elixir](#) layer on top of SQLAlchemy. This approach is less common since the introduction of the declarative extension, but has other features the declarative does not.

Pylons works well with many different types of databases, in addition to other database object-relational mappers.

6.1 Advanced SQLAlchemy

6.1.1 Alternative SQLAlchemy Styles

In addition to the declarative style, SQLAlchemy has a default more verbose and explicit approach.

Definitions using the default SQLAlchemy approach

Here is a sample `model/__init__.py` with a “persons” table, based on the default SQLAlchemy approach:

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

from myapp.model import meta

def init_model(engine):
    meta.Session.configure(bind=engine)
    meta.engine = engine

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

class Person(object):
    pass
```

```
orm.mapper(Person, t_persons)
```

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

Relation example

Here’s an example of a *Person* and an *Address* class with a many:many relationship on *people.my_addresses*. See [Relational Databases for People in a Hurry](#) and the ‘[SQLAlchemy manual](#)’ for details.

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

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

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

class Person(object):
    pass

class Address(object):
    pass

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

Definitions using “reflection” of an existing database table

If the table already exists, SQLAlchemy can read the column definitions directly from the database. This is called *reflecting* the table.

The advantage of this approach is that it allows you to dispense with the task of specifying the column types in Python code.

Reflecting existing database tables must be done inside `init_model()` because to perform the reflection, a live database engine is required and this is not available when the module is imported. A live database engine is bound explicitly in the `init_model()` function and so enables reflection.

(An *engine* is a SQLAlchemy object that knows how to connect to a particular database.)

Here’s the second example with reflection:

```

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

from myapp.model import meta

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

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

t_persons = None

class Person(object):
    pass

```

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

Using the model standalone

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

```

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

```

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

```

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

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

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

# Create all tables, overwriting them if they exist.

```

```
if hasattr(model, "_Base"):\n    # SQLAlchemy 0.5 Declarative syntax\n    model._Base.metadata.drop_all(bind=engine, checkfirst=True)\n    model._Base.metadata.create_all(bind=engine)\nelse:\n    # SQLAlchemy 0.4 and 0.5 syntax without Declarative\n    meta.metadata.drop_all(bind=engine, checkfirst=True)\n    meta.metadata.create_all(bind=engine)\n\n# Create two records and insert them into the database using the ORM.\na = model.Person()\na.name = "Aaa"\na.email = "aaa@example.com"\nmeta.Session.add(a)\n\nb = model.Person()\nb.name = "Bbb"\nb.email = "bbb@example.com"\nmeta.Session.add(b)\n\nmeta.Session.commit()\n\n# Display all records in the persons table.\nprint "Database data:"\nfor p in meta.Session.query(model.Person):\n    print "id:", p.id\n    print "name:", p.name\n    print "email:", p.email\n    print
```

6.1.2 Talking to Multiple Databases at Once

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

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

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

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

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

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

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

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

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

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

binds= works the same way here too.

Multiple Application Instances

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

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

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

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

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

6.2 Non-SQLAlchemy libraries

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

Storm

Geniusql

6.2.1 DB-API

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

6.3 Object Databases

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

ZODB

Durus¹

6.4 Popular No-SQL Databases

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

[Schevo](#) uses Durus to combine some features of relational and object databases. It is written in Python.

[CouchDb](#) is a document-based database. It features a [Python API](#).

The Datastore database in Google App Engine.

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

Pylons comes with two main ways to configure an application:

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

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

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

The applications `config/` directory includes:

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

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

7.1 Runtime Configuration

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

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

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

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

See also:

Configuration file format **and options** are described in great detail in the [Paste Deploy documentation](#).

7.1.1 Getting Information From Configuration Files

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

```
from pylons import config
```

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

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

That can then be read in the projects code:

```
from pylons import config
cache_dir = config['cache.dir']
```

Or the current debug status like this:

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

Evaluating Non-string Data in Configuration Files

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

```
from paste.deploy.converters import asbool

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

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

7.1.2 Production Configuration Files

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

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

Generating the Production Configuration

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

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

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

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

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

7.2 Environment

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

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

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

7.3 URL Configuration

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

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

Changed in version 0.9.7: Prior to Routes 1.9, all `map.connect` statements required variable parts to begin with a `:` like `map.connect('/:controller/:action')`. This syntax is now optional, and the new `{}` syntax is recommended.

Any part of the path inside the curly braces is a variable (a *variable part*) that will match any text in the URL for that 'part'. A 'part' of the URL is the text between two forward slashes. Every part of the URL must be present for the *route* to match, otherwise a 404 will be returned.

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

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

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

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

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

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

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

```
class ArticlesController(BaseController):

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

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

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

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

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

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

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

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

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

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

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

7.3.1 Adding a route to match /

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

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

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

Note: By default, projects' static files (in the `public/` directory) are served in preference to controllers. New Pylons projects include a welcome page (`public/index.html`) that shows up at the `/` url. You'll want to remove this file before mapping a route there.

7.3.2 Generating URLs

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

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

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

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

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

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

See also:

[Routes manual](#) Full details and source code.

7.4 Middleware

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

The default stack that is setup for a Pylons application is described in detail in [WSGI Middleware](#).

Default middleware stack:

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

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

# CUSTOM MIDDLEWARE HERE (filtered by error handling middlewares)

if asbool(full_stack):
```

```
# Handle Python exceptions
app = ErrorHandler(app, global_conf, **config['pylons.errorware'])

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

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

if asbool(static_files):
    # Serve static files
    static_app = StaticURLParser(config['pylons.paths']['static_files'])
    app = Cascade([static_app, app])

return app
```

Since each piece of middleware wraps the one before it, the stack needs to be assembled in reverse order from the order in which its called. That is, the very last middleware that wraps the WSGI Application, is the very first that will be called by the server.

The last piece of middleware in the stack, called Cascade, is used to serve static content files during development. For top performance, consider disabling the Cascade middleware via setting the `static_files = false` in the configuration file. Then have the webserver or a [CDN](#) serve static files.

Warning: When unsure about whether or not to change the middleware, **don't**. The order of the middleware is important to the proper functioning of a Pylons application, and shouldn't be altered unless needed.

7.4.1 Adding custom middleware

Custom middleware should be included in the `config/middleware.py` at comment marker:

```
# CUSTOM MIDDLEWARE HERE (filtered by error handling middlewares)
```

For example, to add a middleware component named *MyMiddleware*, include it in `config/middleware.py`:

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

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

# CUSTOM MIDDLEWARE HERE (filtered by error handling middlewares)
app = MyMiddleware(app)
```

The app object is simply passed as a parameter to the *MyMiddleware* middleware which in turn should return a wrapped WSGI application.

Care should be taken when deciding in which layer to place custom middleware. In most cases middleware should be placed before the Pylons WSGI application and its supporting Routes/Session/Cache middlewares, however if the

middleware should run *after* the CacheMiddleware:

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

# MyMiddleware can only see the cache object, nothing *above* here
app = MyMiddleware(app)

app = CacheMiddleware(app, config)
```

7.4.2 What is full_stack?

In the Pylons ini file {development.ini or production.ini} this block determines if the flag full_stack is set to true or false:

```
[app:main]
use = egg:app_name
full_stack = true
```

The full_stack flag determines if the ErrorHandler and StatusCodeRedirect is included as a layer in the middleware wrapping process. The only condition in which this option would be set to *false* is if multiple Pylons applications are running and will be wrapped in the appropriate middleware elsewhere.

7.5 Application Setup

There are two kinds of ‘Application Setup’ that are occasionally referenced with regards to a project using Pylons.

- Setting up a new application
- Configuring project information and package dependencies

7.5.1 Setting Up a New Application

To make it easier to setup a new instance of a project, such as setting up the basic database schema, populating necessary defaults, etc. a setup script can be created.

In a Pylons project, the setup script to be run is located in the projects’ websetup.py file. The default script loads the projects configuration to make it easier to write application setup steps:

```
import logging

from helloworld.config.environment import load_environment

log = logging.getLogger(__name__)

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

Note: If the project was configured during creation to use SQLAlchemy this file will include some commands to setup the database connection to make it easier to setup database tables.

To run the setup script using the development configuration:

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

7.5.2 Configuring the Package

A newly created project with Pylons is a standard Python package. As a Python package, it has a `setup.py` file that records meta-information about the package. Most of the options in it are fairly self-explanatory, the most important being the ‘install_requires’ option:

```
install_requires=[
    "Pylons>=0.9.7",
],
```

These lines indicate what packages are required for the proper functioning of the application, and should be updated as needed. To re-parse the `setup.py` line for new dependencies:

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

In addition to updating the packages as needed so that the dependency requirements are made, this command will ensure that this package is active in the system (without requiring the traditional **`python setup.py install`**).

See also:

[Declaring Dependencies](#)

8.1 Logging messages

As of Pylons 0.9.6, Pylons controllers (created via `paste controller/restcontroller`) and `websetup.py` create their own Logger objects via [Python's logging module](#).

For example, in the `helloworld` project's `hello` controller (`helloworld/controllers/hello.py`):

```
import logging

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

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        ...
```

Python's special `__name__` variable refers to the current module's fully qualified name; in this case, `helloworld.controllers.hello`.

To log messages, simply use methods available on that Logger object:

```
import logging

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

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
```

```
content_type = 'text/plain'
content = 'Hello World!'

log.debug('Returning: %s (content-type: %s)', content, content_type)
response.content_type = content_type
return content
```

Which will result in the following printed to the console, on stderr:

```
16:20:20,440 DEBUG [helloworld.controllers.hello] Returning: Hello World!
              (content-type: text/plain)
```

8.2 Basic Logging configuration

As of Pylons 0.9.6, the default ini files include a basic configuration for the logging module. Paste ini files use the Python standard `ConfigParser` format; the same format used for the Python logging module's `Configuration file format`.

paster, when loading an application via the `paster serve`, `shell` or `setup-app` commands, calls the `logging.fileConfig` function on that specified ini file if it contains a 'loggers' entry. `logging.fileConfig` reads the logging configuration from a `ConfigParser` file.

Logging configuration is provided in both the default `development.ini` and the production ini file (created via `paster make-config <package_name> <ini_file>`). The production ini's logging setup is a little simpler than the `development.ini`'s, and is as follows:

```
# Logging configuration
[loggers]
keys = root

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

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

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

One root Logger is created that logs only messages at a level above or equal to the `INFO` level to `stderr`, with the following format:

```
2007-08-17 15:04:08,704 INFO [helloworld.controllers.hello] Loading resource, id: 86
```

For those familiar with the `logging.basicConfig` function, this configuration is equivalent to the code:

```
logging.basicConfig(level=logging.INFO,
format='% (asctime)s % (levelname)-5.5s [% (name)s] % (message)s')
```

The default `development.ini`'s logging section has a couple of differences: it uses a less verbose timestamp, and defaults your application's log messages to the `DEBUG` level (described in the next section).

Pylons and many other libraries (such as Beaker, SQLAlchemy, Paste) log a number of messages for debugging purposes. Switching the root Logger level to `DEBUG` reveals them:

```
[logger_root]
#level = INFO
level = DEBUG
handlers = console
```

8.3 Filtering log messages

Often there's too much log output to sift through, such as when switching the root Logger's level to `DEBUG`.

An example: you're diagnosing database connection issues in your application and only want to see SQLAlchemy's `DEBUG` messages in relation to database connection pooling. You can leave the root Logger's level at the less verbose `INFO` level and set that particular SQLAlchemy Logger to `DEBUG` on its own, apart from the root Logger:

```
[logger_sqlalchemy.pool]
level = DEBUG
handlers =
qualname = sqlalchemy.pool
```

then add it to the list of Loggers:

```
[loggers]
keys = root, sqlalchemy.pool
```

No Handlers need to be configured for this Logger as by default non root Loggers will propagate their log records up to their parent Logger's Handlers. The root Logger is the top level parent of all Loggers.

This technique is used in the default `development.ini`. The root Logger's level is set to `INFO`, whereas the application's log level is set to `DEBUG`:

```
# Logging configuration
[loggers]
keys = root, helloworld
```

```
[logger_helloworld]
level = DEBUG
handlers =
qualname = helloworld
```

All of the child Loggers of the helloworld Logger will inherit the `DEBUG` level unless they're explicitly set differently. Meaning the `helloworld.controllers.hello`, `helloworld.websetup` (and all your app's modules') Loggers by default have an effective level of `DEBUG` too.

For more advanced filtering, the logging module provides a [Filter](#) object; however it cannot be used directly from the configuration file.

8.4 Advanced Configuration

To capture log output to a separate file, use a `FileHandler` (or a `RotatingFileHandler`):

```
[handler_accesslog]
class = FileHandler
args = ('access.log', 'a')
level = INFO
formatter = generic
```

Before it's recognized, it needs to be added to the list of Handlers:

```
[handlers]
keys = console, accesslog
```

and finally utilized by a Logger.

```
[logger_root]
level = INFO
handlers = console, accesslog
```

These final 3 lines of configuration directs all of the root Logger's output to the `access.log` as well as the console; we'll want to disable this for the next section.

8.5 Request logging with Paste's TransLogger

Paste provides the `TransLogger` middleware for logging requests using the `Apache Combined Log Format`. `TransLogger` combined with a `FileHandler` can be used to create an `access.log` file similar to Apache's.

Like any standard middleware with a Paste entry point, `TransLogger` can be configured to wrap your application in the `[app:main]` section of the ini file:

```
filter-with = translogger

[filter:translogger]
use = egg:Paste#translogger
setup_console_handler = False
```

This is equivalent to wrapping your app in a `TransLogger` instance via the bottom of your project's `config/middleware.py` file:

```
from paste.translogger import TransLogger
app = TransLogger(app, setup_console_handler=False)
return app
```

`TransLogger` will automatically setup a logging Handler to the console when called with no arguments, so it 'just works' in environments that don't configure logging. Since we've configured our own logging Handlers, we need to disable that option via `setup_console_handler = False`.

With the filter in place, `TransLogger`'s Logger (named the 'wsgi' Logger) will propagate its log messages to the parent Logger (the root Logger), sending its output to the console when we request a page:

```
00:50:53,694 INFO [helloworld.controllers.hello] Returning: Hello World!
              (content-type: text/plain)
00:50:53,695 INFO [wsgi] 192.168.1.111 - - [11/Aug/2007:20:09:33 -0700] "GET /hello
```

```
HTTP/1.1" 404 - "-"
"Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.6) Gecko/20070725
Firefox/2.0.0.6"
```

To direct TransLogger to the `access.log` FileHandler defined above, we need to add that FileHandler to the wsgi Logger's list of Handlers:

```
# Logging configuration
[loggers]
keys = root, wsgi
```

```
[logger_wsgi]
level = INFO
handlers = handler_accesslog
qualname = wsgi
propagate = 0
```

As mentioned above, non-root Loggers by default propagate their log Records to the root Logger's Handlers (currently the console Handler). Setting `propagate` to 0 (false) here disables this; so the `wsgi` Logger directs its records only to the `accesslog` Handler.

Finally, there's no need to use the generic Formatter with TransLogger as TransLogger itself provides all the information we need. We'll use a Formatter that passes-through the log messages as is:

```
[formatters]
keys = generic, accesslog
```

```
[formatter_accesslog]
format = %(message)s
```

Then wire this new `accesslog` Formatter into the FileHandler:

```
[handler_accesslog]
class = FileHandler
args = ('access.log', 'a')
level = INFO
formatter = accesslog
```

8.6 Logging to wsgi.errors

Pylons provides a custom logging Handler class, `pylons.log.WSGIErrorHandler`, for logging output to `environ['wsgi.errors']`: the WSGI server's error stream (see the [WSGI Specification](#), [PEP 333](#) for more information). `wsgi.errors` can be useful to log to in certain situations, such as when deployed under Apache `mod_wsgi/mod_python`, where the `wsgi.errors` stream is the Apache error log.

To configure logging of only ERROR (and CRITICAL) messages to `wsgi.errors`, add the following to the ini file:

```
[handlers]
keys = console, wsgierrors
```

```
[handler_wsgierrors]
class = pylons.log.WSGIErrorHandler
args = ()
```

```
level = ERROR
format = generic
```

then add the new Handler name to the list of Handlers used by the root Logger:

```
[logger_root]
level = INFO
handlers = console, wsgierrors
```

Warning: `WSGIErrorsHandler` does not receive log messages created during application startup. This is due to the `wsgi.errors` stream only being available through the `environ` dictionary; which isn't available until a request is made.

8.6.1 Lumberjacking with log4j's Chainsaw

Java's `log4j` project provides the Java GUI application `Chainsaw` for viewing and managing log messages. Among its features are the ability to filter log messages on the fly, and customizable color highlighting of log messages.

We can configure Python's logging module to output to a format parsable by Chainsaw, `log4j`'s `XMLLayout` format.

To do so, we first need to install the `Python XMLLayout` package:

```
$ easy_install XMLLayout
```

It provides a log Formatter that generates `XMLLayout` XML. It also provides `RawSocketHandler`; like the logging module's `SocketHandler`, it sends log messages across the network, but does not pickle them.

The following is an example configuration for sending `XMLLayout` log messages across the network to Chainsaw, if it were listening on `localhost` port `4448`:

```
[handlers]
keys = console, chainsaw

[formatters]
keys = generic, xmllayout

[logger_root]
level = INFO
handlers = console, chainsaw
```

```
[handler_chainsaw]
class = xmllayout.RawSocketHandler
args = ('localhost', 4448)
level = NOTSET
formatter = xmllayout
```

```
[formatter_xmllayout]
class = xmllayout.XMLLayout
```

This configures any log messages handled by the root Logger to also be sent to Chainsaw. The default `development.ini` configures the root Logger to the `INFO` level, however in the case of using Chainsaw, it is preferable to configure the root Logger to `NOTSET` so *all* log messages are sent to Chainsaw. Instead, we can restrict the console handler to the `INFO` level:

```
[logger_root]
level = NOTSET
handlers = console

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

Chainsaw can be downloaded from its [home page](#), but can also be launched directly from a Java-enabled browser via the link: [Chainsaw web start](#).

It can be configured from the GUI, but it also supports reading its configuration from a `log4j.xml` file.

The following `log4j.xml` file configures Chainsaw to listen on port `4448` for XMLLayout style log messages. It also hides Chainsaw's own logging messages under the `WARN` level, so only your app's log messages are displayed:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration>
<configuration xmlns="http://logging.apache.org/">

  <plugin name="XMLSocketReceiver" class="org.apache.log4j.net.XMLSocketReceiver">
    <param name="decoder" value="org.apache.log4j.xml.XMLDecoder"/>
    <param name="port" value="4448"/>
  </plugin>

  <logger name="org.apache.log4j">
    <level value="warn"/>
  </logger>

  <root>
    <level value="debug"/>
  </root>

</configuration>
```

Chainsaw will prompt for a configuration file upon startup. The configuration can also be loaded later by clicking *File/Load Log4J File...* You should see an `XMLSocketReceiver` instance loaded in Chainsaw's Receiver list, configured at port `4448`, ready to receive log messages.

Here's how the Pylons stack's log messages can look with colors defined (using Chainsaw on OS X):

Timestamp	Level	Logger	Message	Thread	Class	Line	Method	Thread
13:10:06	DEBUG	routes.middleware	Route path: 'controller/action/id', defaults: {'action': 'u:index', 'controller': 'u:spotting', 'id': None}	CP WSGIServer Thread-2	middleware	82	__call__	
13:10:06	DEBUG	routes.middleware	Match dict: {'action': 'u:index', 'controller': 'u:spotting', 'id': None}	CP WSGIServer Thread-2	middleware	83	__call__	
13:10:06	DEBUG	pylons.wsgiapp	Setting up Pylons stacked object globals	CP WSGIServer Thread-2	wsgiapp	107	setup_app_env	
13:10:06	DEBUG	pylons.wsgiapp	Resolved URL to controller: 'u:spotting'	CP WSGIServer Thread-2	wsgiapp	159	resolve	
13:10:06	DEBUG	pylons.wsgiapp	Found controller, module: 'streetspotting.controllers.spotting', class: 'SpottingController'	CP WSGIServer Thread-2	wsgiapp	180	find_controller	
13:10:06	DEBUG	pylons.wsgiapp	Controller appears to be a class, instantiating	CP WSGIServer Thread-2	wsgiapp	209	dispatch	
13:10:06	DEBUG	pylons.wsgiapp	Calling controller class with WSGI interface	CP WSGIServer Thread-2	wsgiapp	213	dispatch	
13:10:06	DEBUG	pylons.controllers.core	Calling "before_" method with keyword args: {}	CP WSGIServer Thread-2	core	75	_inspect_call	
13:10:06	DEBUG	beaker.container	[MemoryContainer:tag_cache:popular_tags] lock_createfunc (waiting)	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[MemoryContainer:tag_cache:popular_tags] lock_createfunc (waited)	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[MemoryContainer:tag_cache:popular_tags] get_value creating new value	CP WSGIServer Thread-2	container	49	debug	
13:10:06	INFO	sqlalchemy.engine.base.Engine	SELECT tags.tag_id AS tags.tag_id, tags.tag_name AS tags.tag_name FROM tags JOIN spotting_tag...	CP WSGIServer Thread-2	base	576	_execute_raw	
13:10:06	INFO	sqlalchemy.engine.base.Engine	SELECT tags.tag_id AS tags.tag_id, tags.tag_name AS tags.tag_name FROM tags JOIN spotting_tag...	CP WSGIServer Thread-2	base	577	_execute_raw	
13:10:06	DEBUG	beaker.container	[MemoryContainer:tag_cache:popular_tags] set_value stored time 1187554206	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[MemoryContainer:tag_cache:popular_tags] unlock_createfunc	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	pylons.controllers.core	Looking for 'u:index' method to handle the request	CP WSGIServer Thread-2	core	105	_dispatch_call	
13:10:06	DEBUG	pylons.controllers.core	Calling 'index' method with keyword args: {}	CP WSGIServer Thread-2	core	75	_inspect_call	
13:10:06	DEBUG	pylons.decorators.cache	Wrapped with key: cache_default, expire: 1, type: dbm, query_args: False	CP WSGIServer Thread-2	cache	37	wrapper	
13:10:06	DEBUG	beaker.container	[streetspotting.controllers.spotting.index] data file /beans/home/pjenvey/src/python/streetspot...	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[streetspotting.controllers.spotting.index] opening dbm file /beans/home/pjenvey/src/python/stre...	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[streetspotting.controllers.spotting.index] closing dbm file /beans/home/pjenvey/src/python/stre...	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[DBMContainer:streetspotting.controllers.spotting.index:[]] lock_createfunc (didn't wait)	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[streetspotting.controllers.spotting.index] opening dbm file /beans/home/pjenvey/src/python/stre...	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[streetspotting.controllers.spotting.index] closing dbm file /beans/home/pjenvey/src/python/stre...	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	beaker.container	[DBMContainer:streetspotting.controllers.spotting.index:[]] get_value creating new value	CP WSGIServer Thread-2	container	49	debug	
13:10:06	DEBUG	pylons.decorators.cache	Creating new cache copy with key: [], type: dbm	CP WSGIServer Thread-2	cache	55	create_func	
13:10:06	DEBUG	streetspotting.controllers.spo...	Loading index page	CP WSGIServer Thread-2	spotting	47	index	
13:10:06	INFO	sqlalchemy.engine.base.Engine	SELECT count(spotting.spotting_id) FROM spottings	CP WSGIServer Thread-2	base	576	_execute_raw	
13:10:06	INFO	sqlalchemy.engine.base.Engine	SELECT count(spotting.spotting_id) FROM spottings	CP WSGIServer Thread-2	base	577	_execute_raw	

XMLSocketReceiver started!

Logger: sqlalchemy.engine.base.Engine (tx_8c)
Time: 2007-08-19 13:10:06,769
Thread: CP WSGIServer Thread-2
Message: SELECT count(spotting.spotting_id) FROM spottings

8.6.2 Alternate Logging Configuration style

Pylons' default ini files include a basic configuration for Python's logging module. Its format matches the standard Python logging module's [config file format](#). If a more concise format is preferred, here is Max Ischenko's demonstration of an alternative style to setup logging.

The following function is called at the application start up (e.g. Global ctor):

```
def setup_logging():
    logfile = config['logfile']
    if logfile == 'STDOUT': # special value, used for unit testing
        logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
            #format='%(name)s %(levelname)s %(message)s',
            #format='%(asctime)s,%(msecs)d %(levelname)s %(message)s',
            format='%(asctime)s,%(msecs)d %(name)s %(levelname)s %(message)s',
            datefmt='%H:%M:%S')
    else:
        logdir = os.path.dirname(os.path.abspath(logfile))
        if not os.path.exists(logdir):
            os.makedirs(logdir)
        logging.basicConfig(filename=logfile, mode='at+',
            level=logging.DEBUG,
            format='%(asctime)s,%(msecs)d %(name)s %(levelname)s %(message)s',
            datefmt='%Y-%b-%d %H:%M:%S')
```



```
setup_thirdparty_logging()
```

The `setup_thirdparty_logging` function searches through the certain keys of the application `.ini` file which specify logging level for a particular logger (module).

```
def setup_thirdparty_logging():
    for key in config:
        if not key.endswith('logging'):
            continue
        value = config.get(key)
        key = key.rstrip('.logging')
        loglevel = logging.getLevelName(value)
        log.info('Set %s logging for %s', logging.getLevelName(loglevel), key)
        logging.getLogger(key).setLevel(loglevel)
```

Relevant section of the `.ini` file (example):

```
sqlalchemy.logging = WARNING
sqlalchemy.orm.unitofwork.logging = INFO
sqlalchemy.engine.logging = DEBUG
sqlalchemy.orm.logging = INFO
routes.logging = WARNING
```

This means that routes logger (and all sub-loggers such as `routes.mapper`) only passes through messages of at least WARNING level; sqlalchemy defaults to WARNING level but some loggers are configured with more verbose level to aid debugging.

Helpers are functions intended for usage in templates, to assist with common HTML and text manipulation, higher level constructs like a HTML tag builder (that safely escapes variables), and advanced functionality like Pagination of data sets.

The majority of the helpers available in Pylons are provided by the `webhelpers` package. Some of these helpers are also used in controllers to prepare data for use in the template by other helpers, such as the `secure_form_tag()` function which has a corresponding `authenticate_form()`.

To make individual helpers available for use in templates under `h`, the appropriate functions need to be imported in `lib/helpers.py`. All the functions available in this file are then available under `h` just like any other module reference.

By customizing the `lib/helpers.py` module you can quickly add custom functions and classes for use in your templates.

Helper functions are organized into modules by theme. All HTML generators are under the `webhelpers_html` package, except for a few third-party modules which are directly under `webhelpers`. The `webhelpers` modules are separately documented, see `webhelpers`.

9.1 Pagination

Note: The *paginate* module is not compatible to the deprecated *pagination* module that was provided with former versions of the Webhelpers package.

9.1.1 Purpose of a paginator

When you display large amounts of data like a result from an SQL query then usually you cannot display all the results on a single page. It would simply be too much. So you divide the data into smaller chunks. This is what a paginator does. It shows one page of chunk of data at a time. Imagine you are providing a company phonebook through the web and let the user search the entries. Assume the search result contains 23 entries. You may decide to display no more

than 10 entries per page. The first page contains entries 1-10, the second 11-20 and the third 21-23. And you also show a navigational element like Page 1 of 3: [1] 2 3 that allows the user to switch between the available pages.

9.1.2 The Page class

The `webhelpers` package provides a `paginate` module that can be used for this purpose. It can create pages from simple Python lists as well as SQLAlchemy queries and SQLAlchemy select objects. The module provides a `Page` object that represents a single page of items from a larger result set. Such a `Page` mainly behaves like a list of items on that page. Let's take the above example of 23 items spread across 3 pages:

```
# Create a list of items from 1 to 23
>>> items = range(1,24)

# Import the paginate module
>>> import webhelpers.paginate

# Create a Page object from the 'items' for the second page
>>> page2 = webhelpers.paginate.Page(items, page=2, items_per_page=10)

# The Page object can be printed (__repr__) to show details on the page
>>> page2

Page:
Collection type: <type 'list'>
(Current) page: 2
First item: 11
Last item: 20
First page: 1
Last page: 3
Previous page: 1
Next page: 3
Items per page: 10
Number of items: 23
Number of pages: 3

# Show the items on this page
>>> list(page2)

[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

# Print the items in a for loop
>>> for i in page2: print "This is entry", i

This is entry 11
This is entry 12
This is entry 13
This is entry 14
This is entry 15
This is entry 16
This is entry 17
This is entry 18
This is entry 19
This is entry 20
```

There are further parameters to invoking a `Page` object. Please see `webhelpers.paginate.Page`

Note: Page numbers and item numbers start from 1. If you are accessing the items on the page by their index please note that the first item is `item[1]` instead of `item[0]`.

9.1.3 Switching between pages using a *pager*

The user needs a way to get to another page. This is usually done with a list of links like Page 3 of 41 - 1 2 [3] 4 5 .. 41. Such a list can be created by the Page's `pager()` method. Take the above example again:

```
>>> page2.pager()

<a class="pager_link" href="/content?page=1">1</a>
<span class="pager_curpage">2</span>
<a class="pager_link" href="/content?page=3">3</a>
```

Without the HTML tags it looks like 1 [2] 3. The links point to a URL where the respective page is found. And the current page (2) is highlighted.

The appearance of a pager can be customized. By default the format string is `~2~` which means it shows adjacent pages from the current page with a maximal radius of 2. In a larger set this would look like 1 .. 34 35 [36] 37 38 .. 176. The radius of 2 means that two pages before and after the current page 36 are shown.

Several special variables can be used in the format string. See `pager()` for a complete list. Some examples for a pager of 20 pages while being on page 10 currently:

```
>>> page.pager()

1 .. 8 9 [10] 11 12 .. 20

>>> page.pager('~4~')

1 .. 6 7 8 9 [10] 11 12 13 14 .. 20

>>> page.pager('Page $page of $page_count - ~3~')

Page 10 of 20 - 1 .. 7 8 9 [10] 11 12 13 .. 20

>>> page.pager('$link_previous $link_next ~2~')

< > 1 .. 8 9 [10] 11 12 .. 20

>>> page.pager('Items $first_item - $last_item / ~2~')

Items 91 - 100 / 1 .. 8 9 [10] 11 12 .. 20
```

9.1.4 Paging over an SQLAlchemy query

If the data to page over comes from a database via SQLAlchemy then the `paginate` module can access a query object directly. This is useful when using ORM-mapped models. Example:

```
>>> employee_query = Session.query(Employee)
>>> page2 = webhelpers.paginate.Page(
    employee_query,
    page=2,
```

```
items_per_page=10)
>>> for employee in page2: print employee.first_name

John
Jack
Joseph
Kay
Lars
Lynn
Pamela
Sandra
Thomas
Tim
```

The *paginate* module is smart enough to only query the database for the objects that are needed on this page. E.g. if a page consists of the items 11-20 then SQLAlchemy will be asked to fetch exactly that 10 rows through *LIMIT* and *OFFSET* in the actual SQL query. So you must not load the complete result set into memory and pass that. Instead always pass a *query* when creating a *Page*.

9.1.5 Paging over an SQLAlchemy select

SQLAlchemy also allows to run arbitrary SELECTs on database tables. This is useful for non-ORM queries. *paginate* can use such select objects, too. Example:

```
>>> selection = sqlalchemy.select([Employee.c.first_name])
>>> page2 = webhelpers.paginate.Page(
    selection,
    page=2,
    items_per_page=10,
    sqlalchemy_session=model.Session)
>>> for first_name in page2: print first_name

John
Jack
Joseph
Kay
Lars
Lynn
Pamela
Sandra
Thomas
Tim
```

The only difference to using SQLAlchemy *query* objects is that you need to pass an SQLAlchemy *session* via the `sqlalchemy_session` parameter. A bare `select` does not have a database connection assigned. But the session has.

9.1.6 Usage in a Pylons controller and template

A simple example to begin with.

Controller:

```
def list(self):
    c.employees = webhelpers.paginate.Page(
```

```

model.Session.query(model.Employee),
page = int(request.params['page']),
items_per_page = 5)
return render('/employees/list.mako')

```

Template:

```

${c.employees.pager('Page $page: $link_previous $link_next ~4~')}
<ul>
% for employee in c.employees:
    <li>${employee.first_name} ${employee.last_name}</li>
% endfor
</ul>

```

The *pager()* creates links to the previous URL and just sets the *page* parameter appropriately. That's why you need to pass the requested page number (`request.params['page']`) when you create a *Page*.

9.1.7 Partial updates with AJAX

Updating a page partially is easy. All it takes is a little Javascript that - instead of loading the complete page - updates just the part of the page containing the paginated items. The *pager()* method accepts an *onclick* parameter for that purpose. This value is added as an *onclick* parameter to the A-HREF tags. So the *href* parameter points to a URL that loads the complete page while the *onclick* parameter provides Javascript that loads a partial page. An example (using the jQuery Javascript library for simplification) may help explain that.

Controller:

```

def list(self):
    c.employees = webhelpers.paginate.Page(
        model.Session.query(model.Employee),
        page = int(request.params['page']),
        items_per_page = 5)
    if 'partial' in request.params:
        # Render the partial page
        return render('/employees/list-partial.mako')
    else:
        # Render the full page
        return render('/employees/list-full.mako')

```

Template *list-full.mako*:

```

<html>
  <head>
    ${webhelpers.html.tags.javascript_link('/public/jquery.js')}
  </head>
  <body>
    <div id="page-area">
      <%include file="list-partial.mako"%>
    </div>
  </body>
</html>

```

Template *list-partial.mako*:

```

${c.employees.pager(
    'Page $page: $link_previous $link_next ~4~',

```

```
onclick="$('#my-page-area').load('%s'); return false;")}  
<ul>  
% for employee in c.employees:  
    <li>${employee.first_name} ${employee.last_name}</li>  
% endfor  
</ul>
```

To avoid code duplication in the template the full template includes the partial template. If a partial page load is requested then just the `list-partial.mako` gets rendered. And if a full page load is requested then the `list-full.mako` is rendered which in turn includes the `list-partial.mako`.

The `%s` variable in the `onclick` string gets replaced with a URL pointing to the respective page with a `partial=1` added (the name of the parameter can be customized through the `partial_param` parameter). Example:

- `href` parameter points to `/employees/list?page=3`
- `onclick` parameter contains Javascript loading `/employees/list?page=3&partial=1`

jQuery's syntax to load a URL into a certain DOM object (e.g. a DIV) is simply:

```
$('#some-id').load('/the/url')
```

The advantage of this technique is that it degrades gracefully. If the user does not have Javascript enabled then a full page is loaded. And if Javascript works then a partial load is done through the `onclick` action.

9.2 Secure Form Tag Helpers

For prevention of Cross-site request forgery (CSRF) attacks.

Generates form tags that include client-specific authorization tokens to be verified by the destined web app.

Authorization tokens are stored in the client's session. The web app can then verify the request's submitted authorization token with the value in the client's session.

This ensures the request came from the originating page. See the wikipedia entry for [Cross-site request forgery](#) for more information.

Pylons provides an `authenticate_form` decorator that does this verification on the behalf of controllers.

These helpers depend on Pylons' `session` object. Most of them can be easily ported to another framework by changing the API calls.

The helpers are implemented in such a way that it should be easy for developers to create their own helpers if using helpers for AJAX calls.

`authentication_token()` returns the current authentication token, creating one and storing it in the session if it doesn't already exist.

`auth_token_hidden_field()` creates a hidden field containing the authentication token.

`secure_form()` is `form()` plus `auth_token_hidden_field()`.

10.1 The basics

When a user submits a form on a website the data is submitted to the URL specified in the *action* attribute of the `<form>` tag. The data can be submitted either via HTTP *GET* or *POST* as specified by the *method* attribute of the `<form>` tag. If your form doesn't specify an *action*, then it's submitted to the current URL, generally you'll want to specify an *action*. When a file upload field such as `<input type="file" name="file" />` is present, then the HTML `<form>` tag must also specify *enctype="multipart/form-data"* and *method* must be *POST*.

10.2 Getting Started

Add two actions that looks like this:

```
# in the controller

def form(self):
    return render('/form.mako')

def email(self):
    return 'Your email is: %s' % request.params['email']
```

Add a new template called *form.mako* in the *templates* directory that contains the following:

```
<form name="test" method="GET" action="/hello/email">
Email Address: <input type="text" name="email" />
<input type="submit" name="submit" value="Submit" />
</form>
```

If the server is still running (see the *Getting Started Guide*) you can visit <http://localhost:5000/hello/form> and you will see the form. Try entering the email address *test@example.com* and clicking Submit. The URL should change to <http://localhost:5000/hello/email?email=test%40example.com> and you should see the text *Your email is test@example.com*.

In Pylons all form variables can be accessed from the `request.params` object which behaves like a dictionary. The keys are the names of the fields in the form and the value is a string with all the characters entity decoded. For example note how the `@` character was converted by the browser to `%40` in the URL and was converted back ready for use in `request.params`.

Note: *request* and *response* are objects from the *WebOb* library. Full documentation on their attributes and methods is [here](#).

If you have two fields with the same name in the form then using the dictionary interface will return the first string. You can get all the strings returned as a list by using the `.getall()` method. If you only expect one value and want to enforce this you should use `.getone()` which raises an error if more than one value with the same name is submitted.

By default if a field is submitted without a value, the dictionary interface returns an empty string. This means that using `.get(key, default)` on `request.params` will only return a default if the value was not present in the form.

10.2.1 POST vs GET and the Re-Submitted Data Problem

If you change the *form.mako* template so that the method is *POST* and you re-run the example you will see the same message is displayed as before. However, the URL displayed in the browser is simply <http://localhost:5000/hello/email> without the query string. The data is sent in the body of the request instead of the URL, but Pylons makes it available in the same way as for GET requests through the use of *request.params*.

Note: If you are writing forms that contain password fields you should usually use POST to prevent the password being visible to anyone who might be looking at the user's screen.

When writing form-based applications you will occasionally find users will press refresh immediately after submitting a form. This has the effect of repeating whatever actions were performed the first time the form was submitted but often the user will expect that the current page be shown again. If your form was submitted with a POST, most browsers will display a message to the user asking them if they wish to re-submit the data, this will not happen with a GET so POST is preferable to GET in those circumstances.

Of course, the best way to solve this issue is to structure your code differently so:

```
# in the controller

def form(self):
    return render('/form.mako')

def email(self):
    # Code to perform some action based on the form data
    # ...
    redirect(url(controller='home', action='result'))

def result(self):
    return 'Your data was successfully submitted'
```

In this case once the form is submitted the data is saved and an HTTP redirect occurs so that the browser redirects to <http://localhost:5000/hello/result>. If the user then refreshes the page, it simply redisplay the message rather than re-performing the action.

10.3 Using the Helpers

Creating forms can also be done using WebHelpers, which comes with Pylons. Here is the same form created in the previous section but this time using the helpers:

```
${h.form(h.url(action='email'), method='get')}
Email Address: ${h.text('email')}
${h.submit('Submit')}
${h.end_form() }
```

Before doing this you'll have to import the helpers you want to use into your project's *lib/helpers.py* file; then they'll be available under Pylons' *h* global. Most projects will want to import at least these:

```
from webhelpers.html import escape, HTML, literal, url_escape
from webhelpers.html.tags import *
```

There are many other helpers for text formatting, container objects, statistics, and for dividing large query results into pages. See the WebHelpers documentation to choose the helpers you'll need.

10.4 File Uploads

File upload fields are created by using the *file* input field type. The *file_field* helper provides a shortcut for creating these form fields:

```
${h.file_field('myfile')}
```

The HTML form must have its *enctype* attribute set to *multipart/form-data* to enable the browser to upload the file. The *form* helper's *multipart* keyword argument provides a shortcut for setting the appropriate *enctype* value:

```
${h.form(h.url(action='upload'), multipart=True)}
Upload file: ${h.file_field('myfile')} <br />
File description: ${h.text_field('description')} <br />
${h.submit('Submit')}
${h.end_form() }
```

When a file upload has succeeded, the *request.POST* (or *request.params*) *MultiDict* will contain a *cgi.FieldStorage* object as the value of the field.

FieldStorage objects have three important attributes for file uploads:

filename The name of file uploaded as it appeared on the uploader's filesystem.

file A file(-like) object from which the file's data can be read: A python *tempfile* or a *StringIO* object.

value The content of the uploaded file, eagerly read directly from the file object.

The easiest way to gain access to the file's data is via the *value* attribute: it returns the entire contents of the file as a string:

```
def upload(self):
    myfile = request.POST['myfile']
    return 'Successfully uploaded: %s, size: %i, description: %s' % \
        (myfile.filename, len(myfile.value), request.POST['description'])
```

However reading the entire contents of the file into memory is undesirable, especially for large file uploads. A common means of handling file uploads is to store the file somewhere on the filesystem. The *FieldStorage* typically reads the

file onto filesystem, however to a non permanent location, via a python *tempfile* object (though for very small uploads it stores the file in a *StringIO* object instead).

Python *tempfiles* are secure file objects that are automatically destroyed when they are closed (including an implicit close when the object is garbage collected). One of their security features is that their path cannot be determined: a simple *os.rename* from the *tempfile*'s path isn't possible. Alternatively, *shutil.copyfileobj* can perform an efficient copy of the file's data to a permanent location:

```
permanent_store = '/uploads/'

class Uploader(BaseController):
    def upload(self):
        myfile = request.POST['myfile']
        permanent_file = open(os.path.join(permanent_store,
                                           myfile.filename.lstrip(os.sep)),
                             'w')

        shutil.copyfileobj(myfile.file, permanent_file)
        myfile.file.close()
        permanent_file.close()

    return 'Successfully uploaded: %s, description: %s' % \
        (myfile.filename, request.POST['description'])
```

Warning: The previous basic example allows any file uploader to overwrite any file in the *permanent_store* directory that your web application has permissions to.

Also note the use of *myfile.filename.lstrip(os.sep)* here: without it, *os.path.join* is unsafe. *os.path.join* won't join absolute paths (beginning with *os.sep*), i.e. *os.path.join('/uploads/', 'uploaded_file.txt') == 'uploaded_file.txt'*. Always check user submitted data to be used with *os.path.join*.

10.5 Validating user input with FormEncode

10.5.1 Validation the Quick Way

At the moment you could enter any value into the form and it would be displayed in the message, even if it wasn't a valid email address. In most cases this isn't acceptable since the user's input needs validating. The recommended tool for validating forms in Pylons is *FormEncode*.

For each form you create you also create a validation schema. In our case this is fairly easy:

```
import formencode

class EmailForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    email = formencode.validators.Email(not_empty=True)
```

Note: We usually recommend keeping form schemas together so that you have a single place you can go to update them. It's also convenient for inheritance since you can make new form schemas that build on existing ones. If you put your forms in a *models/form.py* file, you can easily use them throughout your controllers as *model.form.EmailForm* in

the case shown.

Our form actually has two fields, an email text field and a submit button. If extra fields are submitted FormEncode's default behavior is to consider the form invalid so we specify *allow_extra_fields = True*. Since we don't want to use the values of the extra fields we also specify *filter_extra_fields = True*. The final line specifies that the email field should be validated with an *Email()* validator. In creating the validator we also specify *not_empty=True* so that the email field will require input.

Pylons comes with an easy to use *validate* decorator, if you wish to use it import it in your *lib/base.py* like this:

```
# other imports

from pylons.decorators import validate
```

Using it in your controller is pretty straight-forward:

```
# in the controller

def form(self):
    return render('/form.mako')

@validate(schema=EmailForm(), form='form')
def email(self):
    return 'Your email is: %s' % self.form_result.get('email')
```

Validation only occurs on POST requests so we need to alter our form definition so that the method is a POST:

```
${h.form(h.url(action='email'), method='post')}
```

If validation is successful, the valid result dict will be saved as *self.form_result* so it can be used in the action. Otherwise, the action will be re-run as if it was a GET request to the controller action specified in *form*, and the output will be filled by FormEncode's *htmlfill* to fill in the form field errors. For simple cases this is really handy because it also avoids having to write code in your templates to display error messages if they are present.

This does exactly the same thing as the example above but works with the original form definition and in fact will work with any HTML form regardless of how it is generated because the *validate* decorator uses *formencode.htmlfill* to find HTML fields and replace them with the values were originally submitted.

Note: Python 2.3 doesn't support decorators so rather than using the *@validate()* syntax you need to put *email = validate(schema=EmailForm(), form='form')(email)* after the email function's declaration.

10.5.2 Validation the Long Way

The *validate* decorator covers up a bit of work, and depending on your needs it's possible you could need direct access to FormEncode abilities it smoothes over.

Here's the longer way to use the *EmailForm* schema:

```
# in the controller

def email(self):
    schema = EmailForm()
    try:
        form_result = schema.to_python(request.params)
    except formencode.validators.Invalid, error:
```

```
        return 'Invalid: %s' % error
    else:
        return 'Your email is: %s' % form_result.get('email')
```

If the values entered are valid, the schema's `to_python()` method returns a dictionary of the validated and coerced `form_result`. This means that you can guarantee that the `form_result` dictionary contains values that are valid and correct Python objects for the data types desired.

In this case the email address is a string so `request.params['email']` happens to be the same as `form_result['email']`. If our form contained a field for age in years and we had used a `formencode.validators.Int()` validator, the value in `form_result` for the age would also be the correct type; in this case a Python integer.

FormEncode comes with a useful set of validators but you can also easily create your own. If you do create your own validators you will find it very useful that all FormEncode schemas' `to_python()` methods take a second argument named `state`. This means you can pass the Pylons `c` object into your validators so that you can set any variables that your validators need in order to validate a particular field as an attribute of the `c` object. It can then be passed as the `c` object to the schema as follows:

```
c.domain = 'example.com'
form_result = schema.to_python(request.params, c)
```

The schema passes `c` to each validator in turn so that you can do things like this:

```
class SimpleEmail(formencode.validators.Email):
    def to_python(self, value, c):
        if not value.endswith(c.domain):
            raise formencode.validators.Invalid(
                'Email addresses must end in: %s' % \
                    c.domain, value, c)
        return formencode.validators.Email._to_python(self, value, c)
```

For this to work, make sure to change the `EmailForm` schema you've defined to use the new `SimpleEmail` validator. In other words,

```
email = formencode.validators.Email(not_empty=True)
# becomes:
email = SimpleEmail(not_empty=True)
```

In reality the invalid error message we get if we don't enter a valid email address isn't very useful. We really want to be able to redisplay the form with the value entered and the error message produced. Replace the line:

```
return 'Invalid: %s' % error
```

with the lines:

```
c.form_result = error.value
c.form_errors = error.error_dict or {}
return render('/form.mako')
```

Now we will need to make some tweaks to `form.mako`. Make it look like this:

```
#{h.form(h.url(action='email'), method='get')}

% if c.form_errors:
<h2>Please correct the errors</h2>
% else:
<h2>Enter Email Address</h2>
% endif
```

```

% if c.form_errors:
Email Address: ${h.text_field('email', value=c.form_result['email'] or '')}
<p>${c.form_errors['email']}</p>
% else:
Email Address: ${h.text_field('email')}
% endif

${h.submit('Submit')}
${h.end_form()}

```

Now when the form is invalid the *form.mako* template is re-rendered with the error messages.

10.6 Other Form Tools

If you are going to be creating a lot of forms you may wish to consider using [FormBuild](#) to help create your forms. To use it you create a custom Form object and use that object to build all your forms. You can then use the API to modify all aspects of the generation and use of all forms built with your custom Form by modifying its definition without any need to change the form templates.

Here is an one example of how you might use it in a controller to handle a form submission:

```

# in the controller

def form(self):
    results, errors, response = formbuild.handle(
        schema=Schema(), # Your FormEncode schema for the form
                        # to be validated
        template='form.mako', # The template containg the code
                            # that builds your form
        form=Form # The FormBuild Form definition you wish to use
    )
    if response:
        # The form validation failed so re-display
        # the form with the auto-generated response
        # containing submitted values and errors or
        # do something with the errors
        return response
    else:
        # The form validated, do something useful with results.
        ...

```

Full documentation of all features is available in the [FormBuild manual](#) which you should read before looking at [Using FormBuild in Pylons](#)

Looking forward it is likely Pylons will soon be able to use the TurboGears widgets system which will probably become the recommended way to build forms in Pylons.

Internationalization and Localization

11.1 Introduction

Internationalization and localization are means of adapting software for non-native environments, especially for other nations and cultures.

Parts of an application which might need to be localized might include:

- Language
- Date/time format
- Formatting of numbers e.g. decimal points, positioning of separators, character used as separator
- Time zones (UTC in internationalized environments)
- Currency
- Weights and measures

The distinction between internationalization and localization is subtle but important. Internationalization is the adaptation of products for potential use virtually everywhere, while localization is the addition of special features for use in a specific locale.

For example, in terms of language used in software, internationalization is the process of marking up all strings that might need to be translated whilst localization is the process of producing translations for a particular locale.

Pylons provides built-in support to enable you to internationalize language but leaves you to handle any other aspects of internationalization which might be appropriate to your application.

Note: Internationalization is often abbreviated as I18N (or i18n or I18n) where the number 18 refers to the number of letters omitted. Localization is often abbreviated L10n or l10n in the same manner. These abbreviations also avoid picking one spelling (internationalisation vs. internationalization, etc.) over the other.

In order to represent characters from multiple languages, you will need to utilize Unicode. This document assumes you have read the unicode.

By now you should have a good idea of what Unicode is, how to use it in Python and which areas of your application need to pay specific attention to decoding and encoding Unicode data.

This final section will look at the issue of making your application work with multiple languages.

Pylons uses the [Python gettext module](#) for internationalization. It is based off the [GNU gettext API](#).

11.2 Getting Started

Everywhere in your code where you want strings to be available in different languages you wrap them in the `__()` function. There are also a number of other translation functions which are documented in the API reference at <http://pylonshq.com/docs/module-pylons.i18n.translation.html>

Note: The `__()` function is a reference to the `ugettext()` function. `__()` is a convention for marking text to be translated and saves on keystrokes. `ugettext()` is the Unicode version of `gettext()`; it returns unicode strings.

In our example we want the string 'Hello' to appear in three different languages: English, French and Spanish. We also want to display the word 'Hello' in the default language. We'll then go on to use some plural words too.

Lets call our project `translate_demo`:

```
$ paster create -t pylons translate_demo
```

Now lets add a friendly controller that says hello:

```
$ cd translate_demo
$ paster controller hello
```

Edit `controllers/hello.py` to make use of the `__()` function everywhere where the string `Hello` appears:

```
import logging

from pylons.i18n import get_lang, set_lang

from translate_demo.lib.base import *

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        response.write('Default: %s<br />' % _('Hello'))
        for lang in ['fr', 'en', 'es']:
            set_lang(lang)
            response.write("%s: %s<br />" % (get_lang(), _('Hello')))
```

When writing wrapping strings in the gettext functions, it is important not to piece sentences together manually; certain languages might need to invert the grammars. Don't do this:

```
# BAD!
msg = _("He told her ")
msg += _("not to go outside.")
```

but this is perfectly acceptable:

```
# GOOD
msg = _("He told her not to go outside")
```

The controller has now been internationalized, but it will raise a `LanguageError` until we have setup the alternative language catalogs.

GNU gettext use three types of files in the translation framework.

11.2.1 POT (Portable Object Template) files

The first step in the localization process. A program is used to search through your project's source code and pick out every string passed to one of the translation functions, such as `_()`. This list is put together in a specially-formatted template file that will form the basis of all translations. This is the `.pot` file.

11.2.2 PO (Portable Object) files

The second step in the localization process. Using the POT file as a template, the list of messages are translated and saved as a `.po` file.

11.2.3 MO (Machine Object) files

The final step in the localization process. The PO file is run through a program that turns it into an optimized machine-readable binary file, which is the `.mo` file. Compiling the translations to machine code makes the localized program much faster in retrieving the translations while it is running.

GNU gettext provides a suite of command line programs for extracting messages from source code and working with the associated gettext catalogs. The [Babel](#) project provides pure Python alternative versions of these tools. Unlike the GNU gettext tool *xgettext*, Babel supports extracting translatable strings from Python templating languages (currently Mako and Genshi).

11.3 Using Babel



To use Babel, you must first install it via `easy_install`. Run the command:

```
$ easy_install Babel
```

Pylons (as of 0.9.6) includes some sane defaults for Babel's `distutils` commands in the `setup.cfg` file.

It also includes an extraction method mapping in the `setup.py` file. It is commented out by default, to avoid `distutils` warning about it being an unrecognized option when Babel is not installed. These lines should be uncommented before proceeding with the rest of this walk through:

```
message_extractors = {'translate_demo': [
    ('**.py', 'python', None),
    ('templates/**/*.mako', 'mako', None),
    ('public/**/*.ignore', 'ignore', None)]},
```

We'll use Babel to extract messages to a `.pot` file in your project's `i18n` directory. First, the directory needs to be created. Don't forget to add it to your revision control system if one is in use:

```
$ cd translate_demo
$ mkdir translate_demo/i18n
$ svn add translate_demo/i18n
```

Next we can extract all messages from the project with the following command:

```
$ python setup.py extract_messages
running extract_messages
extracting messages from translate_demo/__init__.py
extracting messages from translate_demo/websetup.py
...
extracting messages from translate_demo/tests/functional/test_hello.py
writing PO template file to translate_demo/i18n/translate_demo.pot
```

This will create a `.pot` file in the `i18n` directory that looks something like this:

```
# Translations template for translate_demo.
# Copyright (C) 2007 ORGANIZATION
# This file is distributed under the same license as the translate_demo project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2007.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: translate_demo 0.0.0\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2007-08-02 18:01-0700\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 0.9dev-r215\n"

#: translate_demo/controllers/hello.py:10 translate_demo/controllers/hello.py:13
msgid "Hello"
msgstr ""
```

The `.pot` details that appear here can be customized via the `extract_messages` configuration in your project's `setup.cfg` (See the [Babel Command-Line Interface Documentation](#) for all configuration options).

Next, we'll initialize a catalog (`.po` file) for the Spanish language:

```
$ python setup.py init_catalog -l es
running init_catalog
creating catalog 'translate_demo/i18n/es/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
```

Then we can edit the last line of the new Spanish `.po` file to add a translation of "Hello":

```
msgid "Hello"
msgstr "¡Hola!"
```

Finally, to utilize these translations in our application, we need to compile the `.po` file to a `.mo` file:

```
$ python setup.py compile_catalog
running compile_catalog
1 of 1 messages (100%) translated in 'translate_demo/i18n/es/LC_MESSAGES/translate_
→demo.po'
compiling catalog 'translate_demo/i18n/es/LC_MESSAGES/translate_demo.po' to
'translate_demo/i18n/es/LC_MESSAGES/translate_demo.mo'
```

We can also use the `update_catalog` command to merge new messages from the `.pot` to the `.po` files. For example, if we later added the following line of code to the end of `HelloController`'s `index` method:

```
response.write('Goodbye: %s' % _('Goodbye'))
```

We'd then need to re-extract the messages from the project, then run the `update_catalog` command:

```
$ python setup.py extract_messages
running extract_messages
extracting messages from translate_demo/__init__.py
extracting messages from translate_demo/websetup.py
...
extracting messages from translate_demo/tests/functional/test_hello.py
writing PO template file to translate_demo/i18n/translate_demo.pot
$ python setup.py update_catalog
running update_catalog
updating catalog 'translate_demo/i18n/es/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
```

We'd then edit our catalog to add a translation for “Goodbye”, and recompile the `.po` file as we did above.

For more information, see the [Babel documentation](#) and the [GNU Gettext Manual](#).

11.4 Back To Work

Next we'll need to repeat the process of creating a `.mo` file for the `en` and `fr` locales:

```
$ python setup.py init_catalog -l en
running init_catalog
creating catalog 'translate_demo/i18n/en/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
$ python setup.py init_catalog -l fr
running init_catalog
creating catalog 'translate_demo/i18n/fr/LC_MESSAGES/translate_demo.po' based on
'translate_demo/i18n/translate_demo.pot'
```

Modify the last line of the `fr` catalog to look like this:

```
#: translate_demo/controllers/hello.py:10 translate_demo/controllers/hello.py:13
msgid "Hello"
msgstr "Bonjour"
```

Since our original messages are already in English, the `en` catalog can stay blank; gettext will fallback to the original.

Once you've edited these new `.po` files and compiled them to `.mo` files, you'll end up with an `i18n` directory containing:

```
i18n/translate_demo.pot
i18n/en/LC_MESSAGES/translate_demo.mo
```

```
i18n/en/LC_MESSAGES/translate_demo.mo
i18n/es/LC_MESSAGES/translate_demo.po
i18n/es/LC_MESSAGES/translate_demo.mo
i18n/fr/LC_MESSAGES/translate_demo.po
i18n/fr/LC_MESSAGES/translate_demo.mo
```

11.5 Testing the Application

Start the server with the following command:

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

Test your controller by visiting <http://localhost:5000/hello>. You should see the following output:

```
Default: Hello
fr: Bonjour
en: Hello
es: ¡Hola!
```

You can now set the language used in a controller on the fly.

For example this could be used to allow a user to set which language they wanted your application to work in. You could save the value to the session object:

```
session['lang'] = 'en'
session.save()
```

then on each controller call the language to be used could be read from the session and set in your controller's `__before__()` method so that the pages remained in the same language that was previously set:

```
def __before__(self):
    if 'lang' in session:
        set_lang(session['lang'])
```

Pylons also supports defining the default language to be used in the configuration file. Set a `lang` variable to the desired default language in your `development.ini` file, and Pylons will automatically call `set_lang` with that language at the beginning of every request.

E.g. to set the default language to Spanish, you would add `lang = es` to your `development.ini`:

```
[app:main]
use = egg:translate_demo
lang = es
```

If you are running the server with the `--reload` option the server will automatically restart if you change the `development.ini` file. Otherwise restart the server manually and the output would this time be as follows:

```
Default: ¡Hola!
fr: Bonjour
en: Hello
es: ¡Hola!
```

11.6 Fallback Languages

If your code calls `_()` with a string that doesn't exist at all in your language catalog, the string passed to `_()` is returned instead.

Modify the last line of the hello controller to look like this:

```
response.write("%s %s, %s" % (_('Hello'), _('World'), _('Hi!')))
```

Warning: Of course, in real life breaking up sentences in this way is very dangerous because some grammars might require the order of the words to be different.

If you run the example again the output will be:

```
Default: ¡Hola!
fr: Bonjour World!
en: Hello World!
es: ¡Hola! World!
```

This is because we never provided a translation for the string 'World!' so the string itself is used.

Pylons also provides a mechanism for fallback languages, so that you can specify other languages to be used if the word is omitted from the main language's catalog.

In this example we choose `fr` as the main language but `es` as a fallback:

```
import logging

from pylons.i18n import set_lang

from translate_demo.lib.base import *

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        set_lang(['fr', 'es'])
        return "%s %s, %s" % (_('Hello'), _('World'), _('Hi!'))
```

If `Hello` is in the `fr` .mo file as `Bonjour`, `World` is only in `es` as `Mundo` and none of the catalogs contain `Hi!`, you'll get the multilingual message: `Bonjour Mundo, Hi!`. This is a combination of the French, Spanish and original (English in this case, as defined in our source code) words.

You can also add fallback languages after calling `set_lang` via the `pylons.i18n.add_fallback` function. Translations will be tested in the order you add them.

Note: Fallbacks are reset after calling `set_lang(lang)` – that is, fallbacks are associated with the currently selected language.

One case where using fallbacks in this way is particularly useful is when you wish to display content based on the languages requested by the browser in the `HTTP_ACCEPT_LANGUAGE` header. Typically the browser may submit a number of languages so it is useful to be add fallbacks in the order specified by the browser so that you always try to display words in the language of preference and search the other languages in order if a translation cannot be found.

The languages defined in the `HTTP_ACCEPT_LANGUAGE` header are available in Pylons as `request.languages` and can be used like this:

```
for lang in request.languages:
    add_fallback(lang)
```

11.7 Translations Within Templates

You can also use the `_()` function within templates in exactly the same way you do in code. For example, in a Mako template:

```
${_('Hello')}
```

would produce the string `'Hello'` in the language you had set.

Babel currently supports extracting gettext messages from Mako and Genshi templates. The Mako extractor also provides support for translator comments. Babel can be extended to extract messages from other sources via a [custom extraction method plugin](#).

Pylons (as of 0.9.6) automatically configures a Babel extraction mapping for your Python source code and Mako templates. This is defined in your project's `setup.py` file:

```
message_extractors = {'translate_demo': [
    ('**.py', 'python', None),
    ('templates/**/*.mako', 'mako', None),
    ('public/**', 'ignore', None)]},
```

For a project using Genshi instead of Mako, the Mako line might be replaced with:

```
('templates/**/*.html', 'genshi', None),
```

See [Babel's documentation on Message Extraction](#) for more information.

11.8 Lazy Translations

Occasionally you might come across a situation when you need to translate a string when it is accessed, not when the `_()` or other functions are called.

Consider this example:

```
import logging

from pylons.i18n import get_lang, set_lang

from translate_demo.lib.base import *

log = logging.getLogger(__name__)

text = _('Hello')

class HelloController(BaseController):

    def index(self):
        response.write('Default: %s<br />' % _('Hello'))
```



```

for lang in ['fr', 'en', 'es']:
    set_lang(lang)
response.write("%s: %s<br />" % (get_lang(), _('Hello')))
response.write('Text: %s<br />' % text)

```

If we run this we get the following output:

```

Default: Hello
['fr']: Bonjour
['en']: Good morning
['es']: Hola
Text: Hello

```

This is because the function `_('Hello')` just after the imports is called when the default language is `en` so the variable `text` gets the value of the English translation even though when the string was used the default language was Spanish.

The rule of thumb in these situations is to try to avoid using the translation functions in situations where they are not executed on each request. For situations where this isn't possible, perhaps because you are working with legacy code or with a library which doesn't support internationalization, you need to use lazy translations.

If we modify the above example so that the import statements and assignment to `text` look like this:

```

from pylons.i18n import get_lang, lazy_gettext, set_lang

from helloworld.lib.base import *

log = logging.getLogger(__name__)

text = lazy_gettext('Hello')

```

then we get the output we expected:

```

Default: Hello
['fr']: Bonjour
['en']: Good morning
['es']: Hola
Text: Hola

```

There are lazy versions of all the standard Pylons [translation functions](#).

There is one drawback to be aware of when using the lazy translation functions: they are not actually strings. This means that if our example had used the following code it would have failed with an error `cannot concatenate 'str' and 'LazyString' objects`:

```

response.write('Text: ' + text + '<br />')

```

For this reason you should only use the lazy translations where absolutely necessary and should always ensure they are converted to strings by calling `str()` or `repr()` before they are used in operations with real strings.

11.9 Producing a Python Egg

Finally you can produce an egg of your project which includes the translation files like this:

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

The `setup.py` automatically includes the `.mo` language catalogs your application needs so that your application can be distributed as an egg. This is done with the following line in your `setup.py` file:

```
package_data={'translate_demo': ['i18n/*/LC_MESSAGES/*.mo']},
```

11.10 Plural Forms

Pylons also provides the `ungettext()` function. It's designed for internationalizing plural words, and can be used as follows:

```
ungettext('There is %(num)d file here', 'There are %(num)d files here',
          n) % {'num': n}
```

Plural forms have a different type of entry in `.pot/.po` files, as described in [The Format of PO Files in GNU Gettext's Manual](#):

```
#: translate_demo/controllers/hello.py:12
#, python-format
msgid "There is %(num)d file here"
msgid_plural "There are %(num)d files here"
msgstr[0] ""
msgstr[1] ""
```

One thing to keep in mind is that other languages don't have the same plural forms as English. While English only has 2 plural forms, singular and plural, Slovenian has 4! That means that you *must* use `ungettext` for proper pluralization. Specifically, the following will not work:

```
# BAD!
if n == 1:
    msg = _("There was no dog.")
else:
    msg = _("There were no dogs.")
```

11.11 Summary

This document only covers the basics of internationalizing and localizing a web application.

GNU Gettext is an extensive library, and the GNU Gettext Manual is highly recommended for more information.

Babel also provides support for interfacing to the CLDR (Common Locale Data Repository), providing access to various locale display names, localized number and date formatting, etc.

You should also be able to internationalize and then localize your application using Pylons' support for GNU gettext.

11.12 Further Reading

<http://en.wikipedia.org/wiki/Internationalization>

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

Note: This is a work in progress. We hope the internationalization, localization and Unicode support in Pylons is now robust and flexible but we would appreciate hearing about any issues we have. Just drop a line to the pylons-discuss mailing list on Google Groups.

11.13 babel.core – Babel core classes

11.13.1 babel

Integrated collection of utilities that assist in internationalizing and localizing applications.

This package is basically composed of two major parts:

- tools to build and work with `gettext` message catalogs
- a Python interface to the CLDR (Common Locale Data Repository), providing access to various locale display names, localized number and date formatting, etc.

copyright

3. 2013 by the Babel Team.

license BSD, see LICENSE for more details.

11.13.2 Module Contents

class `babel.Locale` (*language*, *territory=None*, *script=None*, *variant=None*)

Representation of a specific locale.

```
>>> locale = Locale('en', 'US')
>>> repr(locale)
"Locale('en', territory='US')"
>>> locale.display_name
u'English (United States)'
```

A *Locale* object can also be instantiated from a raw locale string:

```
>>> locale = Locale.parse('en-US', sep='-')
>>> repr(locale)
"Locale('en', territory='US')"
```

Locale objects provide access to a collection of locale data, such as territory and language names, number and date format patterns, and more:

```
>>> locale.number_symbols['decimal']
u'.'
```

If a locale is requested for which no locale data is available, an *UnknownLocaleError* is raised:

```
>>> Locale.parse('en_XX')
Traceback (most recent call last):
...
UnknownLocaleError: unknown locale 'en_XX'
```

For more information see [RFC 3066](#).

character_order

The text direction for the language.

```
>>> Locale('de', 'DE').character_order
'left-to-right'
>>> Locale('ar', 'SA').character_order
'right-to-left'
```

currencies

Mapping of currency codes to translated currency names. This only returns the generic form of the currency name, not the count specific one. If an actual number is requested use the `babel.numbers.get_currency_name()` function.

```
>>> Locale('en').currencies['COP']
u'Colombian Peso'
>>> Locale('de', 'DE').currencies['COP']
u'Kolumbianischer Peso'
```

currency_formats

Locale patterns for currency number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').currency_formats['standard']
<NumberPattern u'\xa4#,##0.00'>
>>> Locale('en', 'US').currency_formats['accounting']
<NumberPattern u'\xa4#,##0.00;(\xa4#,##0.00)'>
```

currency_symbols

Mapping of currency codes to symbols.

```
>>> Locale('en', 'US').currency_symbols['USD']
u'$'
>>> Locale('es', 'CO').currency_symbols['USD']
u'US$'
```

date_formats

Locale patterns for date formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').date_formats['short']
<DateTimePattern u'M/d/yy'>
>>> Locale('fr', 'FR').date_formats['long']
<DateTimePattern u'd MMMM y'>
```

datetime_formats

Locale patterns for datetime formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en').datetime_formats['full']
u'{1} 'at' {0}'
>>> Locale('th').datetime_formats['medium']
u'{1} {0}'
```

datetime_skeletons

Locale patterns for formatting parts of a datetime.

```
>>> Locale('en').datetime_skeletons['MEd']
<DateTimePattern u'E, M/d'>
>>> Locale('fr').datetime_skeletons['MEd']
<DateTimePattern u'E dd/MM'>
>>> Locale('fr').datetime_skeletons['H']
<DateTimePattern u"HH 'h'">
```

day_period_rules

Day period rules for the locale. Used by *get_period_id*.

day_periods

Locale display names for various day periods (not necessarily only AM/PM).

These are not meant to be used without the relevant *day_period_rules*.

days

Locale display names for weekdays.

```
>>> Locale('de', 'DE').days['format']['wide'][3]
u'Donnerstag'
```

decimal_formats

Locale patterns for decimal number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').decimal_formats[None]
<NumberPattern u'#,#0.###'>
```

classmethod default (*category=None, aliases={*'el': 'el_GR', 'fi': 'fi_FI', 'bg': 'bg_BG', 'uk': 'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr': 'fr_FR', 'da': 'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs': 'bs_BA', 'cs': 'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es': 'es_ES', 'he': 'he_IL', 'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn': 'nn_NO', 'no': 'nb_NO', 'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH', 'ja': 'ja_JP', 'lv': 'lv_LV', 'lt': 'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th': 'th_TH', 'sl': 'sl_SI', 'it': 'it_IT', 'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS', 'pl': 'pl_PL'})

Return the system default locale for the specified category.

```
>>> for name in ['LANGUAGE', 'LC_ALL', 'LC_CTYPE', 'LC_MESSAGES']:
...     os.environ[name] = ''
>>> os.environ['LANG'] = 'fr_FR.UTF-8'
>>> Locale.default('LC_MESSAGES')
Locale('fr', territory='FR')
```

The following fallbacks to the variable are always considered:

- LANGUAGE

- LC_ALL
- LC_CTYPE
- LANG

Parameters

- **category** – one of the LC_XXX environment variable names
- **aliases** – a dictionary of aliases for locale identifiers

display_name

The localized display name of the locale.

```
>>> Locale('en').display_name
u'English'
>>> Locale('en', 'US').display_name
u'English (United States)'
>>> Locale('sv').display_name
u'svenska'
```

Type *unicode*

english_name

The english display name of the locale.

```
>>> Locale('de').english_name
u'German'
>>> Locale('de', 'DE').english_name
u'German (Germany)'
```

Type *unicode*

eras

Locale display names for eras.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').eras['wide'][1]
u'Anno Domini'
>>> Locale('en', 'US').eras['abbreviated'][0]
u'BC'
```

first_week_day

The first day of a week, with 0 being Monday.

```
>>> Locale('de', 'DE').first_week_day
0
>>> Locale('en', 'US').first_week_day
6
```

get_display_name (*locale=None*)

Return the display name of the locale using the given locale.

The display name will include the language, territory, script, and variant, if those are specified.

```
>>> Locale('zh', 'CN', script='Hans').get_display_name('en')
u'Chinese (Simplified, China)'
```

Parameters `locale` – the locale to use

get_language_name (*locale=None*)

Return the language of this locale in the given locale.

```
>>> Locale('zh', 'CN', script='Hans').get_language_name('de')
u'Chinesisch'
```

New in version 1.0.

Parameters `locale` – the locale to use

get_script_name (*locale=None*)

Return the script name in the given locale.

get_territory_name (*locale=None*)

Return the territory name in the given locale.

interval_formats

Locale patterns for interval formatting.

Note: The format of the value returned may change between Babel versions.

How to format date intervals in Finnish when the day is the smallest changing component:

```
>>> Locale('fi_FI').interval_formats['MEd']['d']
[u'E d. \u2013 ', u'E d.M.']
```

See also:

The primary API to use this data is `babel.dates.format_interval()`.

Return type `dict[str, dict[str, list[str]]]`

language = None

the language code

language_name

The localized language name of the locale.

```
>>> Locale('en', 'US').language_name
u'English'
```

languages

Mapping of language codes to translated language names.

```
>>> Locale('de', 'DE').languages['ja']
u'Japanisch'
```

See [ISO 639](#) for more information.

list_patterns

Patterns for generating lists

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en').list_patterns['start']
u'{0}, {1}'
>>> Locale('en').list_patterns['end']
u'{0}, and {1}'
>>> Locale('en_GB').list_patterns['end']
u'{0} and {1}'
```

measurement_systems

Localized names for various measurement systems.

```
>>> Locale('fr', 'FR').measurement_systems['US']
u'am\xe9ricain'
>>> Locale('en', 'US').measurement_systems['US']
u'US'
```

meta_zones

Locale display names for meta time zones.

Meta time zones are basically groups of different Olson time zones that have the same GMT offset and daylight savings time.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').meta_zones['Europe_Central']['long']['daylight']
u'Central European Summer Time'
```

New in version 0.9.

min_week_days

The minimum number of days in a week so that the week is counted as the first week of a year or month.

```
>>> Locale('de', 'DE').min_week_days
4
```

months

Locale display names for months.

```
>>> Locale('de', 'DE').months['format']['wide'][10]
u'Oktober'
```

classmethod negotiate (*preferred, available, sep='_', aliases={ 'el': 'el_GR', 'fi': 'fi_FI', 'bg': 'bg_BG', 'uk': 'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr': 'fr_FR', 'da': 'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs': 'bs_BA', 'cs': 'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es': 'es_ES', 'he': 'he_IL', 'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn': 'nn_NO', 'no': 'nb_NO', 'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH', 'ja': 'ja_JP', 'lv': 'lv_LV', 'lt': 'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th': 'th_TH', 'sl': 'sl_SI', 'it': 'it_IT', 'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS', 'pl': 'pl_PL' }*)

Find the best match between available and requested locale strings.


```
>>> Locale.negotiate(['de_DE', 'en_US'], ['de_DE', 'de_AT'])
Locale('de', territory='DE')
>>> Locale.negotiate(['de_DE', 'en_US'], ['en', 'de'])
Locale('de')
>>> Locale.negotiate(['de_DE', 'de'], ['en_US'])
```

You can specify the character used in the locale identifiers to separate the different components. This separator is applied to both lists. Also, case is ignored in the comparison:

```
>>> Locale.negotiate(['de-DE', 'de'], ['en-us', 'de-de'], sep='-')
Locale('de', territory='DE')
```

Parameters

- **preferred** – the list of locale identifiers preferred by the user
- **available** – the list of locale identifiers available
- **aliases** – a dictionary of aliases for locale identifiers

number_symbols

Symbols used in number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('fr', 'FR').number_symbols['decimal']
u','
```

ordinal_form

Plural rules for the locale.

```
>>> Locale('en').ordinal_form(1)
'one'
>>> Locale('en').ordinal_form(2)
'two'
>>> Locale('en').ordinal_form(3)
'few'
>>> Locale('fr').ordinal_form(2)
'other'
>>> Locale('ru').ordinal_form(100)
'other'
```

classmethod `parse(identifier, sep='_', resolve_likely_subtags=True)`

Create a *Locale* instance for the given locale identifier.

```
>>> l = Locale.parse('de-DE', sep='-')
>>> l.display_name
u'Deutsch (Deutschland)'
```

If the *identifier* parameter is not a string, but actually a *Locale* object, that object is returned:

```
>>> Locale.parse(l)
Locale('de', territory='DE')
```

This also can perform resolving of likely subtags which it does by default. This is for instance useful to figure out the most likely locale for a territory you can use 'und' as the language tag:

```
>>> Locale.parse('und_AT')
Locale('de', territory='AT')
```

Parameters

- **identifier** – the locale identifier string
- **sep** – optional component separator
- **resolve_likely_subtags** – if this is specified then a locale will have its likely subtag resolved if the locale otherwise does not exist. For instance `zh_TW` by itself is not a locale that exists but Babel can automatically expand it to the full form of `zh_hant_TW`. Note that this expansion is only taking place if no locale exists otherwise. For instance there is a locale `en` that can exist by itself.

Raises

- **ValueError** – if the string does not appear to be a valid locale identifier
- **UnknownLocaleError** – if no locale data is available for the requested locale

percent_formats

Locale patterns for percent number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').percent_formats[None]
<NumberPattern u'#,##0%'>
```

periods

Locale display names for day periods (AM/PM).

```
>>> Locale('en', 'US').periods['am']
u'AM'
```

plural_form

Plural rules for the locale.

```
>>> Locale('en').plural_form(1)
'one'
>>> Locale('en').plural_form(0)
'other'
>>> Locale('fr').plural_form(0)
'one'
>>> Locale('ru').plural_form(100)
'many'
```

quarters

Locale display names for quarters.

```
>>> Locale('de', 'DE').quarters['format']['wide'][1]
u'1. Quartal'
```

scientific_formats

Locale patterns for scientific number formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').scientific_formats[None]
<NumberPattern u'#E0'>
```

script = None
the script code

script_name
The localized script name of the locale if available.

```
>>> Locale('sr', 'ME', script='Latn').script_name
u'latinica'
```

scripts
Mapping of script codes to translated script names.

```
>>> Locale('en', 'US').scripts['Hira']
u'Hiragana'
```

See [ISO 15924](#) for more information.

territories
Mapping of script codes to translated script names.

```
>>> Locale('es', 'CO').territories['DE']
u'Alemania'
```

See [ISO 3166](#) for more information.

territory = None
the territory (country or region) code

territory_name
The localized territory name of the locale if available.

```
>>> Locale('de', 'DE').territory_name
u'Deutschland'
```

text_direction
The text direction for the language in CSS short-hand form.

```
>>> Locale('de', 'DE').text_direction
'ltr'
>>> Locale('ar', 'SA').text_direction
'rtl'
```

time_formats
Locale patterns for time formatting.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').time_formats['short']
<DateTimePattern u'h:mm a'>
```

```
>>> Locale('fr', 'FR').time_formats['long']
<DateTimePattern u'HH:mm:ss z'>
```

time_zones

Locale display names for time zones.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').time_zones['Europe/London']['long']['daylight']
u'British Summer Time'
>>> Locale('en', 'US').time_zones['America/St_Johns']['city']
u'St. John\u2019s'
```

unit_display_names

Display names for units of measurement.

See also:

You may want to use `babel.units.get_unit_name()` instead.

Note: The format of the value returned may change between Babel versions.

variant = None

the variant code

variants

Mapping of script codes to translated script names.

```
>>> Locale('de', 'DE').variants['1901']
u'Alte deutsche Rechtschreibung'
```

weekend_end

The day the weekend ends, with 0 being Monday.

```
>>> Locale('de', 'DE').weekend_end
6
```

weekend_start

The day the weekend starts, with 0 being Monday.

```
>>> Locale('de', 'DE').weekend_start
5
```

zone_formats

Patterns related to the formatting of time zones.

Note: The format of the value returned may change between Babel versions.

```
>>> Locale('en', 'US').zone_formats['fallback']
u'%(1)s (%(0)s)'
>>> Locale('pt', 'BR').zone_formats['region']
u'Hor\u00e9rio %s'
```

New in version 0.9.

```
babel.default_locale(category=None, aliases={'el': 'el_GR', 'fi': 'fi_FI', 'bg': 'bg_BG', 'uk':
      'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr': 'fr_FR', 'da':
      'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs': 'bs_BA', 'cs':
      'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es': 'es_ES', 'he': 'he_IL',
      'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn': 'nn_NO', 'no': 'nb_NO',
      'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH', 'ja': 'ja_JP', 'lv': 'lv_LV', 'lt':
      'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th': 'th_TH', 'sl': 'sl_SI', 'it': 'it_IT',
      'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS', 'pl': 'pl_PL'})
```

Returns the system default locale for a given category, based on environment variables.

```
>>> for name in ['LANGUAGE', 'LC_ALL', 'LC_CTYPE']:
...     os.environ[name] = ''
>>> os.environ['LANG'] = 'fr_FR.UTF-8'
>>> default_locale('LC_MESSAGES')
'fr_FR'
```

The “C” or “POSIX” pseudo-locales are treated as aliases for the “en_US_POSIX” locale:

```
>>> os.environ['LC_MESSAGES'] = 'POSIX'
>>> default_locale('LC_MESSAGES')
'en_US_POSIX'
```

The following fallbacks to the variable are always considered:

- LANGUAGE
- LC_ALL
- LC_CTYPE
- LANG

Parameters

- **category** – one of the LC_XXX environment variable names
- **aliases** – a dictionary of aliases for locale identifiers

```
babel.negotiate_locale(preferred, available, sep='_', aliases={'el': 'el_GR', 'fi': 'fi_FI', 'bg':
      'bg_BG', 'uk': 'uk_UA', 'tr': 'tr_TR', 'ca': 'ca_ES', 'de': 'de_DE', 'fr':
      'fr_FR', 'da': 'da_DK', 'fa': 'fa_IR', 'ar': 'ar_SY', 'mk': 'mk_MK', 'bs':
      'bs_BA', 'cs': 'cs_CZ', 'et': 'et_EE', 'gl': 'gl_ES', 'id': 'id_ID', 'es':
      'es_ES', 'he': 'he_IL', 'ru': 'ru_RU', 'nl': 'nl_NL', 'pt': 'pt_PT', 'nn':
      'nn_NO', 'no': 'nb_NO', 'ko': 'ko_KR', 'sv': 'sv_SE', 'km': 'km_KH',
      'ja': 'ja_JP', 'lv': 'lv_LV', 'lt': 'lt_LT', 'en': 'en_US', 'sk': 'sk_SK', 'th':
      'th_TH', 'sl': 'sl_SI', 'it': 'it_IT', 'hu': 'hu_HU', 'ro': 'ro_RO', 'is': 'is_IS',
      'pl': 'pl_PL'})
```

Find the best match between available and requested locale strings.

```
>>> negotiate_locale(['de_DE', 'en_US'], ['de_DE', 'de_AT'])
'de_DE'
>>> negotiate_locale(['de_DE', 'en_US'], ['en', 'de'])
'de'
```

Case is ignored by the algorithm, the result uses the case of the preferred locale identifier:

```
>>> negotiate_locale(['de_DE', 'en_US'], ['de_de', 'de_at'])
'de_DE'
```

```
>>> negotiate_locale(['de_DE', 'en_US'], ['de_de', 'de_at'])
'de_DE'
```

By default, some web browsers unfortunately do not include the territory in the locale identifier for many locales, and some don't even allow the user to easily add the territory. So while you may prefer using qualified locale identifiers in your web-application, they would not normally match the language-only locale sent by such browsers. To workaroud that, this function uses a default mapping of commonly used language-only locale identifiers to identifiers including the territory:

```
>>> negotiate_locale(['ja', 'en_US'], ['ja_JP', 'en_US'])
'ja_JP'
```

Some browsers even use an incorrect or outdated language code, such as “no” for Norwegian, where the correct locale identifier would actually be “nb_NO” (Bokmål) or “nn_NO” (Nynorsk). The aliases are intended to take care of such cases, too:

```
>>> negotiate_locale(['no', 'sv'], ['nb_NO', 'sv_SE'])
'nb_NO'
```

You can override this default mapping by passing a different *aliases* dictionary to this function, or you can bypass the behavior althogher by setting the *aliases* parameter to *None*.

Parameters

- **preferred** – the list of locale strings preferred by the user
- **available** – the list of locale strings available
- **sep** – character that separates the different parts of the locale strings
- **aliases** – a dictionary of aliases for locale identifiers

`babel.parse_locale(identifier, sep='_')`

Parse a locale identifier into a tuple of the form (language, territory, script, variant).

```
>>> parse_locale('zh_CN')
('zh', 'CN', None, None)
>>> parse_locale('zh_Hans_CN')
('zh', 'CN', 'Hans', None)
```

The default component separator is “_”, but a different separator can be specified using the *sep* parameter:

```
>>> parse_locale('zh-CN', sep='-')
('zh', 'CN', None, None)
```

If the identifier cannot be parsed into a locale, a *ValueError* exception is raised:

```
>>> parse_locale('not_a_LOCALE_String')
Traceback (most recent call last):
...
ValueError: 'not_a_LOCALE_String' is not a valid locale identifier
```

Encoding information and locale modifiers are removed from the identifier:

```
>>> parse_locale('it_IT@euro')
('it', 'IT', None, None)
>>> parse_locale('en_US.UTF-8')
('en', 'US', None, None)
```

```
>>> parse_locale('de_DE.iso885915@euro')
('de', 'DE', None, None)
```

See [RFC 4646](#) for more information.

Parameters

- **identifier** – the locale identifier string
- **sep** – character that separates the different components of the locale identifier

Raises **ValueError** – if the string does not appear to be a valid locale identifier

11.14 babel.localedata — Babel locale data

11.14.1 babel.localedata

Low-level locale data access.

note The *Locale* class, which uses this module under the hood, provides a more convenient interface for accessing the locale data.

copyright

3. 2013 by the Babel Team.

license BSD, see LICENSE for more details.

`babel.localedata.exists(name)`

Check whether locale data is available for the given locale.

Returns *True* if it exists, *False* otherwise.

Parameters **name** – the locale identifier string

`babel.localedata.exists(name)`

Check whether locale data is available for the given locale.

Returns *True* if it exists, *False* otherwise.

Parameters **name** – the locale identifier string

11.15 babel.dates – Babel date classes

11.15.1 babel.dates

Locale dependent formatting and parsing of dates and times.

The default locale for the functions in this module is determined by the following environment variables, in that order:

- LC_TIME,
- LC_ALL, and
- LANG

copyright

3. 2013 by the Babel Team.

license BSD, see LICENSE for more details.

11.15.2 Module Contents

class `babel.dates.DateTimeFormat` (*value, locale*)

format_frac_seconds (*num*)

Return fractional seconds.

Rounds the time's microseconds to the precision given by the number of digits passed in.

format_weekday (*char='E', num=4*)

Return weekday from parsed datetime according to format pattern.

```
>>> format = DateTimeFormat(date(2016, 2, 28), Locale.parse('en_US'))
>>> format.format_weekday()
u'Sunday'
```

'E': Day of week - Use one through three letters for the abbreviated day name, four for the full (wide) name, five for the narrow name, or six for the short name.

```
>>> format.format_weekday('E', 2)
u'Sun'
```

'e': Local day of week. Same as E except adds a numeric value that will depend on the local starting day of the week, using one or two letters. For this example, Monday is the first day of the week.

```
>>> format.format_weekday('e', 2)
'01'
```

'c': Stand-Alone local day of week - Use one letter for the local numeric value (same as 'e'), three for the abbreviated day name, four for the full (wide) name, five for the narrow name, or six for the short name.

```
>>> format.format_weekday('c', 1)
'1'
```

Parameters

- **char** – pattern format character ('e','E','c')
- **num** – count of format character

get_week_number (*day_of_period, day_of_week=None*)

Return the number of the week of a day within a period. This may be the week number in a year or the week number in a month.

Usually this will return a value equal to or greater than 1, but if the first week of the period is so short that it actually counts as the last week of the previous period, this function will return 0.

```
>>> format = DateTimeFormat(date(2006, 1, 8), Locale.parse('de_DE'))
>>> format.get_week_number(6)
1
```



```
>>> format = DateTimeFormat(date(2006, 1, 8), Locale.parse('en_US'))
>>> format.get_week_number(6)
2
```

Parameters

- **day_of_period** – the number of the day in the period (usually either the day of month or the day of year)
- **day_of_week** – the week day; if omitted, the week day of the current date is assumed

class babel.dates.**DateTimePattern** (*pattern, format*)

11.16 babel.numbers – Babel number classes

11.16.1 babel.numbers

Locale dependent formatting and parsing of numeric data.

The default locale for the functions in this module is determined by the following environment variables, in that order:

- LC_NUMERIC,
- LC_ALL, and
- LANG

copyright

3. 2013 by the Babel Team.

license BSD, see LICENSE for more details.

11.16.2 Module Contents

class babel.numbers.**NumberFormatError**

Exception raised when a string cannot be parsed into a number.

class babel.numbers.**NumberPattern** (*pattern, prefix, suffix, grouping, int_prec, frac_prec, exp_prec, exp_plus*)

babel.numbers.**format_number** (*number, locale='en_US_POSIX'*)

Return the given number formatted for a specific locale.

```
>>> format_number(1099, locale='en_US')
u'1,099'
>>> format_number(1099, locale='de_DE')
u'1.099'
```

Parameters

- **number** – the number to format
- **locale** – the *Locale* object or locale identifier

babel.numbers.**format_decimal** (*number, format=None, locale='en_US_POSIX'*)

Return the given decimal number formatted for a specific locale.

```
>>> format_decimal(1.2345, locale='en_US')
u'1.234'
>>> format_decimal(1.2346, locale='en_US')
u'1.235'
>>> format_decimal(-1.2346, locale='en_US')
u'-1.235'
>>> format_decimal(1.2345, locale='sv_SE')
u'1,234'
>>> format_decimal(1.2345, locale='de')
u'1,234'
```

The appropriate thousands grouping and the decimal separator are used for each locale:

```
>>> format_decimal(12345.5, locale='en_US')
u'12,345.5'
```

Parameters

- **number** – the number to format
- **format** –
- **locale** – the *Locale* object or locale identifier

`babel.numbers.format_percent` (*number*, *format*=None, *locale*='en_US_POSIX')

Return formatted percent value for a specific locale.

```
>>> format_percent(0.34, locale='en_US')
u'34%'
>>> format_percent(25.1234, locale='en_US')
u'2,512%'
>>> format_percent(25.1234, locale='sv_SE')
u'2\xa0512\xa0%'
```

The format pattern can also be specified explicitly:

```
>>> format_percent(25.1234, u'#,##0\u2030', locale='en_US')
u'25,123\u2030'
```

Parameters

- **number** – the percent number to format
- **format** –
- **locale** – the *Locale* object or locale identifier

`babel.numbers.format_scientific` (*number*, *format*=None, *locale*='en_US_POSIX')

Return value formatted in scientific notation for a specific locale.

```
>>> format_scientific(10000, locale='en_US')
u'1E4'
```

The format pattern can also be specified explicitly:

```
>>> format_scientific(1234567, u'###0E00', locale='en_US')
u'1.23E06'
```

Parameters

- **number** – the number to format
- **format** –
- **locale** – the *Locale* object or locale identifier

`babel.numbers.parse_number(string, locale='en_US_POSIX')`
 Parse localized number string into an integer.

```
>>> parse_number('1,099', locale='en_US')
1099
>>> parse_number('1.099', locale='de_DE')
1099
```

When the given string cannot be parsed, an exception is raised:

```
>>> parse_number('1.099,98', locale='de')
Traceback (most recent call last):
...
NumberFormatError: '1.099,98' is not a valid number
```

Parameters

- **string** – the string to parse
- **locale** – the *Locale* object or locale identifier

Returns the parsed number

Raises *NumberFormatError* – if the string can not be converted to a number

`babel.numbers.parse_decimal(string, locale='en_US_POSIX')`
 Parse localized decimal string into a decimal.

```
>>> parse_decimal('1,099.98', locale='en_US')
Decimal('1099.98')
>>> parse_decimal('1.099,98', locale='de')
Decimal('1099.98')
```

When the given string cannot be parsed, an exception is raised:

```
>>> parse_decimal('2,109,998', locale='de')
Traceback (most recent call last):
...
NumberFormatError: '2,109,998' is not a valid decimal number
```

Parameters

- **string** – the string to parse
- **locale** – the *Locale* object or locale identifier

Raises *NumberFormatError* – if the string can not be converted to a decimal number

12.1 Sessions

Pylons includes a session object: a session is a server-side, semi-permanent storage for data associated with a client. The session object is provided by the [Beaker library](#) which also provides caching functionality as described in [Caching](#).

12.2 The Session Object

The Pylons session object is available at `pylons.session`. Controller modules created via **paster controller/restcontroller** import the session object by default.

The basic session API is simple, it implements a dict-like interface with a few additional methods. The following is an example of using the session to store a token identifying if a client is logged in.

```
class LoginController(BaseController):

    def authenticate(self):
        name = request.POST['name']
        password = request.POST['password']
        user = Session.query(User).filter_by(name=name,
                                             password=password).first()

        if user:
            msg = 'Successfully logged in as %s' % name
            location = url('index')
            session['logged_in'] = True
            session.save()
        else:
            msg = 'Invalid username/password'
            location = url('login')
        flash(msg)
        redirect(location)
```

```
def logout(self):  
    # Clear all values in the session associated with the client  
    session.clear()  
    session.save()
```

Subsequent requests can then determine if a client is logged in or not by checking the session:

```
if not session.get('logged_in'):  
    flash('Please login')  
    redirect(url('login'))
```

The session object acts lazily: it does not load the session data (from disk or whichever backend is used) until the data is first accessed. This laziness is facilitated via an intermediary `beaker.session.SessionObject` that wraps the actual `beaker.session.Session` object. Furthermore the session will not write changes to its backend without an explicit call to its `beaker.session.Session.save()` method (unless configured with the `auto` option).

Session data is generally serialized for storage via the Python `pickle` module, so anything stored in the session must be pickleable.

The lightweight `SessionObject` wrapper is created by the: `beaker.middleware.SessionMiddleware` WSGI middleware. `SessionMiddleware` stores the wrapper in the WSGI environ where Pylons sets a reference to it from `pylons.session`.

Sessions are associated with a client via a client-side cookie. The WSGI middleware is also responsible for sending said cookie to the client.

12.3 Configuring the Session

The basic session defaults are:

- File based sessions (session data is stored on disk)
- Session cookies have no expiration date (cookies expire at the end of the browser session)
- Session cookie domain/path matches the current host/path

Pylons projects by default sets the following couple of session options via their `.ini` files. All Beaker specific session options in the ini file are prefixed with `beaker.session`:

```
cache_dir = %(here)s/data  
beaker.session.key = foo  
beaker.session.secret = somesecret
```

`cache_dir` acts a base directory for both session and cache storage. Session data is stored in this location under a `sessions/` sub-directory.

`session.key` is the name attribute of the cookie sent to the browser. This defaults to your project's name.

`session.secret` is the secret token used to hash the cookie data sent to the client. This should be a secret, ideally randomly generated value on production environments. **paster make-config** will generate a random secret for you when creating a production ini file.

12.3.1 Other Session Options

Some other commonly used session options are:

- `type` The type of the back-end for storing session data. Beaker supports many different backends, see [Beaker Configuration Documentation](#) for the choices. Defaults to 'file'.
- `cookie_domain` The domain name to use for the session Cookie. For example, when using sub-domains, set this to the parent domain name so that the cookie is valid for all sub-domains.

To enable pure [Cookie-based Sessions](#) and force the cookie domain to be valid for all sub-domains of 'example.com', add the following to your Pylons ini file:

```
beaker.session.type = cookie
beaker.session.cookie_domain = .example.com
```

See the [Beaker Configuration Documentation](#) for an exhaustive list of Session options.

12.4 Storing SQLAlchemy mapped objects in Beaker sessions

Mapped objects from SQLAlchemy can be serialized into the beaker session, but care must be taken when retrieving these objects back from the beaker session. They will not be associated with the SQLAlchemy Unit-of-Work Session, however these objects can be reconciled via the SQLAlchemy Session's `merge` method, as follows:

```
address = DBSession.query(Address).get(id)
session[id] = address
...
address = session.get(id)
address = DBSession.merge(address)
```

12.5 Custom and caching middleware

Care should be taken when deciding in which layer to place custom middleware. In most cases middleware should be placed *between* the Pylons WSGI application instantiation and the Routes middleware; however, if the middleware should run *before* the session object or routing is handled:

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

# MyMiddleware can only see the cache object, nothing *above* here
app = MyMiddleware(app)

app = CacheMiddleware(app, config)
```

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

12.6 Using *Session* in Internationalization

How to set the language used in a controller on the fly.

For example this could be used to allow a user to set which language they wanted your application to work in. Save the value to the session object:

```
session['lang'] = 'en'
session.save()
```

then on each controller call the language to be used could be read from the session and set in the controller's `__before__()` method so that the pages remained in the same language that was previously set:

```
def __before__(self):
    if 'lang' in session:
        set_lang(session['lang'])
```

12.7 Using *Session* in Secure Forms

Authorization tokens are stored in the client's session. The web app can then verify the request's submitted authorization token with the value in the client's session.

This ensures the request came from the originating page. See the wikipedia entry for [Cross-site request forgery](#) for more information.

Pylons provides an `authenticate_form` decorator that does this verification on the behalf of controllers.

These helpers depend on Pylons' `session` object. Most of them can be easily ported to another framework by changing the API calls.

12.8 Hacking the session for no cookies

(From a [paste #441](#) baked by Ben Bangert)

Set the session to not use cookies in the dev.ini file

```
beaker.session.use_cookies = False
```

with this as the *mode d'emploi* in the controller action

```
from beaker.session import Session as BeakerSession

# Get the actual session object through the global proxy
real_session = session._get_current_obj()

# Duplicate the session init options to avoid screwing up other sessions in
# other threads
params = real_session.__dict__['_params']

# Now set the id param used to make a session to our session maker,
# if id is None, a new id will be made automatically
params['id'] = find_id_func()
real_session.__dict__['_sess'] = BeakerSession({}, **params)

# Now we can use the session as usual
session['fred'] = 42
session.save()

# At the end, we need to see if the session was used and handle its id
if session.is_new:
```



```
# do something with session.id to make sure its around next time
pass
```

12.9 Using middleware (Beaker) with a composite app

How to allow called WSGI apps to share a common session management utility.

(From a paste #616 baked by Mark Luffel)

```
# Here's an example of configuring multiple apps to use a common
# middleware filter
# The [app:home] section is a standard pylons app
# The ``/servicebroker`` and ``/proxy`` apps both want to be able
# to use the same session management

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000

[filter-app:main]
use = egg:Beaker#beaker_session
next = sessioned
beaker.session.key = my_project_key
beaker.session.secret = i_wear_two_layers_of_socks

[composite:sessioned]
use = egg:Paste#urlmap
/ = home
/servicebroker = servicebroker
/proxy = cross_domain_proxy

[app:servicebroker]
use = egg:Appcelerator#service_broker

[app:cross_domain_proxy]
use = egg:Appcelerator#cross_domain_proxy

[app:home]
use = egg:my_project
full_stack = true
cache_dir = %(here)s/data
```


Inevitably, there will be occasions during applications development or deployment when some task is revealed to be taking a significant amount of time to complete. When this occurs, the best way to speed things up is with *caching*.

Caching is enabled in Pylons using *Beaker*, the same package that provides session handling. Beaker supports a variety of caching backends: in-memory, database, Google Datastore, filesystem, and memcached. Additional extensions are available that support Tokyo Cabinet, Redis, Dynamite, and Ringo. Back-ends can be added with Beaker's extension system.

See also:

[Beaker Extension Add-ons](#)

13.1 Types of Caching

Pylons offers a variety of caching options depending on the granularity of caching desired. Fine-grained caching down to specific sub-sections of a template, arbitrary Python functions, all the way up to entire controller actions and browser-side full-page caching are available.

Available caching options (ordered by granularity, least to most specific):

- **Browser-side** - HTTP/1.1 supports the *ETag* caching system that allows the browser to use its own cache instead of requiring regeneration of the entire page. ETag-based caching avoids repeated generation of content but if the browser has never seen the page before, the page will still be generated. Therefore using ETag caching in conjunction with one of the other types of caching listed here will achieve optimal throughput and avoid unnecessary calls on resource-intensive operations.
- **Controller Actions** - A Pylons controller action can have its entire result cached, including response headers if desired.
- **Templates** - The results of an entire rendered template can be cached using the 3 `cache` keyword arguments to the `render` calls. These render commands can also be used inside templates.
- **Arbitrary Functions** - Any function can be independently cached using Beaker's cache decorators. This allows fine-grained caching of just the parts of the code that can be cached.

- **Template Fragments** - Built-in caching options are available for both [Mako](#) and [Myghty](#) template engines. They allow fine-grained caching of only certain sections of the template. This is also sometimes called fragment caching since individual fragments of a page can be cached.

13.2 Namespaces and Keys

[Beaker](#) is used for caching arbitrary Python functions, template results, and in [Mako](#) for caching individual `<def>` blocks. Browser-side caching does *not* utilize [Beaker](#).

The two primary concepts to bear in mind when caching with [Beaker](#) are:

1. Caches have a *namespace*, this is to organize a cache such that variations of the same thing being cached are associated under a single place.
2. Variations of something being cached, are *keys* which are under that namespace.

For example, if we want to cache a function, the function name along with a unique name for it would be considered the *namespace*. The arguments it takes to differentiate the output to cache, are the *keys*.

An example of caching with the `cache_region()` decorator:

```
@cache_region('short_term', 'search_func')
def get_results(search_param):
    # do something to retrieve data
    data = get_data(search_param)
    return data

results = get_results('gophers')
```

In this example, the namespace will be the function name + module + 'search_func'. Since a single module might have multiple methods of the same name you wish to cache, the `cache_region()` decorator takes another argument in addition to the region to use, which is added to the namespace.

The key in this example is the *search_param* value. For each value of it, a separate result will be cached.

See also:

Stephen Pierzchala's [Caching for Performance](#) (stephen@pierzchala.com) [Beaker Caching Docs](#)

13.3 Configuring

[Beaker](#)'s cache options can be easily configured in the project's INI file. [Beaker's configuration documentation](#) explains how to setup the most common options.

The cache options specified will be used in the absence of more specific keyword arguments to individual cache functions. Functions that support *Cache Regions* will use the settings for that region.

13.3.1 Cache Regions

Cache regions are named groupings of related options. For example, in many web applications, there might be a few cache strategies used in a company, with short-term cached objects ending up in Memcached, and longer-term cached objects stored in the filesystem or a database.

Using cache regions makes it easy to declare the cache strategies in one place, then use them throughout the application by referencing the cache strategy name.

Cache regions should be setup in the `development.ini` file, but can also be configured and passed directly into the *CacheManager* instance that is created in the `lib/app_globals.py` file.

Example INI section for two cache regions (put these under your *[app:main]* section):

```
beaker.cache.regions = short_term, long_term
beaker.cache.short_term.type = ext:memcached
beaker.cache.short_term.url = 127.0.0.1:11211
beaker.cache.short_term.expire = 3600

beaker.cache.long_term.type = ext:database
beaker.cache.long_term.url = mysql://dbuser:dbpass@127.0.0.1/cache_db
beaker.cache.long_term.expire = 86400
```

This sets up two cache regions, *short_term* and *long_term*.

13.4 Browser-Side

Browser-side caching can utilize one of several methods. The entire page can have cache headers associated with it to indicate to the browser that it should be cached. Or, using the ETag Cache header, a page can have more fine-grained caching rules applied.

13.4.1 Cache Headers

Cache headers may be set directly on the *Response* object by setting the headers directly using the `headers()` property, or by using the cache header helpers.

To ensure pages aren't accidentally cached in dynamic web applications, Pylons default behavior sets the *Pragma* and *Cache-Control* headers to *no-cache*. Before setting cache headers, these default values should be cleared.

Clearing the default *no-cache* response headers:

```
class SampleController(BaseController):
    def index(self):
        # Clear the default cache headers
        del response.headers['Cache-Control']
        del response.headers['Pragma']

        return render('/index.html')
```

Using the response cache helpers:

```
# Set an action response to expires in 30 seconds
class SampleController(BaseController):
    def index(self):
        # Clear the default cache headers
        del response.headers['Cache-Control']
        del response.headers['Pragma']

        response.cache_expires(seconds=30)
        return render('/index.html')

# Set the cache-control to private with a max-age of 30 seconds
class SampleController(BaseController):
    def index(self):
        # Clear the default cache headers
```

```
del response.headers['Cache-Control']
del response.headers['Pragma']

response.cache_control = {'max-age': 30, 'public': True}
return render('/index.html')
```

All of the values that can be passed to the *cache_control* property dict, also may be passed into the *cache_expires* function call. It's recommended that you use the *cache_expires* helper as it also sets the Last-Modified and Expires headers to the second interval as well.

See also:

[Cache Control Header RFC](#)

13.4.2 E-Tag Caching

Caching via ETag involves sending the browser an ETag header so that it knows to save and possibly use a cached copy of the page from its own cache, instead of requesting the application to send a fresh copy.

Because the ETag cache relies on sending headers to the browser, it works in a slightly different manner to the other caching mechanisms.

The `etag_cache()` function will set the proper HTTP headers if the browser doesn't yet have a copy of the page. Otherwise, a 304 HTTP Exception will be thrown that is then caught by Paste middleware and turned into a proper 304 response to the browser. This will cause the browser to use its own locally-cached copy.

ETag-based caching requires a single key which is sent in the ETag HTTP header back to the browser. The [RFC specification for HTTP headers](#) indicates that an ETag header merely needs to be a string. This value of this string does not need to be unique for every URL as the browser itself determines whether to use its own copy, this decision is based on the URL and the ETag key.

```
def my_action(self):
    etag_cache('somekey')
    return render('/show.myt', cache_expire=3600)
```

Or to change other aspects of the response:

```
def my_action(self):
    etag_cache('somekey')
    response.headers['content-type'] = 'text/plain'
    return render('/show.myt')
```

The frequency with which an ETag cache key is changed will depend on the web application and the developer's assessment of how often the browser should be prompted to fetch a fresh copy of the page.

13.5 Controller Actions

The `beaker_cache()` decorator is for caching the results of a complete controller action.

Example:

```
from pylons.decorators.cache import beaker_cache

class SampleController(BaseController):

    # Cache this controller action forever (until the cache dir is
```

```
# cleaned)
@beaker_cache()
def home(self):
    c.data = expensive_call()
    return render('/home.myt')

# Cache this controller action by its GET args for 10 mins to memory
@beaker_cache(expire=600, type='memory', query_args=True)
def show(self, id):
    c.data = expensive_call(id)
    return render('/show.myt')
```

By default the decorator uses a composite of all of the decorated function's arguments as the cache key. It can alternatively use a composite of the *request.GET* query args as the cache key when the *query_args* option is enabled.

The cache key can be further customized via the *key* argument.

Warning: By default, the `beaker_cache()` decorator will cache the entire response object. This means the headers that were generated during the action will be cached as well. This can be disabled by providing `cache_response = False` to the decorator.

13.6 Templates

All render commands have caching functionality built in. To use it, merely add the appropriate cache keyword to the render call.

```
class SampleController(BaseController):
    def index(self):
        # Cache the template for 10 mins
        return render('/index.html', cache_expire=600)

    def show(self, id):
        # Cache this version of the template for 3 mins
        return render('/show.html', cache_key=id, cache_expire=180)

    def feed(self):
        # Cache for 20 mins to memory
        return render('/feed.html', cache_type='memory', cache_expire=1200)

    def home(self, user):
        # Cache this version of a page forever (until the cache dir
        # is cleaned)
        return render('/home.html', cache_key=user, cache_expire='never')
```

Note: At the moment, these functions do not support the use of cache region pre-defined argument sets.

13.7 Arbitrary Functions

Any Python function that returns a pickle-able result can be cached using [Beaker](#). The recommended way to cache functions is to use the `cache_region()` decorator. This decorator requires the [Cache Regions](#) to be configured.

Using the `cache_region()` decorator:

```
@cache_region('short_term', 'search_func')
def get_results(search_param):
    # do something to retrieve data
    data = get_data(search_param)
    return data

results = get_results('gophers')
```

See also:

[Beaker Caching Documentation](#)

13.7.1 Invalidating

A cached function can be manually invalidated by using the `region_invalidate()` function.

Example:

```
region_invalidate(get_results, None, 'search_func', search_param)
```

13.8 Fragments

Individual template files, and `<def>` blocks within them can be independently cached. Since the caching system utilizes [Beaker](#), any available [Beaker](#) back-ends are present in [Mako](#) as well.

Example:

```
<%def name="mycomp" cached="True" cache_timeout="30" cache_type="memory">
    other text
</%def>
```

See also:

[Mako Caching Documentation](#)

Unit and functional testing

14.1 Unit Testing with `webtest`

Pylons provides powerful unit testing capabilities for your web application utilizing `webtest` to emulate requests to your web application. You can then ensure that the response was handled appropriately and that the controller set things up properly.

To run the test suite for your web application, Pylons utilizes the `nose` test runner/discovery package. Running `nosetests` in your project directory will run all the tests you create in the tests directory. If you don't have nose installed on your system, it can be installed via `setuptools` with:

```
$ easy_install -U nose
```

To avoid conflicts with your development setup, the tests use the `test.ini` configuration file when run. This means **you must configure any databases, etc. in your `test.ini` file or your tests will not be able to find the database configuration.**

Warning: Nose can trigger errors during its attempt to search for doc tests since it will try and import all your modules one at a time *before* your app was loaded. This will cause files under `models/` that rely on your app to be running, to fail.

Pylons 0.9.6.1 and later includes a plugin for nose that loads the app before the doctests scan your modules, allowing models to be doctested. You can use this option from the command line with nose:

```
nosetests --with-pylons=test.ini
```

Or by setting up a `[nosetests]` block in your `setup.cfg`:

```
[nosetests]
verbose=True
verbosity=2
with-pylons=test.ini
```

```
detailed-errors=1
with-doctest=True
```

Then just run:

```
python setup.py nosetests
```

to run the tests.

14.2 Example: Testing a Controller

First let's create a new project and controller for this example:

```
$ paster create -t pylons TestExample
$ cd TestExample
$ paster controller comments
```

You'll see that it creates two files when you create a controller. The stub controller, and a test for it under `testexample/tests/functional/`.

Modify the `testexample/controllers/comments.py` file so it looks like this:

```
from testexample.lib.base import *

class CommentsController(BaseController):

    def index(self):
        return 'Basic output'

    def sess(self):
        session['name'] = 'Joe Smith'
        session.save()
        return 'Saved a session'
```

Then write a basic set of tests to ensure that the controller actions are functioning properly, modify `testexample/tests/functional/test_comments.py` to match the following:

```
from testexample.tests import *

class TestCommentsController(TestController):
    def test_index(self):
        response = self.app.get(url(controller='/comments'))
        assert 'Basic output' in response

    def test_sess(self):
        response = self.app.get(url(controller='/comments', action='sess'))
        assert response.session['name'] == 'Joe Smith'
        assert 'Saved a session' in response
```

Run `nosetests` in your main project directory and you should see them all pass:

```
..
-----
Ran 2 tests in 2.999s

OK
```

Unfortunately, a plain assert does not provide detailed information about the results of an assertion should it fail, unless you specify it a second argument. For example, add the following test to the `test_sess` function:

```
assert response.session.has_key('address') == True
```

When you run `nosetests` you will get the following, not-very-helpful result:

```
.F
=====
FAIL: test_sess (testexample.tests.functional.test_comments.TestCommentsController)
-----
Traceback (most recent call last):
File "~/TestExample/testexample/tests/functional/test_comments.py", line 12, in test_
↪sess
assert response.session.has_key('address') == True
AssertionError:

-----
Ran 2 tests in 1.417s

FAILED (failures=1)
```

You can augment this result by doing the following:

```
assert response.session.has_key('address') == True, "address not found in session"
```

Which results in:

```
.F
=====
FAIL: test_sess (testexample.tests.functional.test_comments.TestCommentsController)
-----
Traceback (most recent call last):
File "~/TestExample/testexample/tests/functional/test_comments.py", line 12, in test_
↪sess
assert response.session.has_key('address') == True
AssertionError: address not found in session

-----
Ran 2 tests in 1.417s

FAILED (failures=1)
```

But detailing every assert statement could be time consuming. Our `TestController` subclasses the standard Python `unittest.TestCase` class, so we can use utilize its helper methods, such as `assertEqual`, that can automatically provide a more detailed `AssertionError`. The new test line looks like this:

```
self.assertEqual(response.session.has_key('address'), True)
```

Which provides the more useful failure message:

```
.F
=====
FAIL: test_sess (testexample.tests.functional.test_comments.TestCommentsController)
-----
Traceback (most recent call last):
```

```
File "~/TestExample/testexample/tests/functional/test_comments.py", line 12, in test_
↪ sess
self.assertEqual(response.session.has_key('address'), True)
AssertionError: False != True
```

14.3 Testing Pylons Objects

Pylons will provide several additional attributes for the `webtest.WebTest.TestResponse` object that let you access various objects that were created during the web request:

config The configured Pylons applications.

session Session object

req Request object

tmpl_context Object containing variables passed to templates

app_globals Globals object

To use them, merely access the attributes of the response *after* you've used a `get/post` command:

```
response = app.get('/some/url')
assert response.session['var'] == 4
assert 'REQUEST_METHOD' in response.req.environ
```

Note: The response object already has a `TestRequest` object assigned to it, therefore Pylons assigns its request object to the response as `req`.

14.3.1 Accessing Special Globals

Sometimes, you might wish to modify or check a global Pylons variable such as `app_globals` before running the rest of your unit tests. The non-request specific variables are available from a special URL that will respond only in unit testing situations.

For example, to get the `app_globals` object without sending a request to your actual applications:

```
response = app.get('/_test_vars')
app_globals = response.app_globals
```

14.4 Testing Your Own Objects

WebTest's fixture testing allows you to designate your own objects that you'd like to access in your tests. This powerful functionality makes it easy to test the value of objects that are normally only retained for the duration of a single request.

Before making objects available for testing, it's useful to know when your application is being tested. WebTest will provide an environ variable called `paste.testing` that you can test for the presence and truth of so that your application only populates the testing objects when it has to.

Populating the `webtest` response object with your objects is done by adding them to the environ dict under the key `paste.testing_variables`. Pylons creates this dict before calling your application, so testing for its existence

and adding new values to it is recommended. All variables assigned to the `paste.testing_variables` dict will be available on the response object with the key being the attribute name.

Note: WebTest is an extracted stand-alone version of a Paste component called `paste.fixture`. For backwards compatibility, WebTest continues to honor the `paste.testing_variables` key in the `environ`.

Example:

```
# testexample/lib/base.py

from pylons import request
from pylons.controllers import WSGIController
from pylons.templating import render_mako as render

class BaseController(WSGIController):
    def __call__(self, environ, start_response):
        # Create a custom email object
        email = MyCustomEmailObj()
        email.name = 'Fred Smith'
        if 'paste.testing_variables' in request.environ:
            request.environ['paste.testing_variables']['email'] = email
        return WSGIController.__call__(self, environ, start_response)

# testexample/tests/functional/test_controller.py
from testexample.tests import *

class TestCommentsController(TestController):
    def test_index(self):
        response = self.app.get(url(controller='/'))
        assert response.email.name == 'Fred Smith'
```

See also:

WebTest Documentation Documentation covering webtest and its usage

WebTest Module docs Module API reference for methods available for use when testing the application

14.5 Unit Testing

XXX: Describe unit testing an applications models, libraries

14.6 Functional Testing

XXX: Describe functional/integrated testing, WebTest

Errors, Troubleshooting, and Debugging

When a web application has an error in production, a few different options for handling it are available. Pylons comes with error handlers to allow the following options:

- E-mail the traceback as HTML to the administrators
- Show the *Interactive Debugging* interface to the developer
- Programmatically handle the error in another controller
- Display a plain error on the web page

Some of these options can be combined by enabling or disabling the appropriate middleware.

15.1 Error Middleware

In a new Pylons project, the error handling middleware is configured in the projects `config/middleware.py`:

```
# Excerpt of applicable section

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

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

The first middleware configured, `ErrorHandler()`, actually configures one of two `WebError` middlewares depending on whether the project is in debug mode or not. If it is in debug mode, then the *Interactive Debugging* is enabled, otherwise, the *e-mail error handling* will be used.

The second middleware configured is the `StatusCodeRedirect` middleware. This middleware watches the request, and if the application returns a response containing one of the status code's listed, it will call back into the application to the error controller, and use that output instead.

None of these are required for a Pylons project to run, and commenting them all out results in the plain text of the error to display on the web page.

Warning: If no middleware at all is used, the error will appear on the screen in its entirety, *including full traceback output*.

15.1.1 Recommended Configurations

- For plain-text output or errors and non-200 status codes, comment out the `StatusCodeRedirect`. Tracebacks will be e-mailed to you in production, and the *Interactive Debugging* will be used during development.
- For programmatic error and non-200 status code handling, keep the stack as-is.
- To *not* have tracebacks e-mailed, remove only the `ErrorHandler()` middleware. This will also disable *Interactive Debugging* however. To retain *Interactive Debugging* but disable traceback e-mails:

```
if asbool(config['debug']):
    app = ErrorHandler(app, global_conf, **config['pylons.errorware'])
```

Note: To only capture specific non-200 status codes, the `StatusCodeRedirect` middleware can be passed a list of the codes that it should intercept and redirect to the error controller. When in non-debug mode, it captures the 400-404, and 500 status codes. Altering the list will capture more or less types of requests as desired.

15.1.2 Avoiding Displaying Tracebacks

When disabling the `ErrorHandler()` middleware, a replacement middleware should be created and used that captures exceptions and changes them into a normal WSGI response, otherwise the raw traceback error will be displayed on the browser.

An example middleware that just captures exceptions and changes them to a 500 error:

```
from webob import Request, Response

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

    def __call__(self, environ, start_response):
        req = Request(environ)
        try:
            response = req.get_response(self.app)
        except:
            response = Response()
            response.status_int = 500
            response.body = 'An error has occurred'
        return response(environ, start_response)
```

Replacing the `ErrorHandler` with this middleware will cause tracebacks to not be displayed to the user.

15.2 Interactive Debugging

Things break, and when they do, quickly pinpointing what went wrong and why makes a huge difference. By default, Pylons uses a customized version of [Ian Bicking's](#) `EvalException` middleware that also includes full Mako/Myghty Traceback information.

15.2.1 The Debugging Screen

The debugging screen has three tabs at the top:

Traceback Provides the raw exception trace with the interactive debugger

Extra Data Displays CGI, WSGI variables at the time of the exception, in addition to configuration information

Template Human friendly traceback for Mako or Myghty templates

Since Mako and Myghty compile their templates to Python modules, it can be difficult to accurately figure out what line of the template resulted in the error. The *Template* tab provides the full Mako or Myghty traceback which contains accurate line numbers for your templates, and where the error originated from. If your exception was triggered before a template was rendered, no Template information will be available in this section.

15.2.2 Example: Exploring the Traceback

Using the interactive debugger can also be useful to gain a deeper insight into objects present only during the web request like the `session` and `request` objects.

To trigger an error so that we can explore what's happening just raise an exception inside an action you're curious about. In this example, we'll raise an error in the action that's used to display the page you're reading this on. Here's what the docs controller looks like:

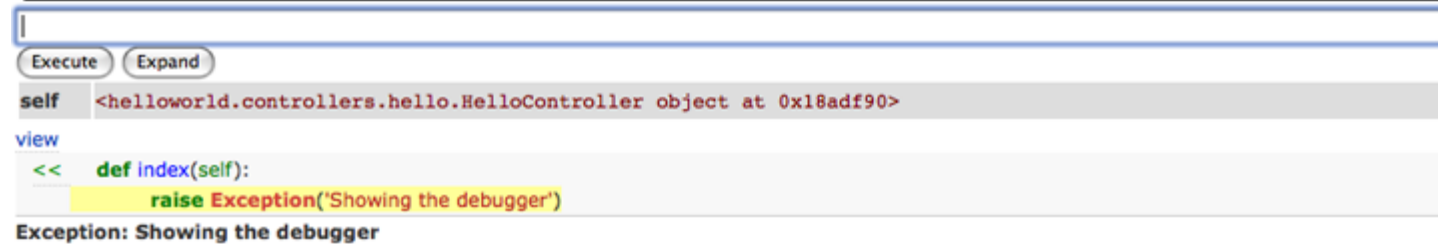
```
class DocsController(BaseController):
    def view(self, url):
        if request.path_info.endswith('docs'):
            redirect(url('/docs/'))
        return render('/docs/' + url)
```

Since we want to explore the `session` and `request`, we'll need to bind them first. Here's what our action now looks like with the binding and raising an exception:

```
def view(self, url):
    raise "hi"
    if request.path_info.endswith('docs'):
        redirect(url('/docs/'))
    return render('/docs/' + url)
```

Here's what exploring the Traceback from the above example looks like (Excerpt of the relevant portion):

```
>>> dir(self)
['_call__', '__class__', '__delattr__', '__dict__', '__doc__', '__getattr__', '__hash__', '__init__',
 '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__', '__weakref__',
 '_dispatch_call', '_get_method_args', '_inspect_call', '_perform_call', '_py_object', '_pylons_log_debug',
 'index', 'start_response']
>>> session
{}
```



15.3 E-mailing Errors

You can make various of changes to how the debugging works. For example if you disable the `debug` variable in the config file Pylons will email you an error report instead of displaying it as long as you provide your email address at the top of the config file:

```
error_email_from = you@example.com
```

This is very useful for a production site. Emails are sent via SMTP so you need to specify a valid SMTP server too.

15.4 Programmatically Handling Errors

By default, the `StatusCodeRedirect` will redirect any response with the designated status codes back into the application again. This will result in the `error` controller in the Pylons project being called. This is why there is a default route in `config/routing.py` of:

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

The error controller allows a project to theme the error message appropriately by changing it to render a template, or redirect as desired.

15.4.1 Original Request Information

The original request and response that resulted in the error controller being called is available inside the error controller as:

```
# Original request
request.environ['pylons.original_request']

# Original response
request.environ['pylons.original_response']
```

If an `HTTPException` was thrown in the controller (the `abort()` function throws these), the original object is available as:

```
request.environ['pylons.controller.exception']
```

This allows access to the error message on the exception object.

16.1 1.0 -> 1.0.1

No changes are necessary, however to take advantage of MarkupSafe's faster HTML escaping, the default filter in `environment.py` that Mako is configured with should be changed from:

```
from webhelpers.html import escape
```

To:: `from markupsafe import escape`

MarkupSafe utilizes a C extension where available for faster escaping which can help on larger pages with substantial variable substitutions.

16.2 0.9.7 -> 1.0

Upgrading your project is slightly different depending on which versions you're upgrading from and to. It's recommended that upgrades be done in minor revision steps, as deprecation warnings are added between revisions to help in the upgrade process.

For any project prior to 0.9.7, you should first follow the applicable docs to upgrade to 0.9.7 before proceeding.

To upgrade to 1.0, first upgrade your project to 0.10. This is a Pylons release that is fully backwards-compatible with 0.9.7. However under 0.10 a variety of warnings will be issued about the various things that need to be changed before upgrading to 1.0.

Tip: Since Pylons 0.10 is only out as a beta at this point, upgrade using the actual URL, for example:

```
$ easy_install -U http://pylonshq.com/download/0.10/Pylons-0.10.tar.gz
```

Beyond the warnings issued, you should also read the following list and ensure these changes have been applied.

Pylons changes from 0.9.7 to 1.0:

- The config object created in `environment.py` is now passed around explicitly. There are also some other minor updates as follows.

Update `config/environment.py` to initialize and return the config:

```
# Add to the imports:
from pylons.configuration import PylonsConfig

# Add under 'def load_environment':
config = PylonsConfig()

# Replace the make_map / app globals line with
config['routes.map'] = make_map(config)
config['pylons.app_globals'] = app_globals.Globals(config)

# Optionally, if removing the CacheMiddleware and using the
# cache in the new 1.0 style, add under the previous lines:
import pylons
pylons.cache._push_object(config['pylons.app_globals'].cache)

# Add at the end of the load_environment function:
return config
```

Update `config/middleware.py` to use the returned *config*:

```
# modify the load_environment call:
config = load_environment(global_conf, app_conf)

# update the middleware calls

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

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

# CUSTOM MIDDLEWARE HERE (filtered by error handling middlewares)

# Add right before 'return app':
app.config = config
```

Note: The `CacheMiddleware` is no longer setup by default through middleware, its now setup under `app_globals` inside its instantiation in `lib/app_globals.py`.

Update `config/routing.py` to accept the *config*:

```
# Replace the def line with
def make_map(config):
```

Update `lib/app_globals.py` to accept the *config*:

```
# Replace the __init__ line with
def __init__(self, config):

# Optionally, if you decided to remove the CacheMiddleware
```

```
# Add these imports
from beaker.cache import CacheManager
from beaker.util import parse_cache_config_options

# and add this line in __init__:
self.cache = CacheManager(**parse_cache_config_options(config))
```

Update tests/__init__.py as needed:

```
from unittest import TestCase

from paste.deploy import loadapp
from paste.script.appinstall import SetupCommand
from pylons import url
from routes.util import URLGenerator
from webtest import TestApp

import pylons.test

__all__ = ['environ', 'url', 'TestController']

# Invoke websetup with the current config file
SetupCommand('setup-app').run([pylons.test.pylonsapp.config['__file__']])

environ = {}

class TestController(TestCase):

    def __init__(self, *args, **kwargs):
        wsgiapp = pylons.test.pylonsapp
        config = wsgiapp.config
        self.app = TestApp(wsgiapp)
        url._push_object(URLGenerator(config['routes.map'], environ))
        TestCase.__init__(self, *args, **kwargs)
```

Note: Change the use of `url_for` in your tests to use `url`, which is imported from `tests/__init__.py` in your unit tests.

Finally, update `websetup.py` to avoid the duplicate app creation that previously could occur during the unit tests:

```
# Add to the imports
import pylons.test

# Add under the 'def setup_app':

# Don't reload the app if it was loaded under the testing environment
if not pylons.test.pylonsapp:
    load_environment(conf.global_conf, conf.local_conf)
```

- Change all instances of `redirect_to(...)` -> `redirect(url(...))`
`redirect_to` processed arguments in a slightly ‘magical’ manner in that some of them went to the `url_for` while sometimes... not. `redirect()` issues a redirect and nothing more, so to generate a url, the `url` instance should be used (import: `from pylons import url`).
- Ensure that all use of `g` is switched to using the new name, *app_globals*

- Change all instances of `url_for` to `url`.

Note that `url` does not retain the current route memory like `url_for` did by default. To get a route generated using the current route, call `url.current`.

For example:

```
# Rather than url_for() for the current route
url.current()
```

`url` can be imported from `pylons`.

- Change `config` import statement if needed

Previously, the `config` object could be imported as if it was a module:

```
import pylons.config
```

The `config` object is now an object in `pylons/__init__.py` so the import needs to be changed to:

```
from pylons import config
```

- Routes is now explicit by default

This won't affect those already using `url` as it ignores route memory. This change does mean that some routes which relied on a default controller of 'content' and a default action of 'index' will not work.

To restore the old behavior, in `config/routing.py`, set the mapper to explicit:

```
map.explicit = True
```

- By default, the `tmpl_context` (a.k.a 'c'), is no longer a `AttribSafeContextObj`. This means accessing attributes that don't exist will raise an `AttributeError`.

To use the attribute-safe `tmpl_context`, add this line to the `config/environment.py`:

```
config['pylons.strict_tmpl_context'] = False
```

Packaging and Deployment Overview

TODO: some of this is redundant to the (more current) *Configuration* doc – should be consolidated and cross-referenced

This document describes how a developer can take advantage of Pylons' application setup functionality to allow webmasters to easily set up their application.

Installation refers to the process of downloading and installing the application with *easy_install* whereas setup refers to the process of setting up an instance of an installed application so it is ready to be deployed.

For example, a wiki application might need to create database tables to use. The webmaster would only install the wiki `.egg` file once using *easy_install* but might want to run 5 wikis on the site so would setup the wiki 5 times, each time specifying a different database to use so that 5 wikis can run from the same code, but store their data in different databases.

17.1 Egg Files

Before you can understand how a user configures an application you have to understand how Pylons applications are distributed. All Pylons applications are distributed in `.egg` format. An egg is simply a Python executable package that has been put together into a single file.

You create an egg from your project by going into the project root directory and running the command:

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

If everything goes smoothly a `.egg` file with the correct name and version number appears in a newly created `dist` directory.

When a webmaster wants to install a Pylons application he will do so by downloading the egg and then installing it.

17.2 Installing as a Non-root User

It's quite possible when using shared hosting accounts that you do not have root access to install packages. In this case you can install *setuptools* based packages like Pylons and Pylons web applications in your home directory using a *virtualenv* setup. This way you can install all the packages you want to use without super-user access.

17.3 Understanding the Setup Process

Say you have written a Pylons wiki application called `wiki`. When a webmaster wants to install your wiki application he will run the following command to generate a config file:

```
$ paster make-config wiki wiki_production.ini
```

He will then edit the config file for his production environment with the settings he wants and then run this command to setup the application:

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

Finally he might choose to deploy the wiki application through the paste server like this (although he could have chosen CGI/FastCGI/SCGI etc):

```
$ paster serve wiki_production.ini
```

The idea is that an application only needs to be installed once but if necessary can be set up multiple times, each with a different configuration.

All Pylons applications are installed in the same way, so you as the developer need to know how the above commands work.

17.3.1 Make Config

The `paster make-config` command looks for the file `deployment.ini_tmpl` and uses it as a basis for generating a new `.ini` file.

Using our new wiki example again, the `wiki/config/deployment.ini_tmpl` file contains the text:

```
[DEFAULT]
debug = true
email_to = you@yourdomain.com
smtp_server = localhost
error_email_from = paste@localhost

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000

[app:main]
use = egg:wiki
full_stack = true
static_files = true
cache_dir = %(here)s/data
beaker.session.key = wiki
beaker.session.secret = ${app_instance_secret}
```

```

app_instance_uuid = ${app_instance_uuid}

# If you'd like to fine-tune the individual locations of the cache data dirs
# for the Cache data, or the Session saves, un-comment the desired settings
# here:
#beaker.cache.data_dir = %(here)s/data/cache
#beaker.session.data_dir = %(here)s/data/sessions

# WARNING: *THE LINE BELOW MUST BE UNCOMMENTED ON A PRODUCTION ENVIRONMENT*
# Debug mode will enable the interactive debugging tool, allowing ANYONE to
# execute malicious code after an exception is raised.
set debug = false

# Logging configuration
[loggers]
keys = root

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

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

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

```

When the command `paster make-config wiki wiki_production.ini` is run, the contents of this file are produced so you should tweak this file to provide sensible default configuration for production deployment of your app.

17.3.2 Setup App

The `paster setup-app` command references the newly created `.ini` file and calls the function `wiki.websetup.setup_app()` to set up the application. If your application needs to be set up before it can be used, you should edit the `websetup.py` file.

Here's an example which just prints the location of the cache directory via Python's logging facilities:

```

"""Setup the helloworld application"""
import logging

from pylons import config
from helloworld.config.environment import load_environment

log = logging.getLogger(__name__)

```

```
def setup_app(command, conf, vars):
    """Place any commands to setup helloworld here"""
    load_environment(conf.global_conf, conf.local_conf)
    log.info("Using cache directory %s" % config['cache.dir'])
```

For a more useful example, say your application needs a database set up and loaded with initial data. The user will specify the location of the database to use by editing the config file before running the `paster setup-app` command. The `setup_app()` function will then be able to load the configuration and act on it in the function body. This way, the `setup_app()` function can be used to initialize the database when `paster setup-app` is run. Using the optional *SQLAlchemy* project template support when creating a Pylons project will set all of this up for you in a basic way. The `quickwiki_tutorial` illustrates an example of this configuration.

17.4 Deploying the Application

Once the application is setup it is ready to be deployed. There are lots of ways of deploying an application, one of which is to use the `paster serve` command which takes the configuration file that has already been used to setup the application and serves it on a local HTTP server for production use:

```
$ paster serve wiki_production.ini
```

More information on Paste deployment options is available on the Paste website at <http://pythonpaste.org>. See *Running Pylons Apps with Other Web Servers* for alternative Pylons deployment scenarios.

17.5 Advanced Usage

So far everything we have done has happened through the `paste.script.appinstall.Installer` class which looks for the `deployment.ini_tmpl` and `websetup.py` file and behaves accordingly.

If you need more control over how your application is installed you can use your own installer class. Create a file, for example `wiki/installer.py` and code your new installer class in the file by deriving it from the existing one:

```
from paste.script.appinstall import Installer
class MyInstaller(Installer):
    pass
```

You then override the functionality as necessary (have a look at the source code for `Installer` as a basis. You then change your application's `setup.py` file so that the `paste.app_install` entry point `main` points to your new installer:

```
entry_points="""
...
[paste.app_install]
main=wiki.installer:MyInstaller
...
""",
```

Depending on how you code your `MyInstaller` class you may not even need your `websetup.py` or `deployment.ini_tmpl` as you might have decided to create the `.ini` file and setup the application in an entirely different way.

Running Pylons Apps with Other Web Servers

This document assumes that you have already installed a Pylons web application, and *Runtime Configuration* for it. Pylons applications use *PasteDeploy* to start up your Pylons WSGI application, and can use the *flup* package to provide a Fast-CGI, SCGI, or AJP connection to it.

18.1 Using Fast-CGI

Fast-CGI is a gateway to connect web servers like *Apache* and *lighttpd* to a CGI-style application. Out of the box, Pylons applications can run with Fast-CGI in either a threaded or forking mode. (Threaded is the recommended choice)

Setting a Pylons application to use Fast-CGI is very easy, and merely requires you to change the config line like so:

```
# default
[server:main]
use = egg:Paste#http

# Use Fastcgi threaded
[server:main]
use = egg:PasteScript#flup_fcgi_thread
host = 0.0.0.0
port = 6500
```

Note that you will need to install the *flup* package, which can be installed via *easy_install*:

```
$ easy_install -U flup
```

The options in the config file are passed onto *flup*. The two common ways to run Fast CGI is either using a socket to listen for requests, or listening on a port/host which allows a webserver to send your requests to web applications on a different machine.

To configure for a socket, your `server:main` section should look like this:

```
[server:main]
use = egg:PasteScript#flup_fcgi_thread
socket = /location/to/app.socket
```

If you want to listen on a host/port, the configuration cited in the first example will do the trick.

18.2 Apache Configuration

For this example, we will assume you're using Apache 2, though Apache 1 configuration will be very similar. First, make sure that you have the Apache `mod_fastcgi` module installed in your Apache.

There will most likely be a section where you declare your FastCGI servers, and whether they're external:

```
<IfModule mod_fastcgi.c>
FastCgiIpcDir /tmp
FastCgiExternalServer /some/path/to/app/myapp.fcgi -host some.host.com:6200
</IfModule>
```

In our example we'll assume you're going to run a Pylons web application listening on a host/port. Changing `-host` to `-socket` will let you use a Pylons web application listening on a socket.

The filename you give in the second option does not need to physically exist on the webserver, URIs that Apache resolve to this filename will be handled by the FastCGI application.

The other important line to ensure that your Apache webserver has is to indicate that fcgi scripts should be handled with Fast-CGI:

```
AddHandler fastcgi-script .fcgi
```

Finally, to configure your website to use the Fast CGI application you will need to indicate the script to be used:

```
<VirtualHost *:80>
    ServerAdmin george@monkey.com
    ServerName monkey.com
    ServerAlias www.monkey.com
    DocumentRoot /some/path/to/app

    ScriptAliasMatch ^(/.*)$ /some/path/to/app/myapp.fcgi$1
</VirtualHost>
```

Other useful directives should be added as needed, for example, the `ErrorLog` directive, etc. This configuration will result in all requests being sent to your FastCGI application.

18.3 PrefixMiddleware

`PrefixMiddleware` provides a way to manually override the root prefix (`SCRIPT_NAME`) of your application for certain situations.

When running an application under a prefix (such as `/james`) in FastCGI/apache, the `SCRIPT_NAME` environment variable is automatically set to the appropriate value: `/james`. Pylons' URL generators such as `url` always take the `SCRIPT_NAME` value into account.

One situation where `PrefixMiddleware` is required is when an application is accessed via a reverse proxy with a prefix. The application is accessed through the reverse proxy via the URL prefix `/james`, whereas the reverse proxy forwards those requests to the application at the prefix `/`.

The reverse proxy, being an entirely separate web server, has no way of specifying the `SCRIPT_NAME` variable; it must be manually set by a `PrefixMiddleware` instance. Without setting `SCRIPT_NAME`, url will generate URLs such as: `‘/purchase_orders/1’`, when it should be generating: `‘/james/purchase_orders/1’`.

To filter your application through a `PrefixMiddleware` instance, add the following to the `‘[app:main]’` section of your `.ini` file:

```
filter-with = proxy-prefix

[filter:proxy-prefix]
use = egg:PasteDeploy#prefix
prefix = /james
```

The name `proxy-prefix` simply acts as an identifier of the filter section; feel free to rename it.

These `.ini` settings are equivalent to adding the following to the end of your application’s `config/middleware.py`, right before the `return app` line:

```
# This app is served behind a proxy via the following prefix (SCRIPT_NAME)
app = PrefixMiddleware(app, global_conf, prefix='/james')
```

This requires the additional import line:

```
from paste.deploy.config import PrefixMiddleware
```

Whereas the modification to `config/middleware.py` will setup an instance of `PrefixMiddleware` under every environment (`.ini`).

18.4 Using Java Web Servers with Jython

See *Deploying to Java Web servers*.

Documenting Your Application

TODO: this needs to be rewritten – Pudge is effectively dead

While the information in this document should be correct, it may not be entirely complete... Pudge is somewhat unruly to work with at this time, and you may need to experiment to find a working combination of package versions. In particular, it has been noted that an older version of Kid, like 0.9.1, may be required. You might also need to install `{{RuleDispatch}}` if you get errors related to `{{FormEncode}}` when attempting to build documentation.

Apologies for this suboptimal situation. Considerations are being taken to fix Pudge or supplant it for future versions of Pylons.

19.1 Introduction

Pylons comes with support for automatic documentation generation tools like [Pudge](#).

Automatic documentation generation allows you to write your main documentation in the docs directory of your project as well as throughout the code itself using docstrings.

When you run a simple command all the documentation is built into sophisticated HTML.

19.2 Tutorial

First create a project as described in [Getting Started](#).

You will notice a docs directory within your main project directory. This is where you should write your main documentation.

There is already an `index.txt` file in docs so you can already generate documentation. First we'll install Pudge and buildutils. By default, Pylons sets an option to use [Pygments](#) for syntax-highlighting of code in your documentation, so you'll need to install it too (unless you wish to remove the option from `setup.cfg`):

```
$ easy_install pudge buildutils
$ easy_install Pygments
```

then run the following command from your project's main directory where the `setup.py` file is:

```
$ python setup.py pudge
```

Note: The `pudge` command is currently disabled by default. Run the following command first to enable it:

```
..code-block:: bash
```

```
$ python setup.py addcommand -p buildutils.pudge_command
```

Thanks to Yannick Gingras for the tip.

Pudge will produce output similar to the following to tell you what it is doing and show you any problems:

```
running pudge
generating documentation
copying: pudge\template\pythonpaste.org\rst.css -> do/docs/html\rst.css
copying: pudge\template\base\pudge.css -> do/docs/html\pudge.css
copying: pudge\template\pythonpaste.org\layout.css -> do/docs/html\layout.css
rendering: pudge\template\pythonpaste.org\site.css.kid -> site.css
colorizing: do/docs/html\do\__init__.py.html
colorizing: do/docs/html\do\tests\__init__.py.html
colorizing: do/docs/html\do\i18n\__init__.py.html
colorizing: do/docs/html\do\lib\__init__.py.html
colorizing: do/docs/html\do\controllers\__init__.py.html
colorizing: do/docs/html\do\model.py.html
```

Once finished you will notice a `docs/html` directory. The `index.html` is the main file which was generated from `docs/index.txt`.

19.3 Learning ReStructuredText

Python programs typically use a rather odd format for documentation called [reStructuredText](#). It is designed so that the text file used to generate the HTML is as readable as possible but as a result can be a bit confusing for beginners.

Read the [reStructuredText](#) tutorial which is part of the [docutils](#) project.

Once you have mastered [reStructuredText](#) you can write documentation until your heart's content.

19.4 Using Docstrings

Docstrings are one of Python's most useful features if used properly. They are described in detail in the [Python](#) documentation but basically allow you to document any module, class, method or function, in fact just about anything. Users can then access this documentation interactively.

Try this:

```
>>> import pylons
>>> help(pylons)
...
```

As you can see if you tried it you get detailed information about the `pylons` module including the information in the docstring.

Docstrings are also extracted by Pudge so you can describe how to use all the controllers, actions and modules that make up your application. Pudge will extract that information and turn it into useful API documentation automatically.

Try clicking the `Modules` link in the HTML documentation you generated earlier or look at the Pylons source code for some examples of how to use docstrings.

19.5 Using doctest

The final useful thing about docstrings is that you can use the `doctest` module with them. `doctest` again is described in the Python documentation but it looks through your docstrings for things that look like Python code written at a Python prompt. Consider this example:

```
>>> a = 2
>>> b = 3
>>> a + b
5
```

If `doctest` was run on this file it would have found the example above and executed it. If when the expression `a + b` is executed the result was not 5, `doctest` would raise an `Exception`.

This is a very handy way of checking that the examples in your documentation are actually correct.

To run `doctest` on a module use:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

The `if __name__ == "__main__":` part ensures that your module won't be tested if it is just imported, only if it is run from the command line

To run `doctest` on a file use:

```
import doctest
doctest.testfile("docs/index.txt")
```

You might consider incorporating this functionality in your `tests/test.py` file to improve the testing of your application.

19.6 Summary

So if you write your documentation in reStructuredText, in the `docs` directory and in your code's docstrings, liberally scattered with example code, Pylons provides a very useful and powerful system for you.

If you want to find out more information have a look at the Pudge documentation or try tinkering with your project's `setup.cfg` file which contains the Pudge settings.

CHAPTER 20

Distributing Your Application

TODO: this assumes helloworld tutorial context that is no longer present, and could be consolidated with packaging info in *Packaging and Deployment Overview*

As mentioned earlier eggs are a convenient format for packaging applications. You can create an egg for your project like this:

```
$ cd helloworld
$ python setup.py bdist_egg
```

Your egg will be in the `dist` directory and will be called `helloworld-0.0.0dev-py2.4.egg`.

You can change options in `setup.py` to change information about your project. For example change version to `version="0.1.0"`, and run `python setup.py bdist_egg` again to produce a new egg with an updated version number.

You can then register your application with the [Python Package Index \(PyPI\)](#) with the following command:

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

Note: You should not do this unless you actually want to register a package!

If users want to install your software and have installed *easy_install* they can install your new egg as follows:

```
$ easy_install helloworld==0.1.0
```

This will retrieve the package from PyPI and install it. Alternatively you can install the egg locally:

```
$ easy_install -f C:\path\with\the\egg\files\in helloworld==0.1.0
```

In order to use the egg in a website you need to use Paste. You have already used Paste to create your Pylons template and to run a test server to test the tutorial application.

Paste is a set of tools available at <http://pythonpaste.org> for providing a uniform way in which all compatible Python web frameworks can work together. To run a paste application such as any Pylons application you need to create

a Paste configuration file. The idea is that the your paste configuration file will contain all the configuration for all the different Paste applications you run. A configuration file suitable for development is in the `helloworld/development.ini` file of the tutorial but the idea is that the person using your egg will add relevant configuration options to their own Paste configuration file so that your egg behaves they way they want. See the section below for more on this configuration.

Paste configuration files can be run in many different ways, from CGI scripts, as standalone servers, with FastCGI, SCGI, `mod_python` and more. This flexibility means that your Pylons application can be run in virtually any environment and also take advantage of the speed benefits that the deployment option offers.

See also:

[Running Pylons Apps with Other Web Servers](#)

20.1 Running Your Application

In order to run your application your users will need to install it as described above but then generate a config file and setup your application before deploying it. This is described in *[Runtime Configuration](#)* and *[Packaging and Deployment Overview](#)*.

Python 2.3 Installation Instructions

21.1 Advice of end of support for Python 2.3

Warning: END OF SUPPORT FOR PYTHON 2.3 This is the **LAST** version to support Python 2.3 **BEGIN UPGRADING OR DIE**

21.2 Preparation

First, please note that Python 2.3 users on Windows will need to install [subprocess.exe](#) before beginning the installation (whereas Python 2.4 users on Windows do not). All windows users also should read the section *Windows Notes* after installation. Users of Ubuntu/debian will also likely need to install the python-dev package.

21.3 System-wide Install

To install Pylons so it can be used by everyone (you'll need root access).

If you already have easy install:

```
$ easy_install Pylons==0.9.7
```

Note: On rare occasions, the python.org Cheeseshop goes down. It is still possible to install Pylons and its dependencies however by specifying our local package directory for installation with:

```
$ easy_install -f http://pylonshq.com/download/ Pylons==0.9.7
```

Which will use the packages necessary for the latest release. If you're using an older version of Pylons, you can get the packages that went with it by specifying the version desired:

```
$ easy_install -f http://pylonshq.com/download/0.9.7/ Pylons==0.9.7
```

Otherwise:

1. Download the easy install setup file from http://peak.telecommunity.com/dist/ez_setup.py
2. Run:

```
$ python ez_setup.py Pylons==0.9.7
```

Warning: END OF SUPPORT FOR PYTHON 2.3 This is the **LAST** version to support Python 2.3 **BEGIN UPGRADING OR DIE**

CHAPTER 22

Windows Notes

Python scripts installed as part of the Pylons install process will be put in the `Scripts` directory of your Python installation, typically in `C:\Python24\Scripts`. By default on Windows, this directory is not in your `PATH`; this can cause the following error message when running a command such as `paster` from the command prompt:

```
C:\Documents and Settings\James>paster
'paster' is not recognized as an internal or external command,
operable program or batch file.
```

To run the scripts installed with Pylons either the full path must be specified:

```
C:\Documents and Settings\James>C:\Python24\Scripts\paster
Usage: C:\Python24\Scripts\paster-script.py COMMAND
usage: paster-script.py [paster_options] COMMAND [command_options]

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

or (the preferable solution) the `Scripts` directory must be added to the `PATH` as described below.

22.1 For Win2K or WinXP

1. From the desktop or Start Menu, right click My Computer and click Properties.
2. In the System Properties window, click on the Advanced tab.
3. In the Advanced section, click the Environment Variables button.

4. Finally, in the Environment Variables window, highlight the path variable in the Systems Variable section and click edit. Add or modify the path lines with the paths you wish the computer to access. Each different directory is separated with a semicolon as shown below:

```
C:\Program Files;C:\WINDOWS;C:\WINDOWS\System32
```

1. Add the path to your scripts directory:

```
C:\Program Files;C:\WINDOWS;C:\WINDOWS\System32;C:\Python24\Scripts
```

See *Finally* below.

22.2 For Windows 95, 98 and ME

Edit `autoexec.bat`, and add the following line to the end of the file:

```
set PATH=%PATH%;C:\Python24\Scripts
```

See *Finally* below.

22.3 Finally

Restarting your computer may be required to enable the change to the `PATH`. Then commands may be entered from any location:

```
C:\Documents and Settings\James>paster
Usage: C:\Python24\Scripts\paster-script.py COMMAND
usage: paster-script.py [paster_options] COMMAND [command_options]

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

All documentation assumes the `PATH` is setup correctly as described above.

Pylons supports [Jython](#) as of v0.9.7.

23.1 Installation

The installation process is the same as CPython, as described in [Getting Started](#). At least Jython 2.5b2 is required.

23.2 Deploying to Java Web servers

The Java platform defines the [Servlet API](#) for creating web applications. The [modjy](#) library included with Jython provides a gateway between Java Servlets and WSGI applications.

The [snakefight](#) tool can create a [WAR file](#) from a Pylons application (and modjy) that's suitable for deployment to the various [Servlet containers](#) (such as [Apache Tomcat](#) or [Sun's Glassfish](#)).

23.2.1 Creating .wars with snakefight

First, install snakefight:

```
$ easy_install snakefight
```

This adds an additional command to distutils: **bdist_war**.

Pylons applications are loaded from Paste, via its `paste.app_factory` entry point and a Paste style configuration file. **bdist_war** knows how to setup Paste apps for deployment when specified the `--paste-config` option:

```
$ paster make-config MyApp production.ini
$ jython setup.py bdist_war --paste-config production.ini
```

As with any distutils command the preferred options can instead be added to the `setup.cfg` in the root directory of the project:

```
[bdist_war]
paste-config = production.ini
```

Then we can simply run:

```
$ jython setup.py bdist_war
```

bdist_war creates a `.war` with the following:

- Jython's jar files in `WEB-INF/lib`
- Jython's stdlib in `WEB-INF/lib-python`
- Your application's required eggs in `WEB-INF/lib-python`

With the `--paste-config` option, it also:

- Creates a simple loader for the application/config
- Generates a `web.xml` deployment descriptor configuring modjy to load the application with the simple loader

For further information/usages, see [snakefight's documentation](#).

Security policy for bugs

24.1 Receiving Security Updates

The Pylons team have set up a mailing list at wsgi-security-announce@googlegroups.com to which any security vulnerabilities that affect Pylons will be announced. Anyone wishing to be notified of vulnerabilities in Pylons should subscribe to this list. Security announcements will only be made once a solution to the problem has been discovered.

24.2 Reporting Security Issues

Please report security issues by email to both the lead developers of Pylons at the following addresses:

ben  groovie.org

security  3aims.com

Please DO NOT announce the vulnerability to any mailing lists or on the ticket system because we would not want any malicious person to be aware of the problem before a solution is available.

In the event of a confirmed vulnerability in Pylons itself, we will take the following actions:

- Acknowledge to the reporter that we've received the report and that a fix is forthcoming. We'll give a rough timeline and ask the reporter to keep the issue confidential until we announce it.
- Halt all other development as long as is needed to develop a fix, including patches against the current release.
- Publicly announce the vulnerability and the fix as soon as it is available to the WSGI security list at wsgi-security-announce@googlegroups.com.

This will probably mean a new release of Pylons, but in some cases it may simply be the release of documentation explaining how to avoid the vulnerability.

In the event of a confirmed vulnerability in one of the components that Pylons uses, we will take the following actions:

- Acknowledge to the reporter that we've received the report and ask the reporter to keep the issue confidential until we announce it.

- Contact the developer or maintainer of the package containing the vulnerability.
- If the developer or maintainer fails to release a new version in a reasonable time-scale and the vulnerability is serious we will either create documentation explaining how to avoid the problem or as a last resort, create a patched version.
- Publicly announce the vulnerability and the fix as soon as it is available to the WSGI security list at wsgi-security-announce@googlegroups.com.

24.3 Minimising Risk

- Only use official production versions of Pylons released publicly on the [Python Package Index](#).
- Only use stable releases of third party software not development, alpha, beta or release candidate code.
- Do not assume that related software is of the same quality as Pylons itself, even if Pylons users frequently make use of it.
- Subscribe to the wsgi-security-announce@googlegroups.com mailing list to be informed of security issues and their solutions.

The Web Server Gateway Interface defined in [PEP 333](#) is a standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers.

Pylons supports the Web Server Gateway Interface (or WSGI for short, pronounced “wizgy”) throughout its stack. This is important for developers because it means that as well coming with all the features you would expect of a modern web framework, Pylons is also extremely flexible. With the WSGI it is possible to change any part of the Pylons stack to add new functionality or modify a request or a response without having to take apart the whole framework.

25.1 Paste and WSGI

Most of Pylons’ WSGI capability comes from its close integration with Paste. Paste provides all the tools and middleware necessary to deploy WSGI applications. It can be thought of as a low-level WSGI framework designed for other web frameworks to build upon. Pylons is an example of a framework which makes full use of the possibilities of Paste.

If you want to, you can get the WSGI application object from your Pylons configuration file like this:

```
from paste.deploy import loadapp
wsgi_app = loadapp('config:/path/to/config.ini')
```

You can then serve the file using a WSGI server. Here is an example using the WSGI Reference Implementation included with Python 2.5:

```
from paste.deploy import loadapp
wsgi_app = loadapp('config:/path/to/config.ini')

from wsgiref import simple_server
httpd = simple_server.WSGIServer('', 8000), simple_server.WSGIRequestHandler)
httpd.set_app(wsgi_app)
httpd.serve_forever()
```

The `paste serve` command you will be using during the development of Pylons projects combines these two steps of creating a WSGI app from the config file and serving the resulting file to give the illusion that it is serving the config file directly.

Because the resulting Pylons application is a WSGI application it means you can do the same things with it that you can do with any WSGI application. For example add a middleware chain to it or serve it via FastCGI/SCGI/CGI/mod_python/AJP or standalone.

You can also configure extra WSGI middleware, applications and more directly using the configuration file. The various options are described in the [Paste Deploy Documentation](#) so we won't repeat them here.

25.2 Using a WSGI Application as a Pylons 0.9 Controller

In Pylons 0.9 controllers are derived from `pylons.controllers.WSGIController` and are also valid WSGI applications. Unless your controller is derived from the legacy `pylons.controllers.Controller` class it is also assumed to be a WSGI application. This means that you don't actually need to use a Pylons controller class in your controller, any WSGI application will work as long as you give it the same name.

For example, if you added a `hello` controller by executing `paste controller hello`, you could modify it to look like this:

```
def HelloController(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ['Hello World!']
```

or use `yield` statements like this:

```
def HelloController(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    yield 'Hello '
    yield 'World!'
```

or use the standard Pylons `Response` object which is a valid WSGI response which takes care of calling `start_response()` for you:

```
def HelloController(environ, start_response):
    return Response('Hello World!')
```

and you could use the `render()` and `render_response()` objects exactly like you would in a normal controller action.

As well as writing your WSGI application as a function you could write it as a class:

```
class HelloController:

    def __call__(self, environ, start_response):
        start_response('200 OK', [('Content-Type', 'text/html')])
        return ['Hello World!']
```

All the standard Pylons middleware defined in `config/middleware.py` is still available.

25.3 Running a WSGI Application From Within a Controller

There may be occasions where you don't want to replace your entire controller with a WSGI application but simply want to run a WSGI application from within a controller action. If your project was called `test` and you had a WSGI

application called `wsgi_app` you could even do this:

```
from test.lib.base import *

def wsgi_app(environ, start_response):
    start_response('200 OK', [('Content-type', 'text/html')])
    return ['<html>\n<body>\nHello World!\n</body>\n</html>']

class HelloController(BaseController):
    def index(self):
        return wsgi_app(request.environ, self.start_response)
```

25.4 Configuring Middleware Within a Pylons Application

A Pylons application middleware stack is directly exposed in the project's `config/middleware.py` file. This means that you can add and remove pieces from the stack as you choose.

Warning: If you remove any of the default middleware you are likely to find that various parts of Pylons stop working!

As an example, if you wanted to add middleware that added a new key to the `environ` dictionary you might do this:

```
# YOUR MIDDLEWARE
# Put your own middleware here, so that any problems are caught by the error
# handling middleware underneath

class KeyAdder:
    def __init__(self, app, key, value):
        self.app = app
        if '.' not in key:
            raise Exception("WSGI environ keys must contain a '.' character")
        self.key = key
        self.value = value

    def __call__(self, environ, start_response):
        environ[self.key] = self.value
        return self.app(environ, start_response)

app = KeyAdder(app, 'test.hello', 'Hello World')
```

Then in your controller you could write:

```
return Response(request.environ['test.hello'])
```

and you would see your `Hello World!` message.

Of course, this isn't a particularly useful thing to do. Middleware classes can do one of four things or a combination of them:

- Change the `environ` dictionary
- Change the status
- Change the HTTP headers
- Change the response body of the application

With the ability to do these things as a middleware you can create authentication code, error handling middleware and more but the great thing about WSGI is that someone probably already has so you can consult the [wsgi.org middleware list](http://wsgi.org/middleware-list) or have a look at the [Paste project](#) and reuse an existing piece of middleware.

25.5 The Cascade

Towards the end of the middleware stack in your project's `config/middleware.py` file you will find a special piece of middleware called the cascade:

```
app = Cascade([static_app, javascripts_app, app])
```

Passed a list of applications, `Cascade` will try each of them in turn. If one returns a 404 status code then the next application is tried until one of the applications returns a code other than 404 in which case its response is returned. If all applications fail, then the last application's failure response is used.

The three WSGI applications in the cascade serve files from your project's `public` directory first then if nothing matches, the `WebHelpers` module `JavaScripts` are searched and finally if no `JavaScripts` are found your Pylons app is tried. This is why the `public/index.html` file is served before your controller is executed and why you can put `/javascripts/` into your HTML and the files will be found.

You are free to change the order of the cascade or add extra WSGI applications to it before `app` so that other locations are checked before your Pylons application is executed.

25.6 Useful Resources

Whilst other frameworks have put WSGI adapters at the end of their stacks so that their applications can be served by WSGI servers, we hope you can see how fully Pylons embraces WSGI throughout its design to be the most flexible and extensible of the main Python web frameworks.

To find out more about the Web Server Gateway Interface you might find the following resources useful:

- [PEP 333](#)
- [The WSGI website at wsgi.org](http://wsgi.org)
- XML.com articles: [Introducing WSGI - Python's Secret Web Weapon.html](#) [Part 1](#) [Part 2](#)

26.1 WSGI, CLI scripts

26.1.1 Working with `wsgiwrappers.WSGIRequest`

Pylons uses a specialised *WSGIRequest* class that is accessible via the `paste.wsgiwrappers` module.

The `wsgiwrappers.WSGIRequest` object represents a WSGI request that has a more programmer-friendly interface. This interface does not expose every detail of the WSGI environment (*why?*) and does not attempt to express anything beyond what is available in the environment dictionary.

The only state maintained in this object is the desired `charset`, an associated errors handler and a `decode_param_names` option.

Unicode notes

When `charset` is set, the incoming parameter values will be automatically coerced to unicode objects of the charset encoding.

When unicode is expected, `charset` will be overridden by the the value of the `charset` parameter set in the Content-Type header, if one was specified by the client.

The incoming parameter names are not decoded to unicode unless the `decode_param_names` option is enabled.

The class variable `defaults` specifies default values for `charset`, `errors`, and `language`. These default values can be overridden for the current request via the registry (*what's a registry?*).

The language default value is considered the fallback during i18n translations to ensure in odd cases that mixed languages don't occur should the language file contain the string but not another language in the accepted languages list. The language value only applies when getting a list of accepted languages from the HTTP Accept header.

This behavior is duplicated from Aquarium, and may seem strange but is very useful. Normally, everything in the code is in "en-us". However, the "en-us" translation catalog is usually empty. If the user requests ["en-us", "zh-cn"] and a translation isn't found for a string in "en-us", you don't want gettext to fallback to "zh-cn". You want it to just use the string itself. Hence, if a string isn't found in the language catalog, the string in the source code will be used.

All other state is kept in the environment dictionary; this is essential for interoperability.

You are free to subclass this object.

26.1.2 Attributes

GET

A dictionary-like object representing the `QUERY_STRING` parameters. Always present, possibly empty.

If the same key is present in the query string multiple times, a list of its values can be retrieved from the `MultiDict` via the `:meth:getall` method.

Returns a `MultiDict` container or, when `charset` is set, a `UnicodeMultiDict`.

POST

A dictionary-like object representing the `POST` body.

Most values are encoded strings, or unicode strings when `charset` is set. There may also be `FieldStorage` objects representing file uploads. If this is not a `POST` request, or the body is not encoded fields (e.g., an `XMLRPC` request) then this will be empty.

This will consume `wsgi.input` when first accessed if applicable, but the raw version will be put in `environ['paste.parsed_formvars']`.

Returns a `MultiDict` container or a `UnicodeMultiDict` when `charset` is set.

cookies

A dictionary of cookies, keyed by cookie name.

Just a plain dictionary, may be empty but not `None`.

defaults

```
{'errors': 'replace',
 'decode_param_names': False,
 'charset': None,
 'language': 'en-us'}
```

host

The host name, as provided in `HTTP_HOST` with a fall-back to `SERVER_NAME`

is_xhr

Returns a boolean if `X-Requested-With` is present and is a `XMLHttpRequest`

languages

Returns a (possibly empty) list of preferred languages, most preferred first.

params

A dictionary-like object of keys from POST, GET, URL dicts

Return a key value from the parameters, they are checked in the following order: POST, GET, URL

26.1.3 Additional methods supported:

getlist(key)

Returns a list of all the values by that key, collected from POST, GET, URL dicts

Returns a `MultiDict` container or a `UnicodeMultiDict` when `charset` is set.

urlvars

Return any variables matched in the URL (e.g. `wsgiorg.routing_args`).

26.1.4 Methods

`__init__(self, environ)`

`determine_browser_charset(self)`

Determine the encoding as specified by the browser via the Content-Type's `charset` parameter, if one is set

`match_accept(self, mimetypes)`

Return a list of specified mime-types that the browser's HTTP Accept header allows in the order provided.

26.2 Adding commands to Paster

26.2.1 Paster command

The command line will be `paster my-command arg1 arg2` if the current directory is the application egg, or `paster --plugin=MyPylonsApp my-command arg1 arg2` otherwise. In the latter case, `MyPylonsApp` must have been installed via `easy_install` or `python setup.py develop`.

Make a package directory for your commands:

```
$ mkdir myapp/commands
$ touch myapp/commands/__init__.py
```

Create a module `myapp/commands/my_command.py` like this:

```
from paste.script.command import Command

class MyCommand(Command):
    # Parser configuration
    summary = "--NO SUMMARY--"
    usage = "--NO USAGE--"
```

```
group_name = "myapp"
parser = Command.standard_parser(verbose=False)

def command(self):
    import pprint
    print "Hello, app script world!"
    print
    print "My options are:"
    print "    ", pprint.pformat(vars(self.options))
    print "My args are:"
    print "    ", pprint.pformat(self.args)
    print
    print "My parser help is:"
    print
    print self.parser.format_help()
```

Note: The class `_must_` define `.command`, `.parser`, and `.summary`

Modify the `entry_points` argument in `setup.py` to contain:

```
[paste.paster_command]
my-command = myapp.commands.my_command:MyCommand
```

Run `python setup.py develop` or `easy_install .` to update the entry points in the egg in `sys.path`.

Now you should be able to run:

```
$ paster --plugin=MyApp my-command arg1 arg2
Hello, MyApp script world!

My options are:
    {'interactive': False, 'overwrite': False, 'quiet': 0, 'verbose': 0}
My args are:
    ['arg1', 'arg2']

My parser help is:

Usage: /usr/local/bin/paster my-command [options] --NO USAGE--
--NO SUMMARY--

Options:
  -h, --help  show this help message and exit

$ paster --plugin=MyApp --help
Usage: paster [paster_options] COMMAND [command_options]

...
myapp:
  my-command      --NO SUMMARY--

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

26.2.2 Required class attributes

In addition to the `.command` method, the class should define `.parser` and `.summary`.

26.2.3 Command-line options

`Command.standard_parser()` returns a Python `OptionParser`. Calling `parser.add_option` enables the developer to add as many options as desired. Inside the `.command` method, the user's options are available under `self.options`, and any additional arguments are in `self.args`.

There are several other class attributes that affect the parser; see them defined in `paste.script.Command:Command`. The most useful attributes are `.usage`, `.description`, `.min_args`, and `.max_args`. `.usage` is the part of the usage string `_after_` the command name. The `.standard_parser()` method has several optional arguments to add standardized options; some of these got added to my parser although I don't see how.

See the `paster shell` command, `pylons.commands:ShellCommand`, for an example of using command-line options and loading the `.ini` file and model.

Also see “paster setup-app” where it is defined in `paste.script.appinstall.SetupCommand`. This is evident from the entry point in `PasteScript` (`PasteScript-VERSION.egg/EGG_INFO/entry_points.txt`). It is a complex example of reading a config file and delegating to another entry point.

The code for calling `myapp.websetup:setup_config` is in `paste.script.appinstall`.

The `Command` class also has several convenience methods to handle console prompts, enable logging, verify directories exist and that files have expected content, insert text into a file, run a shell command, add files to Subversion, parse “var=value” arguments, add variables to an `.ini` file.

26.2.4 Using paster to access a Pylons app

Paster provides `request` and `post` commands for running requests on an application. These commands will be run in the full configuration context of a normal application. Useful for cron jobs, the error handler will also be in place and you can get email reports of failed requests.

Because arguments all just go in `QUERY_STRING`, `request.GET` and `request.PARAMS` won't look like you expect. But you can parse them with something like:

```
parser = optparse.OptionParser()
parser.add_option(etc)

args = [item[0] for item in
        cgi.parse_qs1(request.environ['QUERY_STRING'])]

options, args = parser.parse_args(args)
```

paster request / post

Usage: `paster request / post [options] CONFIG_FILE URL [OPTIONS/ARGUMENTS]`

Run a request for the described application

This command makes an artificial request to a web application that uses a `paste.deploy` configuration file for the server and application. Use ‘paster request config.ini /url’ to request /url.

Use ‘paster post config.ini /url < data’ to do a POST with the given request body.

If the URL is relative (i.e. doesn't begin with /) it is interpreted as relative to `./command/`.

The variable `environ['paste.command_request']` will be set to `True` in the request, so your application can distinguish these calls from normal requests.

Note that you can pass options besides the options listed here; any unknown options will be passed to the application in `environ['QUERY_STRING']`.

```
Options:
-h, --help                show this help message and exit
-v, --verbose
-q, --quiet
-n NAME, --app-name=NAME
                        Load the named application (default main)
--config-var=NAME:VALUE
                        Variable to make available in the config for %()s
                        substitution (you can use this option multiple times)
--header=NAME:VALUE      Header to add to request (you can use this option
                        multiple times)
--display-headers        Display headers before the response body
```

Future development

A Pylons controller that handled some of this would probably be quite useful. Probably even nicer with additions to the current template, so that `/.command/` all gets routed to a single controller that uses actions for the various sub-commands, and can provide a useful response to `/.command/?-h`, etc.

26.3 Creating Paste templates

26.3.1 Introduction

[Python Paste](#) is an extremely powerful package that isn't just about WSGI middleware. The related document [Using Entry Points to Write Plugins](#) demonstrates how to use entry_points to create simple plugins. This document describes how to write just such a plugin for use Paste's project template creation facility and how to add a command to Paste's `paster` script.

The example task is to create a template for an imaginary content management system. The template is going to produce a project directory structure for a Python package, so we need to be able to specify a package name.

26.3.2 Creating The Directory Structure and Templates

The directory structure for the new project needs to look like this:

```
- default_project
  - +package+
    - __init__.py
    - static
      - layout
      - region
      - renderer
    - service
      - layout
        - __init__.py
      - region
        - __init__.py
```



```

- renderer
  - __init__.py
- setup.py_tmpl
- setup.cfg_tmpl
- development.ini_tmpl
- README.txt_tmpl
- ez_setup.py

```

Of course, the actual project's directory structure might look very different. In fact the `paster create` command can even be used to generate directory structures which *aren't* project templates — although this wasn't what it was designed for.

When the `paster create` command is run, any directories with `+package+` in their name will have that portion of the name replaced by a simplified package name and likewise any directories with `+egg+` in their name will have that portion replaced by the name of the egg directory, although we don't make use of that feature in this example.

All of the files with `_tmpl` at the end of their filenames are treated as templates and will have the variables they contain replaced automatically. All other files will remain unchanged.

Note: The small templating language used with `paster create` in files ending in `_tmpl` is described in detail in the [Paste util module documentation](#)

When specifying a package name it can include capitalisation and `_` characters but it should be borne in mind that the actual name of the package will be the *lowercase* package name with the `_` characters removed. If the package name contains an `_`, the egg name will contain a `_` character so occasionally the `+egg+` name is different to the `+package+` name.

To avoid difficulty always recommend to users that they stick with package names that contain no `_` characters so that the names remain unique when made lowercase.

26.3.3 Implementing the Code

Now that the directory structure has been defined, the next step is to implement the commands that will convert this to a ready-to-run project. The template creation commands are implemented by a class derived from `paste.script.templates.Template`. This is how our example appears:

```

from paste.script.templates import Template, var

vars = [
    var('version', 'Version (like 0.1)'),
    var('description', 'One-line description of the package'),
    var('long_description', 'Multi-line description (in reST)'),
    var('keywords', 'Space-separated keywords/tags'),
    var('author', 'Author name'),
    var('author_email', 'Author email'),
    var('url', 'URL of homepage'),
    var('license_name', 'License name'),
    var('zip_safe', 'True/False: if the package can be distributed as a .zip file',
        default=False),
]

class ArtProjectTemplate(Template):
    _template_dir = 'templates/default_project'
    summary = 'Art project template'
    vars = vars

```

The vars arguments can all be set at run time and will be available to be used as (in this instance) Cheetah template variables in the files which end `_tmpl`. For example the `setup.py_tmpl` file for the `default_project` might look like this:

```
from setuptools import setup, find_packages

version = ${repr(version)}|"0.0"}

setup(name=${repr(project)},
      version=version,
      description="${description|nothing}",
      long_description="""\
${long_description|nothing}""",
      classifiers=[],
      keywords=${repr(keywords)}|empty},
      author=${repr(author)}|empty},
      author_email=${repr(author_email)}|empty},
      url=${repr(url)}|empty},
      license=${repr(license_name)}|empty},
      packages=find_packages(exclude=['ez_setup']),
      include_package_data=True,
      zip_safe=${repr(bool(zip_safe))}|False},
      install_requires=[
          # Extra requirements go here #
      ],
      entry_points="""
          [paste.app_factory]
              main=${package}:make_app
      """,
  )
```

Note how the variables specified in vars earlier are used to generate the actual `setup.py` file.

In order to use the new templates they must be hooked up to the `paster create` command by means of an entry point. In the `setup.py` file of the project (in which created the project template is going to be stored) we need to add the following:

```
entry_points="""
    [paste.paster_create_template]
        art_project=art.entry.template:ArtProjectTemplate
    """,
```

We also need to add `PasteScript>=1.3` to the `install_requires` line.

```
install_requires=["PasteScript>=1.3"],
```

We just need to install the entry points now by running:

```
python setup.py develop
```

We should now be able to see a list of available templates with this command:

```
$ paster create --list-templates
```

Note: Windows users will need to add their Python scripts directory to their path or enter the full version of the command, similar to this:

```
C:\Python24\Scripts\paster.exe create --list-templates
```

You should see the following:

```
Available templates:
art_project:           Art project template
basic_package:         A basic setuptools-enabled package
```

There may be other projects too.

26.3.4 Troubleshooting

If the Art entries don't show up, check whether it is possible to import the `template.py` file because any errors are simply ignored by the `paster create` command rather than output as a warning.

If the code is correct, the issue might be that the entry points data hasn't been updated. Examine the `Python site-packages` directory and delete the `Art.egg-link` files, any `Art*.egg` files or directories and remove any entries for `art` from `easy_install.pth` (replacing `Art` with the name chosen for the project of course). Then re-run `python setup.py develop` to install the correct information.

If problems are still evident, then running the following code will print out a list of all entry points. It might help track the problem down:

```
import pkg_resources
for x in pkg_resources.iter_group_name(None, None):
    print x
```

26.3.5 Using the Template

Now that the entry point is working, a new project can be created:

```
$ paster create --template=art TestProject
```

Paster will ask lots of questions based on the variables set up in `vars` earlier. Pressing `return` will cause the default to be used. The final result is a nice project template ready for people to start coding with.

26.3.6 Implementing Pylons Templates

If the development context is subject to a frequent need to create lots of Pylons projects, each with a slightly different setup from the standard Pylons defaults then it is probably desirable to create a customised Pylons template to use when generating projects. This can be done in exactly the way described in this document.

First, set up a new Python package, perhaps called something like `CustomPylons` (obviously, don't use the Pylons name because Pylons itself is already using it). Then check out the Pylons source code and copy the `pylons/templates/default_project` directory into the new project as a starting point. The next stage is to add the custom `vars` and `Template` class and set up the entry points in the `CustomPylons setup.py` file.

After those tasks have been completed, it is then possible to create customised templates (ultimately based on the Pylons one) by using the `CustomPylons` package.

26.4 Using Entry Points to Write Plugins

26.4.1 Introduction

An entry point is a Python object in a project's code that is identified by a string in the project's `setup.py` file. The entry point is referenced by a group and a name so that the object may be discoverable. This means that another application can search for all the installed software that has an entry point with a particular group name, and then access the Python object associated with that name.

This is extremely useful because it means it is possible to write plugins for an appropriately-designed application that can be loaded at run time. This document describes just such an application.

It is important to understand that entry points are a feature of the new Python eggs package format and are *not* a standard feature of Python. To learn about eggs, their benefits, how to install them and how to set them up, see:

- [Python Eggs](#)
- [Easy Install](#)
- [Setuptools](#)

If reading the above documentation is inconvenient, suffice it to say that eggs are created via a similar `setup.py` file to the one used by Python's own `distutils` module — except that eggs have some powerful extra features such as entry points and the ability to specify module dependencies and have them automatically installed by `easy_install` when the application itself is installed.

For those developers unfamiliar with `distutils`: it is the standard mechanism by which Python packages should be distributed. To use it, add a `setup.py` file to the desired project, insert the required metadata and specify the important files. The `setup.py` file can be used to issue various commands which create distributions of the package in various formats for users to install.

26.4.2 Creating Plugins

This document describes how to use entry points to create a plugin mechanism which allows new types of content to be added to a content management system but we are going to start by looking at the plugin.

Say the standard way the CMS creates a plugin is with the `make_plugin()` function. In order for a plugin to be a plugin it must therefore have the function which takes the same arguments as the `make_plugin()` function and returns a plugin. We are going to add some image plugins to the CMS so we setup a project with the following directory structure:

```
+ image_plugins
+   __init__.py
+   setup.py
```

The `image_plugins/__init__.py` file looks like this:

```
def make_jpeg_image_plugin():
    return "This would return the JPEG image plugin"

def make_png_image_plugin():
    return "This would return the PNG image plugin"
```

We have now defined our plugins so we need to define our entry points. First let's write a basic `setup.py` for the project:

```

from setuptools import setup, find_packages

setup(
    name='ImagePlugins',
    version="1.0",
    description="Image plugins for the imaginary CMS 1.0 project",
    author="James Gardner",
    packages=find_packages(),
    include_package_data=True,
)

```

When using `setuptools` we can specify the `find_packages()` function and `include_package_data=True` rather than having to manually list all the modules and package data like we had to do in the old `distutils setup.py`.

Because the plugin is designed to work with the (imaginary) CMS 1.0 package, we need to specify that the plugin requires the CMS to be installed too and so we add this line to the `setup()` function:

```
install_requires=["CMS>=1.0"],
```

Now when the plugins are installed, CMS 1.0 or above will be installed automatically if it is not already present.

There are lots of other arguments such as `author_email` or `url` which you can add to the `setup.py` function too.

We are interested in adding the entry points. We need to decide on a group name for the entry points. It is traditional to use the name of the package using the entry point, separated by a `.` character and then use a name that describes what the entry point does. For our example `cms.plugin` might be an appropriate name for the entry point. Since the `image_plugin` module contains two plugins we will need two entries. Add the following to the `setup.py` function:

```

entry_points="""
    [cms.plugin]
    jpg_image=image_plugin:make_jpeg_image_plugin
    png_image=image_plugin:make_png_image_plugin
    """

```

Group names are specified in square brackets, plugin names are specified in the format `name=module.import.path:object_within_the_module`. The object doesn't have to be a function and can have any valid Python name. The module import path doesn't have to be a top level component as it is in this example and the name of the entry point doesn't have to be the same as the name of the object it is pointing to.

The developer can add as many entries as desired in each group as long as the names are different and the same holds for adding groups. It is also possible to specify the entry points as a Python dictionary rather than a string if that approach is preferred.

There are two more things we need to do to complete the plugin. The first is to include an `ez_setup` module so that if the user installing the plugin doesn't have `setuptools` installed, it will be installed for them. We do this by adding the following to the very top of the `setup.py` file before the import:

```

from ez_setup import use_setuptools
use_setuptools()

```

We also need to download the `ez_setup.py` file into our project directory at the same level as `setup.py`.

Note: If you keep your project in SVN there is a [trick you can use with the 'SVN:externals](#) to keep the `ez_setup.py` file up to date.

Finally in order for the CMS to find the plugins we need to install them. We can do this with:

```
$ python setup.py install
```

as usual or, since we might go on to develop the plugins further we can install them using a special development mode which sets up the paths to run the plugins from the source rather than installing them to Python's `site-packages` directory:

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

Both commands will download and install `setuptools` if you don't already have it installed.

26.4.3 Using Plugins

Now that the plugin is written we need to write the code in the CMS package to load it. Luckily this is even easier.

There are actually lots of ways of discovering plugins. For example: by distribution name and version requirement (such as `ImagePlugins>=1.0`) or by the entry point group and name (eg `jpg_image`). For this example we are choosing the latter, here is a simple script for loading the plugins:

```
from pkg_resources import iter_entry_points
for entry_point in iter_entry_points(group='cms.plugin', name=None):
    print(entry_point)

from pkg_resources import iter_entry_points
available_methods = []
for entry_point in iter_entry_points(group='authkit.method', name=None):
    available_methods.append(entry_point.load())
```

Executing this short script, will result in the following output:

```
This would return the JPEG image plugin
This would return the PNG image plugin
```

The `iter_entry_points()` function has looped though all the objects in the `cms.plugin` group and returned the function they were associated with. The application then called the function that the entry point was pointing to.

We hope that we have demonstrated the power of entry points for building extensible code and developers are encouraged to read the `pkg_resources` module documentation to learn about some more features of the eggs format.

Pylons Execution Analysis

By Mike Orr and Alfredo Deza

This chapter shows how Pylons calls your application, and how Pylons interacts with Paste, Routes, Mako, and its other dependencies. We'll create a simple application and then analyze the Python code executed starting from the moment we run the “paster serve” command.

Abbreviations: **\$APP** is your top-level application directory. **\$SP** is the site-packages directory where Pylons is installed. **\$BIN** is the location of `paster` and other executables. **\$SP** paths are shown in pip style (`$SP/pylons`) rather than `easy_install` style (`$SP/Pylons-VERSION.egg/pylons`).

27.1 The sample application

1. Create an application called “Analysis” with a controller called “main”:

```
$ paster create -t pylons Analysis
$ cd Analysis
$ paster controller main
```

Press Enter at all question prompts.

2. Edit `analysis/controllers/main.py` to look like this:

```
from analysis.lib.base import BaseController

class MainController(BaseController):

    def index(self):
        return '<h1>Welcome to the Analysis Demo</h1>Here is a <a href="/page2">
↩link</a>.'
```

```
    def page2(self):
        return 'Thank you for using the Analysis Demo. <a href="/">Home</a>'
```

There are two shortcuts here which you would not use in a normal application. One, we're returning incomplete HTML documents. Two, we've hardcoded the URLs to make the analysis easier to follow, rather than using the `url` object.

3. Now edit **analysis/config/routing.py**. Add these lines after "CUSTOM ROUTES HERE" (line 21):

```
map.connect("home", "/", controller="main", action="index")
map.connect("page2", "/page2", controller="main", action="page2")
```

4. Delete the file **analysis/public/index.html**.
5. Now run the server. (Press ctrl-C to quit it.)

```
$ paster serve development.ini
Starting server in PID 7341.
serving on http://127.0.0.1:5000
```

27.2 Pylons' dependencies

Pylons 1.0 has the following direct and indirect dependencies, which will be found in your site-packages directory (\$SP):

- Beaker 1.5.4
- decorator 3.2.0
- FormEncode 1.2.2
- Mako 0.3.4
- MarkupSafe 0.9.3
- Nose 0.11.4
- Paste 1.7.3.1
- PasteDeploy 1.3.3
- PasteScript 1.7.3
- Routes 1.12.3
- simplejson 2.0.9 (if Python < 2.6)
- Tempita 0.4
- WebError 0.10.2
- WebHelpers 1.2
- WebOb 0.9.8
- Webtest 1.2.1

These are the current versions as of August 29, 2010. Your installation may have slightly newer or older versions.

27.3 The analysis

27.3.1 Startup (PasteScript)

When you run `paster serve development.ini`, it runs the “\$BIN/paster” program. This is a platform-specific stub created by `pip` or `easy_install`. It does this:

```
__requires__ = 'PasteScript==1.7.3'
import sys
from pkg_resources import load_entry_point

sys.exit(
    load_entry_point('PasteScript==1.7.3', 'console_scripts', 'paster')()
)
```

This says to load a Python object “paster” located in an egg “PasteScript”, version 1.7.3, under the entry point group [console_scripts].

To explain what this means we have to get into Setuptools. Setuptools is Python’s de facto package manager, and was installed as part of your virtualenv or Pylons installation. (If you’re using Distribute 0.6, an alternative package manager, it works the same way.) `load_entry_point` is a function that looks up a Python object via entry point and returns it.

So what’s an entry point? It’s an alias for a Python object. Here’s the entry point itself:

```
[console_scripts]
paster=paste.script.command:run
```

This is from `$SP/PasteScript-VERSION.egg-info/entry_points.txt`. (If you used `easy_install` rather than `pip`, the path would be slightly different: `$APP/PasteScript-VERSION.egg/EGG-INFO/entry_points.txt`.)

“console_scripts” is the entry point group. “paster” is the entry point. The right side of the value tells which module to import (`paste.script.command`) and which object in it to return (the `run` function). (To create an entry point, define it in your package’s `setup.py`. `Pip` or `easy_install` will create the `egg_info` metadata from that. If you modify a package’s entry points, you must reinstall the package to update the `egg_info`.)

The most common use case for entry points is for plugins. So `Nose` for instance defines an entry point group by which it will look for plugins. Any other package can provide plugins for `Nose` by defining entry points in that group. `Paster` uses plugins extensively, as we’ll soon see.

So to make a long story short, “paster serve” calls this `run` function. I inserted print statements into `paste.script.command` to figure out what it does. Here’s a simplified description:

1. The `run()` function parses the command-line options into a subcommand “serve” with arguments [“development.ini”].
2. It calls `get_commands()`, which loads `Paster` commands from plugins located at various entry points. (You can add custom commands with the “-plugin” command-line argument.) `Paste`’s standard commands are listed in the same `entry_points.txt` file we saw above:

```
[paste.global_paster_command]
serve=paste.script.serve:ServeCommand [Config]
#... other commands like "make-config", "setup-app", etc ...
```

3. It calls `invoke()`, which essentially does `paste.script.serve.ServeCommand(["development.ini"]).run()`. This in turn calls `ServeCommand.command()`, which handles daemonizing and other top-level stuff. Since our command line is short, there’s no top-level stuff to do. It creates ‘server’ and ‘app’ objects based on the configuration file, and calls `server(app)`.

27.3.2 Loading the server and the application (PasteDeploy)

This all happens during step 3 of the application startup. We need to find and instantiate the WSGI application and server based on the configuration file. The application is our Analysis application. The server is Paste’s built-in multithreaded HTTP server. A simplified version of the code is:

```
# Inside paste.script.serve module, ServeCommand.command() method.
from paste.deploy.loadwsgi import loadapp, loadserver
server = self.loadserver(server_spec, name=server_name,
                        relative_to=base, global_conf=vars)
app = self.loadapp(app_spec, name=app_name,
                  relative_to=base, global_conf=vars)
```

`loadserver()` and `loadapp()` are defined in module `paste.deploy.loadwsgi`. The code here is complex, so we’ll just look at its general behavior. Both functions see the “config:” URI and read our config file. Since there is no server name or app name they both default to “main”. Therefore `loadserver()` looks for a “[server:main]” section in the config file, and `loadapp()` looks for “[app:main]”. Here’s what they find in “development.ini”:

```
[server:main]
use = egg:Paste#http
host = 127.0.0.1
port = 5000

[app:main]
use = egg:Analysis
full_stack = true
static_files = true
...
```

The “use =” line in each section tells which object to load. The other lines are configuration parameters for that object, or for plugins that object is expected to load. We can also put custom parameters in [app:main] for our application to read directly.

Server loading

1. `loadserver()`’s args are `uri="config:development.ini", name=None, relative_to="$APP"`.
2. A “config:” URI means to read a config file.
3. A server name was not specified so it defaults to “main”. So `loadserver()` looks for a section “[server:main]”. The “server” part comes from the `loadwsgi._Server.config_prefixes` class attribute in `$SP/paste/deploy/loadwsgi.py`.
4. “use = egg:Paste#http” says to load an egg called “Paste”.
5. `loadwsgi._Server.egg_protocols` lists two protocols it supports: “server_factory” and “server_runner”.
6. “paste.server_runner” is an entry point group in the “Paste” egg, and it has an entry point “http”. The relevant lines in `$SP/Paste*.egg_info/entry_points.txt` are:

```
[paste.server_runner]
http = paste.httpserver:server_runner
```

7. There’s a `server_runner()` function in the `paste.httpserver` module (`$SP/paste/httpserver.py`).

We’ll stop here for a moment and look at how the application is loaded.

Application loading

1. `loadapp()` looks for a section “[app:main]” in the config file. The “app” part comes from the `loadwsgi.App.config_prefixes` class attribute (in `$SP/paste/deploy/loadwsgi.py`).
2. “`use = egg:Analysis`” says to find an egg called “Analysis”.
3. `loadwsgi.App.egg_protocols` lists “`paste.app_factory`” as one of the protocols it supports.
4. “`paste.app_factory`” is also an entry point group in the egg, as seen in `$APP/Analysis.egg-info/entry_points.txt`:

```
[paste.app_factory]
main = analysis.config.middleware:make_app
```

5. The line “`main = analysis.config.middleware:make_app`” means to look for a `make_app()` object in the `analysis` package. This is a function imported from `analysis.config.middleware` (`$APP/analysis/config/middleware.py`).

27.3.3 Instantiating the application (Analysis)

Here’s a closer look at our application’s `make_app` function:

```
# In $APP/analysis/config/middleware.py
def make_app(global_conf, full_stack=True, static_files=True, **app_conf):
    config = load_environment(global_conf, app_conf)
    app = PylonsApp(config=config)
    app = SomeMiddleware(app, ...) # Repeated for several middlewares.
    app.config = config
    return app
```

This sets up the Pylons environment (next subsection), creates the application object (following subsection), wraps it in several layers of middleware (listed in “Anatomy of a Request” below), and returns the complete application object.

The [DEFAULT] section of the config file is passed as dict `global_conf`. The [app:main] section is passed as keyword arguments into dict `app_conf`.

`full_stack` defaults to `True` because we’re running the application standalone. If we were embedding this application as a WSGI component of some larger application, we’d set `full_stack` to `False` to disable some of the middleware.

`static_files=True` means to serve static files from our public directory (`$APP/analysis/public`). Advanced users can arrange for Apache to serve the static files itself, and put “`static_files = false`” in their configuration file to gain a bit of efficiency.

load_environment & pylons.config

Before we begin, remember that `pylons.config`, `pylons.app_globals`, `pylons.request`, `pylons.response`, `pylons.session`, `pylons.url`, and `pylons.cache` are special globals that change value depending on the current request. The objects are proxies which maintain a thread-local stack of real values. Pylons pushes the actual values onto them at the beginning of a request, and pops them off at the end. (Some of them it also pushes at other times so they can be used outside of requests.) The proxies delegate attribute access and key access to the topmost actual object on the stack. (You can also call `myproxy._current_obj()` to get the actual object itself.) The proxy code is in `paste.registry.StackedObjectProxy`, so these are called “StackedObjectProxies”, or “SOPs” for short.

The first thing `analysis.config.middleware.make_app()` does is call `analysis.config.environment.load_environment()`:

```
def load_environment(global_conf, app_conf):
    config = PylonsConfig()
    root = os.path.dirname(os.path.abspath(__file__))
    paths = dict(root=root,
                 controllers=os.path.join(root, 'controllers'),
                 static_files=os.path.join(root, 'public'),
                 templates=[os.path.join(root, 'templates')])

    # Initialize config with the basic options
    config.init_app(global_conf, app_conf, package='analysis',
                   paths=paths)
    config['routes.map'] = make_map(config)
    config['pylons.app_globals'] = app_globals.Globals(config)
    config['pylons.h'] = analysis.lib.helpers

    # Setup cache object as early as possible
    import pylons
    pylons.cache._push_object(config['pylons.app_globals'].cache)

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

    # CONFIGURATION OPTIONS HERE (note: all config options will override
    # any Pylons config options)

    return config
```

`config` is the Pylons configuration object, which will later be pushed onto `pylons.config`. It's an instance of `pylons.configuration.PylonsConfig`, a dict subclass. `config.init_app()` initializes the dict's keys. It sets the keys to a merger of `app_conf` and `global_conf` (with `app_conf` overriding). It also adds “`app_conf`” and “`global_conf`” keys so you can access the original `app_conf` and `global_conf` if desired. It also adds several Pylons-specific keys.

`config["routes.map"]` is the Routes map defined in `analysis.config.routing.make_map()`.

`config["pylons.app_globals"]` is the application's globals object, which will later be pushed onto `pylons.app_globals`. It's an instance of `analysis.lib.app_globals.Globals`.

`config["pylons.h"]` is the helpers module, `analysis.lib.helpers`. Pylons will assign it to `h` in the templates' namespace.

The “cache” lines push `pylons.app_globals.cache` onto `pylons.cache` for backward compatibility. This gives a preview of how `StackedObjectProxies` work.

The Mako stanza creates a `TemplateLookup`, which `render()` will use to find templates. The object is put on `app_globals`.

If you've used older versions of Pylons, you'll notice a couple differences in 1.0. The `config` object is created as a local variable and returned, and it's passed explicitly to the route map factory and globals factory. Previous versions pushed it onto `pylons.config` immediately and used it from there. This was changed to make it easier to nest Pylons applications inside other Pylons applications.

The other difference is that `Buffer` is gone, and along with it the `template_engine` argument and template config options. Pylons 1.0 gets out of the business of initializing template engines. You use one of the standard render

functions such as `render_mako` or write your own, and define any attributes in `app_globals` that your render function depends on.

PylonsApp

The second line of `make_app()` creates a Pylons application object based on your configuration. Again the `config` object is passed around explicitly, unlike older versions of Pylons. A Pylons application is an instance of `pylons.wsgiapp.PylonsApp` instance. (Older versions of Pylons had a `PylonsBaseWSGIApp` superclass, but that has been merged into `PylonsApp`.)

Middleware

`make_app()` then wraps the application (the `app` variable) in several layers of middleware. Each middleware provides an optional add-on service.

Mid- dle- ware	Service	Effect if disabled
RoutesMid- dle- ware	Use Routes to manage URLs.	Routes and <code>pylons.url</code> won't work.
Ses- sion- Mid- dle- ware	HTTP sessions using Beaker, with flexible persistence backends (disk, memcached, database).	<code>pylons.session</code> won't work.
Er- rorHan- dler	Display interactive traceback if an exception occurs. In production mode, email the traceback to the site admin.	Paste will catch exceptions and convert them to Internal Server Error.
Status- CodeRedi- rect	If an HTTP error occurs, make a subrequest to display a fancy styled HTML error page.	If an HTTP error occurs, display a plain white HTML page with the error message.
Reg- istry- Man- ager	Handles the special globals (<code>pylons.request</code> , etc).	The special globals won't work. There are other ways to access the objects without going through the special globals.
Static- URL- Parser	Serve the static files in the application's public directory.	The static files won't be found. Presumably you've configured Apache to serve them directly.
Cas- cade	Call several sub-middlewares in order, and use the first one that doesn't return "404 Not Found". Used in conjunction with <code>StaticURLParser</code> .	No cascading through alternative apps.

At the end of the function, `app.config` is set to the `config` object, so that any part of the application can access the config without going through the special global.

27.3.4 Anatomy of a request

Let's say you're running the demo and click the "link" link on the home page. The browser sends a request for "`http://localhost:5000/page2`". In my Firefox the HTTP request headers are:

```
GET /page2
Host: 127.0.0.1:5000
User-Agent: Mozilla/5.0 ...
Accept: text/html,...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://127.0.0.1/5000/
Cache-Control    max-age=0
```

The response is:

```
HTTP/1.x 200 OK
Server: PasteWSGIServer/0.5 Python/2.6.4
Date: Sun, 06 Dec 2009 14:06:05 GMT
Content-Type: text/html; charset=utf-8
Pragma: no-cache
Cache-Control: no-cache
Content-Length: 59

Thank you for using the Analysis Demo.  <a href="/">Home</a>
```

Here's the processing sequence:

1. `server (app)` is still running, called by `ServeCommand.command()` in `$SP/paste/script/serve.py`.
2. `server` is actually `paste.httpserver.server_runner()` in `$SP/paste/httpserver`. The only keyword args are 'host' and 'port' extracted from the config file. `server_runner` de-stringifies the arguments and calls `serve(wsgi_app, **kwargs)` (same module).
3. `serve()`'s 'use_threadpool' arg defaults to `True`, so it creates a `WSGIThreadPoolServer` instance called (`server`) with the following inheritance:

```
SocketServer.BaseServer      # In SocketServer.py in Python stdlib.
BaseHTTPServer.HTTPServer    # In BaseHTTPServer.py in Python stdlib.
paste.httpserver.SecureHTTPServer # Adds SSL (HTTPS).
paste.httpserver.WSGIServerBase # Adds WSGI.
paste.httpserver.WSGIThreadPoolServer
    multiple inheritance: ThreadPoolMixin <= ThreadPool
```

Note that `SecureHTTPServer` overrides the implementation of Python's `SocketServer.TCPServer`

4. It calls `server.serve_forever()`, implemented by the `ThreadPoolMixin` superclass. This calls `self.handle_request()` in a loop until `self.running` becomes false. That initiates this call stack:

```
# In paste.httpserver.serve(), calling 'server.serve_forever()'
ThreadPoolMixin.serve_forever() # Defined in paste.httpserver.
-> TCPServer.handle_request()   # Called for every request.
-> WSGIServerBase.get_request()
-> SecureHTTPServer.get_request()
-> self.socket.accept()         # Defined in stdlib socket module.
```

`self.socket.accept()` blocks, waiting for the next request.

5. The request arrives and `self.socket.accept()` returns a new socket for the connection. `TCPServer.handle_request()` continues. It calls `ThreadPoolMixin.process_request()`, which puts the

request in a thread queue:

```
self.thread_pool.add_task(
    lambda: self.process_request_in_thread(request, client_address))
# 'request' is the connection socket.
```

The thread pool is defined in the `ThreadPool` class. It spawns a number of threads which each wait on the queue for a callable to run. In this case the callable will be a complete Web transaction including sending the HTML page to the client. Each thread will repeatedly process transactions from the queue until they receive a sentinel value ordering them to die.

The main thread goes back to listening for other requests, so we're no longer interested in it.

6. **Thread #2** pulls the lambda out of the queue and calls it:

```
lambda
-> ThreadPoolMixin.process_request_in_thread()
-> BaseServer.finish_request()
-> self.RequestHandlerClass(request, client_address, self) # Instantiates this.
    The class instantiated is paste.httpserver.WSGIHandler; i.e., the 'handler'
    variable in serve().
```

7. The newly-created request handler takes over:

```
SocketServer.BaseRequestHandler.__init__(request, client_address, server)
-> WSGIHandler.handle()
-> BaseHTTPRequestHandler.handle() # In stdlib BaseHTTPServer.py
    Handles requests in a loop until self.close_connection is true. (For HTTP
    keepalive?)
-> WSGIHandler.handle_one_request()
    Reads the command from the socket. The command is
    "GET /page2 HTTP/1.1" plus the HTTP headers above.
    BaseHTTPRequestHandler.parse_request() parses this into attributes
    .command, .path, .request_version, and .headers.
-> WSGIHandlerMixin.wsgi_execute().
-> WSGIHandlerMixin.wsgi_setup()
    Creates the .wsgi_environ dict.
```

The WSGI environment dict is described in PEP 333, the WSGI specification. It contains various keys specifying the URL to fetch, query parameters, server info, etc. All keys required by the CGI specification are present, as are other keys specific to WSGI or to particular middleware. The application will calculate a response based on the dict. The application is wrapped in layers of middleware – nested function calls – which modify the dict on the way in and modify the response on the way out.

8. The request handler, still in `WSGIHandlerMixin.wsgi_execute()`, calls the application thus:

```
result = self.server.wsgi_application(self.wsgi_environ,
                                     self.wsgi_start_response)
```

`wsgi_start_response` is a callable mandated by the WSGI spec. The application will call it to specify the HTTP headers. The return value is an iteration of strings, which when concatenated form the HTML document to send to the browser. Other MIME types are handled analogously.

9. The application, as we remember, was returned by `analysis.config.middleware.make_app()`. It's wrapped in several layers of middleware, so calling it will execute the middleware in reverse order of how they're listed in `$APP/analysis/config/middleware.py` and `$SP/pylons/wsgiapp.py`:

- Cascade (defined in `$SP/paste/cascade.py`) lists a series of applications which will be tried in order (Skipped if `static_files` is set to `False`):

- (a) `StaticURLParser` (defined in `$SP/paste/urlparser`) looks for a file URL under `$APP/analysis/public` that matches the URL. The demo has no static files.
- (b) If that fails the cascader tries your application. But first there are other middleware to go through...
- `RegistryManager` (defined in `$SP/paste/registry.py`) makes Pylons special globals both thread-local and middleware-local. This includes **app_globals**, **cache**, **request**, **response**, **session**, **tmpl_context**, **url**, and any other `StackedObjectProxy` listed in `$SP/pylons/__init__.py`. (**h** is a module so it doesn't need a proxy.)
- `StatusCodeRedirect` (defined in `$SP/pylons/middleware.py`) intercepts any HTTP error status returned by the application (e.g., "Page Not Found", "Internal Server Error") and sends another request to the application to get the appropriate error page to display instead. (Skipped if `full_stack` argument was false.)
- `ErrorHandler` (defined in `$SP/pylons/middleware.py`) sends an interactive traceback to the browser if the app raises an exception, if "debug" is true in the config file. Otherwise it attempts to email the traceback to the site administrator, and substitutes a generic Internal Server Error for the response. (Skipped if `full_stack` argument was false.)
- User-defined middleware goes here.
- `SessionMiddleware` (`wsgiapp.py`) adds **Beaker** session support (the `pylons.session` object). (Skipped if the WSGI environment has a key 'session' – it doesn't in this demo.)
- `RoutesMiddleware` (`wsgiapp.py`) compares the request URI against the routing rules in `$APP/analysis/config/routing.py` and sets 'wsgi.routing_args' to the routing match dict (useful) and 'routes.route' to the Route (probably not useful). Pylons 1.0 apps have a `singleton=False` argument that suppresses initializing the deprecated `url_for()` function. Routes now puts a URL generator in the WSGI environment, which Pylons aliases to `pylons.url`.
- The innermost middleware calls the `PylonsApp` instance it was initialized with.

Note: `CacheMiddleware` is no longer used in Pylons 1.0. Instead, `app_globals` creates the cache as an attribute, and a line in `environment.py` aliases `pylons.cache` to it.

10. Surprise! `PylonsApp` is itself middleware. Its `__call__()` method does:

```
self.setup_app_env(envIRON, start_response)
controller = self.resolve(envIRON, start_response)
response = self.dispatch(controller, envIRON, start_response)
return response
```

`.setup_app_env()` registers all those special globals.

`.resolve()` calculates the controller class based on the route chosen by the `RoutesMiddleware`, and returns the controller class.

`.dispatch` instantiates the controller class and calls in the WSGI manner. If the controller does not exist (`.resolve()` returned `None`), raise an `Exception` that tells you what controller did not have any content.

This method also handles the special URL `"/_test_vars"`, which is enabled if the application is running under a Nose test. This URL initializes Pylons' special globals, for tests that have to access them before making a regular request.

11. `analysis.controllers.main.MainController` does not have a `.___call__()` method, so control falls to its parent, `analysis.lib.base.BaseController`. This trivially calls the grandparent, `pylons.controllers.WSGIController`. It calls the action method `MainController.page2()`. The action method may have any number of positional arguments as long as they correspond to variables in the

routing match dict. (GET/POST variables are in the **request.params** dict.) If the method has a `**kwargs` argument, all other match variables are put there. Any variables passed to the action method are also put on the **tmpl_context** object as attributes. If an action method name starts with “_”, it’s private and HTTPNotFound is raised.

12. If the controller has `__before__()` and/or `__after__()` methods, they are called before and after the action, respectively. These can perform authorization, lock OS resources, etc. These methods can have arguments in the same manner as the action method. However, if the code is used by all controllers, most Pylons programmers prefer to it in the base controller’s `__call__` method instead.
13. The action method returns a string, unicode, Response object, or is a generator of strings. In this trivial case it returns a string. A typical Pylons action would set some *tmpl_context* attributes and ‘return render(‘/some/template.html’)’. In either case the global *response* object’s body would be set to the string.
14. `WSGIController.__call__()` continues, converting the Response object to an appropriate WSGI return value. (First it calls the `start_response` callback to specify the HTTP headers, then it returns an iteration of strings. The Response object converts unicode to utf-8 encoded strings, or whatever encoding you’ve specified in the config file.)
15. The stack of middleware calls unwinds, each modifying the return value and headers if it desires.
16. The server receives the final return value. (We’re way back in `paste.httpserver.WSGIHandlerMixin.wsgi_execute()` now.) The outermost middleware has called back to `server.start_response()`, which has saved the status and HTTP headers in `.wsgi_curr_headers`. `.wsgi_execute()` then iterates the application’s return value, calling `.wsgi_write_chunk(chunk)` for each encoded string yielded. `.wsgi_write_chunk('')` formats the status and HTTP headers and sends them on the socket if they haven’t been sent yet, then sends the chunk. The convoluted header behavior here is mandated by the WSGI spec.
17. Control returns to `BaseHTTPRequestHandler.handle()`. `.close_connection` is `true` so this method returns. The call stack continues unwinding all the way to `paste.httpserver.ThreadPoolMixin.process_request_in_thread()`. This tries to finish the request first and then close it unless it finds errors in it to end raising an Exception.
18. The request lambda finishes and control returns to `ThreadPool.worker_thread_callback()`. It waits for another request in the thread queue. If the next item in the queue is the shutdown sentinel value, thread #2 dies.

Thus endeth our request’s long journey, and this analysis is finished too.

28.1 `pylons.commands` – Command line functions

28.1.1 Module Contents

28.2 `pylons.configuration` – Configuration object and defaults setup

28.2.1 Module Contents

28.3 `pylons.controllers` – Controllers

This module makes available the `WSGIController` and `XMLRPCController` for easier importing.

28.4 pylons.controllers.core – WSGIController Class

28.4.1 Module Contents

28.5 pylons.controllers.util – Controller Utility functions

28.5.1 Module Contents

28.6 pylons.controllers.xmlrpc – XMLRPCController Class

28.6.1 Module Contents

28.7 pylons.decorators – Decorators

28.7.1 Module Contents

28.8 pylons.decorators.cache – Cache Decorators

28.8.1 Module Contents

28.9 pylons.decorators.rest – REST-ful Decorators

28.9.1 Module Contents

28.10 pylons.decorators.secure – Secure Decorators

28.10.1 Module Contents

28.11 pylons.error – Error handling support

28.12 pylons.i18n.translation – Translation/Localization functions

28.12.1 Module Contents

28.13 pylons.log – Logging for WSGI errors

28.13.1 Module Contents

28.14 pylons.middleware – WSGI Middleware

28.14.1 Module Contents

Note:

The **errorware** dictionary is constructed from the settings in the *DEFAULT* section of `development.ini`. the recognised keys are:

- `error_email = conf.get('email_to')`
 - `error_log = conf.get('error_log', None)`
 - `smtp_server = conf.get('smtp_server', 'localhost')`
 - `error_subject_prefix = conf.get('error_subject_prefix', 'WebApp Error: ')`
 - `from_address = conf.get('from_address', conf.get('error_email_from', 'pylons@yourapp.com'))`
 - `error_message = conf.get('error_message', 'An internal server error occurred')`
-

28.14.2 Referenced classes

Pylons middleware uses `WebError` to effect the error-handling. The two classes implicated are:

ErrorMiddleware

`weberror.errormiddleware` `weberror.errormiddleware.ErrorMiddleware`

EvalException

`weberror.evaexception` `weberror.evaexception.EvalException`

28.15 pylons.templating – Render functions and helpers

28.15.1 Module Contents

28.16 pylons.test – Test related functionality

28.16.1 Module Contents

28.17 pylons.util – Paste Template and Pylons utility functions

28.17.1 Module Contents

28.18 pylons.wsgiapp – PylonsWSGI App Creator

28.18.1 Module Contents

Third-party components

29.1 FormEncode

FormEncode is a validation and form generation package. The validation can be used separately from the form generation. The validation works on compound data structures, with all parts being nestable. It is separate from HTTP or any other input mechanism.

These module API docs are divided into section by category.

29.1.1 Core API

`formencode.api`

These functions are used mostly internally by FormEncode.

`is_validator(obj)`

Returns whether `obj` is a validator object or not.

`formencode.schema`

The FormEncode schema is one of the most important parts of using FormEncode, as it lets you organize validators into parts that can be re-used between schemas. Generally, a single schema will represent an entire form, but may inherit other schemas for re-usable validation parts (ie, maybe multiple forms all requires first and last name).

29.1.2 Validators

Wrapper Validators

29.1.3 Validator Modifiers

`formencode.compound`

`formencode.foreach`

29.1.4 HTML Parsing and Form Filling

`formencode.htmlfill`

29.2 weberror – Weberror

29.2.1 `weberror.errormiddleware`

29.2.2 `weberror.evalcontext`

29.2.3 `weberror.evaexception`

29.2.4 `weberror.formatter`

29.2.5 `weberror.reporter`

29.2.6 `weberror.collector`

29.3 webtest – WebTest

29.4 webob – Request/Response objects

29.4.1 Request

29.4.2 Response

29.4.3 Misc Functions

29.4.4 Descriptors

action The class method in a Pylons applications' controller that handles a request.

API Application Programming Interface. The means of communication between a programmer and a software program or operating system.

app_globals The `app_globals` object is created on application instantiation by the `Globals` class in a projects `lib/app_globals.py` module.

This object is created once when the application is loaded by the projects `config/environment.py` module (See [Environment](#)). It remains persistent during the lifecycle of the web application, and is *not* thread-safe which means that it is best used for global options that should be *read-only*, or as an object to attach db connections or other objects which ensure their own access is thread-safe.

c Commonly used alias for [templ_context](#) to save on the typing when using lots of controller populated variables in templates.

caching The storage of the results of expensive or length computations for later re-use at a point more quickly accessed by the end user.

CDN Content Delivery Networks (CDN's) are generally globally distributed content delivery networks optimized for low latency for static file distribution. They can significantly increase page-load times by ensuring that the static resources on a page are delivered by servers geographically close to the client in addition to lightening the load placed on the application server.

ColdFusion Components CFCs represent an attempt by Macromedia to bring ColdFusion closer to an Object Oriented Programming (OOP) language. ColdFusion is in no way an OOP language, but thanks in part to CFCs, it does boast some of the attributes that make OOP languages so popular.

config The `PylonsConfig` instance for a given application. This can be accessed as `pylons.config` after an Pylons application has been loaded.

controller The 'C' in MVC. The controller is given a request, does the necessary logic to prepare data for display, then renders a template with the data and returns it to the user. See [Controllers](#).

easy_install A tool that lets you download, build, install and manage Python packages and their dependencies. [easy_install](#) is the end-user facing component of [setuptools](#).

Pylons can be installed with `easy_install`, and applications built with Pylons can easily be deployed this way as well.

See also:

Pylons *Packaging and Deployment Overview*

dotted name string A reference to a Python module by name using a string to identify it, e.g. `pylons.controllers.util`. These strings are evaluated to import the module being referenced without having to import it in the code used. This is generally used to avoid import-time side-effects.

egg Python egg's are bundled Python packages, generally installed by a package called *setuptools*. Unlike normal Python package installs, egg's allow a few additional features, such as package dependencies, and dynamic discovery.

See also:

The Quick Guide to Python Eggs

EJBs Enterprise JavaBeans (EJB) technology is the server-side component architecture for Java Platform, Enterprise Edition (Java EE). EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology.

environ `environ` is a dictionary passed into all *WSGI* application. It generally contains unparsed header information, CGI style variables and other objects inserted by *WSGI Middleware*.

ETag An ETag (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL. See http://wikipedia.org/wiki/HTTP_ETag

g Alias used in prior versions of Pylons for *app_globals*.

Google App Engine A cloud computing platform for hosting web applications implemented in Python. Building Pylons applications for App Engine is facilitated by Ian Bicking's *appengine-monkey project*.

See also:

What is Google App Engine? - Official Doc

h The helpers reference, `h`, is made available for use inside templates to assist with common rendering tasks. `h` is just a reference to the `lib/helpers.py` module and can be used in the same manner as any other module import.

Model-View-Controller An architectural pattern used in software engineering. In Pylons, the MVC paradigm is extended slightly with a pipeline that may transform and extend the data available to a controller, as well as the Pylons *WSGI* app itself that determines the appropriate Controller to call.

See also:

MVC at Wikipedia

MVC See *Model-View-Controller*

ORM (Object-Relational Mapper) Maps relational databases such as MySQL, Postgres, Oracle to objects providing a cleaner API. Most ORM's also make it easier to prevent SQL Injection attacks by binding variables, and can handle generating sometimes extensive SQL.

Pylons A Python-based WSGI oriented web framework.

Rails Abbreviated as RoR, Ruby on Rails (also referred to as just Rails) is an open source Web application framework, written in Ruby

request Refers to the current request being processed. Available to import from `pylons` and is available for use in templates by the same name. See *Request*.

response Refers to the response to the current request. Available to import from `pylons` and is available for use in template by the same name. See [Response](#).

route Routes determine how the URL's are mapped to the controllers and which URL is generated. See [URL Configuration](#)

setuptools An extension to the basic `distutils`, `setuptools` allows packages to specify package dependencies and have dynamic discovery of other installed Python packages.

See also:

[Building and Distributing Packages with setuptools](#)

SQLAlchemy One of the most popular Python database object-relational mappers (*ORM*). [SQLAlchemy](#) is the default ORM recommended in Pylons. SQLAlchemy at the ORM level can look similar to Rails ActiveRecord, but uses the [DataMapper](#) pattern for additional flexibility with the ability to map simple to extremely complex databases.

tmpl_context The `tmpl_context` is available in the `pylons` module, and refers to the template context. Objects attached to it are available in the template namespace as either `tmpl_context` or `c` for convenience.

UI User interface. The means of communication between a person and a software program or operating system.

virtualenv A tool to create isolated Python environments, designed to supersede the `workingenv` package and [virtual python](#) configurations. In addition to isolating packages from possible system conflicts, `virtualenv` makes it easy to install Python libraries using [easy_install](#) without dumping lots of packages into the system-wide Python.

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

`virtualenv` is employed automatically by the `go-pylons.py` script described in [Getting Started](#). The Pylons wiki has more information on [working with virtualenv](#).

web server gateway interface A specification for web servers and application servers to communicate with web applications. Also referred to by its initials, as [WSGI](#).

WSGI The [WSGI Specification](#), also commonly referred to as PEP 333 and described by [PEP 333](#).

WSGI Middleware [WSGI](#) Middleware refers to the ability of WSGI applications to modify the environ, and/or the content of other WSGI applications by being placed in between the request and the other WSGI application.

See also:

[WSGI Middleware in Concepts of Pylons WSGI Middleware Configuration](#)

b

`babel`, [91](#)
`babel.core`, [91](#)
`babel.dates`, [103](#)
`babel.localedata`, [103](#)
`babel.numbers`, [105](#)

f

`formencode.schema`, [191](#)

p

`pylons.controllers`, [187](#)

A

action, [193](#)
API, [193](#)
app_globals, [193](#)

B

babel (module), [91](#)
babel.core (module), [91](#)
babel.dates (module), [103](#)
babel.localedata (module), [103](#)
babel.numbers (module), [105](#)

C

c, [193](#)
caching, [193](#)
CDN, [193](#)
character_order (babel.Locale attribute), [91](#)
ColdFusion Components, [193](#)
config, [193](#)
controller, [193](#)
currencies (babel.Locale attribute), [92](#)
currency_formats (babel.Locale attribute), [92](#)
currency_symbols (babel.Locale attribute), [92](#)

D

date_formats (babel.Locale attribute), [92](#)
datetime_formats (babel.Locale attribute), [92](#)
datetime_skeletons (babel.Locale attribute), [93](#)
DateTimeFormat (class in babel.dates), [104](#)
DateTimePattern (class in babel.dates), [105](#)
day_period_rules (babel.Locale attribute), [93](#)
day_periods (babel.Locale attribute), [93](#)
days (babel.Locale attribute), [93](#)
decimal_formats (babel.Locale attribute), [93](#)
default() (babel.Locale class method), [93](#)
default_locale() (in module babel), [101](#)
display_name (babel.Locale attribute), [94](#)
dotted name string, [194](#)

E

easy_install, [193](#)
egg, [194](#)
EJBs, [194](#)
english_name (babel.Locale attribute), [94](#)
environ, [194](#)
environment variable
 PATH_INFO, [15](#), [16](#)
 SCRIPT_NAME, [15](#), [16](#)
 SERVER_NAME, [164](#)
eras (babel.Locale attribute), [94](#)
ETag, [194](#)
exists() (in module babel.localedata), [103](#)

F

first_week_day (babel.Locale attribute), [94](#)
format_decimal() (in module babel.numbers), [105](#)
format_frac_seconds() (babel.dates.DateTimeFormat
 method), [104](#)
format_number() (in module babel.numbers), [105](#)
format_percent() (in module babel.numbers), [106](#)
format_scientific() (in module babel.numbers), [106](#)
format_weekday() (babel.dates.DateTimeFormat
 method), [104](#)
formencode.schema (module), [191](#)

G

g, [194](#)
get_display_name() (babel.Locale method), [94](#)
get_language_name() (babel.Locale method), [95](#)
get_script_name() (babel.Locale method), [95](#)
get_territory_name() (babel.Locale method), [95](#)
get_week_number() (babel.dates.DateTimeFormat
 method), [104](#)
Google App Engine, [194](#)

H

h, [194](#)

I

interval_formats (babel.Locale attribute), 95
is_validator() (built-in function), 191

L

language (babel.Locale attribute), 95
language_name (babel.Locale attribute), 95
languages (babel.Locale attribute), 95
list_patterns (babel.Locale attribute), 95
Locale (class in babel), 91

M

measurement_systems (babel.Locale attribute), 96
meta_zones (babel.Locale attribute), 96
min_week_days (babel.Locale attribute), 96
Model-View-Controller, 194
months (babel.Locale attribute), 96
MVC, 194

N

negotiate() (babel.Locale class method), 96
negotiate_locale() (in module babel), 101
number_symbols (babel.Locale attribute), 97
NumberFormatError (class in babel.numbers), 105
NumberPattern (class in babel.numbers), 105

O

ordinal_form (babel.Locale attribute), 97
ORM, 194

P

parse() (babel.Locale class method), 97
parse_decimal() (in module babel.numbers), 107
parse_locale() (in module babel), 102
parse_number() (in module babel.numbers), 107
PATH_INFO, 15, 16
percent_formats (babel.Locale attribute), 98
periods (babel.Locale attribute), 98
plural_form (babel.Locale attribute), 98
Pylons, 194
pylons.controllers (module), 187
Python Enhancement Proposals
 PEP 333, 8, 195

Q

quarters (babel.Locale attribute), 98

R

Rails, 194
request, 194
response, 195
RFC
 RFC 3066, 91

 RFC 4646, 103

route, 195

S

scientific_formats (babel.Locale attribute), 98
script (babel.Locale attribute), 99
SCRIPT_NAME, 15, 16
script_name (babel.Locale attribute), 99
scripts (babel.Locale attribute), 99
SERVER_NAME, 164
setuptools, 195
SQLAlchemy, 195

T

territories (babel.Locale attribute), 99
territory (babel.Locale attribute), 99
territory_name (babel.Locale attribute), 99
text_direction (babel.Locale attribute), 99
time_formats (babel.Locale attribute), 99
time_zones (babel.Locale attribute), 100
tmpl_context, 195

U

UI, 195
unit_display_names (babel.Locale attribute), 100

V

variant (babel.Locale attribute), 100
variants (babel.Locale attribute), 100
virtualenv, 195

W

web server gateway interface, 195
weekend_end (babel.Locale attribute), 100
weekend_start (babel.Locale attribute), 100
WSGI, 195
WSGI Middleware, 195

Z

zone_formats (babel.Locale attribute), 100