

The Pyramid Web Application Development Framework

Version 1.0

Chris McDonough

CONTENTS

Front Matter	i
Copyright, Trademarks, and Attributions	iii
Attributions	iv
Print Production	iv
Contacting The Publisher	iv
HTML Version and Source Code	iv
Typographical Conventions	v
Author Introduction	vii
Audience	vii
Book Content	viii
The Genesis of <code>repoze.bfg</code>	viii
The Genesis of Pyramid	ix
Thanks	ix
I Narrative Documentation	1
1 Pyramid Introduction	3
1.1 What Is The Pylons Project?	4
1.2 Pyramid and Other Web Frameworks	4
2 Installing Pyramid	7
2.1 Before You Install	7
2.1.1 If You Don't Yet Have A Python Interpreter (UNIX)	7
2.1.2 If You Don't Yet Have A Python Interpreter (Windows)	9
2.2 Installing Pyramid on a UNIX System	9

2.2.1	Installing the <code>virtualenv</code> Package	10
2.2.2	Creating the Virtual Python Environment	10
2.2.3	Installing Pyramid Into the Virtual Python Environment	11
2.3	Installing Pyramid on a Windows System	11
2.4	Installing Pyramid on Google App Engine	12
2.5	Installing Pyramid on Jython	12
2.6	What Gets Installed	12
3	Application Configuration	13
3.1	Imperative Configuration	13
3.2	Configuration Decorations and Code Scanning	14
3.3	Declarative Configuration	15
4	Creating Your First Pyramid Application	17
4.1	Hello World, Goodbye World	17
4.1.1	Imports	18
4.1.2	View Callable Declarations	18
4.1.3	Application Configuration	19
4.1.4	Configurator Construction	20
4.1.5	Adding Configuration	20
4.1.6	WSGI Application Creation	21
4.1.7	WSGI Application Serving	22
4.1.8	Conclusion	22
4.2	References	23
5	Creating a Pyramid Project	25
5.1	Paster Templates Included with Pyramid	25
5.2	Creating the Project	26
5.3	Installing your Newly Created Project for Development	28
5.4	Running The Tests For Your Application	28
5.5	The Interactive Shell	29
5.6	Running The Project Application	31
5.7	Viewing the Application	32
5.8	The Project Structure	34
5.9	The <code>MyProject</code> <i>Project</i>	35
5.9.1	<code>development.ini</code>	35
5.9.2	<code>production.ini</code>	39
5.9.3	<code>setup.py</code>	39
5.9.4	<code>setup.cfg</code>	41
5.10	The <code>myproject</code> <i>Package</i>	42
5.10.1	<code>__init__.py</code>	43
5.10.2	<code>views.py</code>	44
5.10.3	<code>resources.py</code>	44
5.10.4	<code>static</code>	45

5.10.5	templates/mytemplate.pt	45
5.10.6	tests.py	45
5.11	Modifying Package Structure	46
6	URL Dispatch	49
6.1	High-Level Operational Overview	49
6.2	Route Configuration	50
6.2.1	Configuring a Route via The <code>add_route</code> Configurator Method	50
6.2.2	Route Configuration That Names a View Callable	50
6.2.3	Route Pattern Syntax	51
6.2.4	Route Declaration Ordering	54
6.2.5	Route Factories	55
6.2.6	Route Configuration Arguments	55
6.2.7	Custom Route Predicates	56
6.3	Route Matching	58
6.3.1	The Matchdict	59
6.3.2	The Matched Route	59
6.4	Routing Examples	59
6.4.1	Example 1	59
6.4.2	Example 2	60
6.4.3	Example 3	61
6.4.4	Example 4	62
6.5	Matching the Root URL	62
6.6	Generating Route URLs	62
6.7	Redirecting to Slash-Appended Routes	63
6.7.1	Custom Not Found View With Slash Appended Routes	64
6.8	Cleaning Up After a Request	64
6.9	Using Pyramid Security With URL Dispatch	65
6.10	Debugging Route Matching	66
6.11	Displaying All Application Routes	66
6.12	Route View Callable Registration and Lookup Details	67
6.13	References	68
7	Much Ado About Traversal	69
7.1	URL Dispatch	70
7.2	Historical Refresher	70
7.3	Traversal (aka Resource Location)	71
7.4	What Is a “Resource”?	72
7.5	View Lookup	73
7.6	Use Cases	74
8	Traversal	77
8.1	Traversal Details	77
8.2	The Resource Tree	78

8.3	The Traversal Algorithm	80
8.3.1	A Description of The Traversal Algorithm	81
8.3.2	Traversal Algorithm Examples	84
8.4	References	86
9	Views	87
9.1	View Callables	88
9.2	Defining a View Callable as a Function	88
9.3	Defining a View Callable as a Class	88
9.4	Alternate View Callable Argument/Calling Conventions	89
9.5	View Callable Responses	91
9.6	Using a View Callable to Do an HTTP Redirect	91
9.7	Using Special Exceptions In View Callables	92
9.8	Exception Views	92
9.9	Handling Form Submissions in View Callables (Unicode and Character Set Issues)	94
10	Renderers	97
10.1	Writing View Callables Which Use a Renderer	98
10.2	Built-In Renderers	98
10.2.1	string: String Renderer	99
10.2.2	json: JSON Renderer	99
10.2.3	*.pt or *.txt: Chameleon Template Renderers	100
10.2.4	*.mak or *.mako: Mako Template Renderer	101
10.3	Varying Attributes of Rendered Responses	102
10.4	Adding and Changing Renderers	103
10.4.1	Adding a New Renderer	103
10.4.2	Changing an Existing Renderer	106
10.5	Overriding A Renderer At Runtime	106
11	Templates	109
11.1	Using Templates Directly	109
11.2	System Values Used During Rendering	113
11.3	Templates Used as Renderers via Configuration	114
11.4	<i>Chameleon</i> ZPT Templates	115
11.4.1	A Sample ZPT Template	116
11.4.2	Using ZPT Macros in Pyramid	117
11.5	Templating with <i>Chameleon</i> Text Templates	118
11.6	Side Effects of Rendering a Chameleon Template	118
11.7	Nicer Exceptions in Chameleon Templates	119
11.8	<i>Chameleon</i> Template Internationalization	120
11.9	Templating With Mako Templates	120
11.9.1	A Sample Mako Template	121
11.10	Automatically Reloading Templates	122
11.11	Available Add-On Template System Bindings	122

12 View Configuration	123
12.1 View Lookup and Invocation	123
12.2 Mapping a Resource or URL Pattern to a View Callable	124
12.2.1 View Configuration Parameters	124
12.2.2 View Configuration Using the <code>@view_config</code> Decorator	129
12.2.3 View Registration Using <code>add_view()</code>	133
12.2.4 Using Resource Interfaces In View Configuration	133
12.2.5 Configuring View Security	135
12.2.6 <code>NotFound</code> Errors	135
13 Resources	137
13.1 Defining a Resource Tree	138
13.2 Location-Aware Resources	139
13.3 Generating The URL Of A Resource	140
13.3.1 Overriding Resource URL Generation	142
13.4 Generating the Path To a Resource	142
13.5 Finding a Resource by Path	143
13.6 Obtaining the Lineage of a Resource	144
13.7 Determining if a Resource is In The Lineage of Another Resource	144
13.8 Finding the Root Resource	145
13.9 Resources Which Implement Interfaces	145
13.10 Finding a Resource With a Class or Interface in Lineage	147
13.11 Pyramid API Functions That Act Against Resources	148
14 Static Assets	149
14.1 Understanding Asset Specifications	149
14.2 Serving Static Assets	150
14.2.1 Generating Static Asset URLs	152
14.3 Advanced: Serving Static Assets Using a View Callable	153
14.3.1 Root-Relative Custom Static View (URL Dispatch Only)	154
14.3.2 Registering A View Callable to Serve a “Static” Asset	154
14.4 Overriding Assets	155
14.4.1 The <code>override_asset</code> API	156
15 Request and Response Objects	159
15.1 Request	160
15.1.1 Special Attributes Added to the Request by Pyramid	160
15.1.2 URLs	161
15.1.3 Methods	161
15.1.4 Unicode	162
15.1.5 More Details	162
15.2 Response	162
15.2.1 Headers	163
15.2.2 Instantiating the Response	163

15.2.3	Exception Responses	164
15.2.4	More Details	165
15.3	Multidict	165
16	Sessions	167
16.1	Using The Default Session Factory	167
16.2	Using a Session Object	168
16.3	Using Alternate Session Factories	169
16.4	Creating Your Own Session Factory	169
16.5	Flash Messages	170
16.5.1	Using the <code>session.flash</code> Method	170
16.5.2	Using the <code>session.pop_flash</code> Method	171
16.5.3	Using the <code>session.peek_flash</code> Method	171
16.6	Preventing Cross-Site Request Forgery Attacks	172
16.6.1	Using the <code>session.get_csrf_token</code> Method	172
16.6.2	Using the <code>session.new_csrf_token</code> Method	172
17	Security	175
17.1	Enabling an Authorization Policy	176
17.1.1	Enabling an Authorization Policy Imperatively	176
17.2	Protecting Views with Permissions	177
17.2.1	Setting a Default Permission	177
17.3	Assigning ACLs to your Resource Objects	178
17.4	Elements of an ACL	179
17.5	Special Principal Names	181
17.6	Special Permissions	181
17.7	Special ACEs	182
17.8	ACL Inheritance and Location-Awareness	182
17.9	Changing the Forbidden View	183
17.10	Debugging View Authorization Failures	183
17.11	Debugging Imperative Authorization Failures	183
17.12	Creating Your Own Authentication Policy	184
17.13	Creating Your Own Authorization Policy	184
18	Combining Traversal and URL Dispatch	187
18.1	A Review of Non-Hybrid Applications	187
18.1.1	URL Dispatch Only	187
18.1.2	Traversal Only	188
18.2	Hybrid Applications	188
18.2.1	The Root Object for a Route Match	189
18.2.2	Using <code>*traverse</code> In a Route Pattern	190
18.2.3	Using the <code>traverse</code> Argument In a Route Definition	193
18.2.4	Using <code>*subpath</code> in a Route Pattern	194
18.3	Corner Cases	194

18.3.1	Registering a Default View for a Route That Has a <code>view</code> Attribute	194
18.3.2	Binding Extra Views Against a Route Configuration that Doesn't Have a <code>*traverse</code> Element In Its Pattern	195
19	Internationalization and Localization	197
19.1	Creating a Translation String	197
19.1.1	Using The <code>TranslationString</code> Class	197
19.1.2	Using the <code>TranslationStringFactory</code> Class	199
19.2	Working With <code>gettext</code> Translation Files	200
19.2.1	Installing Babel	201
19.2.2	Extracting Messages from Code and Templates	202
19.2.3	Initializing a Message Catalog File	204
19.2.4	Updating a Catalog File	204
19.2.5	Compiling a Message Catalog File	205
19.3	Using a Localizer	205
19.3.1	Performing a Translation	205
19.3.2	Performing a Pluralization	206
19.4	Obtaining the Locale Name for a Request	207
19.5	Performing Date Formatting and Currency Formatting	207
19.6	Chameleon Template Support for Translation Strings	208
19.7	Mako Pyramid I18N Support	209
19.8	Localization-Related Deployment Settings	209
19.9	“Detecting” Available Languages	210
19.10	Activating Translation	211
19.10.1	Adding a Translation Directory	211
19.10.2	Setting the Locale	212
19.11	Locale Negotiators	212
19.11.1	The Default Locale Negotiator	212
19.11.2	Using a Custom Locale Negotiator	213
20	Virtual Hosting	215
20.1	Hosting an Application Under a URL Prefix	215
20.2	Virtual Root Support	216
20.3	Further Documentation and Examples	217
21	Using Events	219
21.1	Configuring an Event Listener Imperatively	219
21.2	Configuring an Event Listener Using a Decorator	220
21.3	An Example	221
22	Environment Variables and <code>.ini</code> File Settings	223
22.1	Reloading Templates	223
22.2	Reloading Assets	224
22.3	Debugging Authorization	224

22.4	Debugging Not Found Errors	224
22.5	Debugging Route Matching	224
22.6	Debugging All	225
22.7	Reloading All	225
22.8	Default Locale Name	225
22.9	Mako Template Render Settings	225
22.9.1	Mako Directories	225
22.9.2	Mako Module Directory	226
22.9.3	Mako Input Encoding	226
22.9.4	Mako Error Handler	226
22.9.5	Mako Default Filters	226
22.9.6	Mako Import	227
22.9.7	Mako Strict Undefined	227
22.10	Examples	227
22.11	Understanding the Distinction Between <code>reload_templates</code> and <code>reload_assets</code>	228
23	Unit, Integration, and Functional Testing	229
23.1	Test Set Up and Tear Down	230
23.1.1	What?	232
23.2	Using the <code>Configurator</code> and <code>pyramid.testing</code> APIs in Unit Tests	232
23.3	Creating Integration Tests	234
23.4	Creating Functional Tests	235
24	Using Hooks	237
24.1	Changing the Not Found View	237
24.2	Changing the Forbidden View	238
24.3	Changing the Request Factory	239
24.4	Adding Renderer Globals	240
24.5	Using The Before Render Event	241
24.6	Using Response Callbacks	242
24.7	Using Finished Callbacks	242
24.8	Changing the Traverser	243
24.9	Changing How <code>pyramid.url.resource_url</code> Generates a URL	245
24.10	Using a View Mapper	246
24.11	Registering Configuration Decorators	248
25	Advanced Configuration	251
25.1	Conflict Detection	251
25.1.1	Manually Resolving Conflicts	253
25.1.2	Automatic Conflict Resolution	256
25.1.3	Methods Which Provide Conflict Detection	256
25.2	Including Configuration from External Sources	256
25.3	Two-Phase Configuration	257
25.4	Adding Methods to the Configurator via <code>add_directive</code>	258

26	Extending An Existing Pyramid Application	261
26.1	The Difference Between “Extensible” and “Pluggable” Applications	261
26.2	Rules for Building An Extensible Application	262
26.2.1	Fundamental Plugpoints	263
26.3	Extending an Existing Application	263
26.3.1	If The Application Has Configuration Decorations	263
26.3.2	Extending the Application	264
26.3.3	Overriding Views	265
26.3.4	Overriding Routes	266
26.3.5	Overriding Assets	266
27	Startup	267
27.1	The Startup Process	267
27.2	Deployment Settings	271
28	Thread Locals	273
28.1	Why and How Pyramid Uses Thread Local Variables	273
28.2	Why You Shouldn’t Abuse Thread Locals	274
29	Using the Zope Component Architecture in Pyramid	277
29.1	Using the ZCA Global API in a Pyramid Application	278
29.1.1	Disusing the Global ZCA API	278
29.1.2	Enabling the ZCA Global API by Using <code>hook_zca</code>	279
29.1.3	Enabling the ZCA Global API by Using The ZCA Global Registry	280
II	Tutorials	283
30	ZODB + Traversal Wiki Tutorial	285
30.1	Background	285
30.2	Installation	286
30.2.1	Preparation	286
30.2.2	Making a Project	288
30.2.3	Installing the Project in “Development Mode”	288
30.2.4	Running the Tests	289
30.2.5	Starting the Application	289
30.2.6	Exposing Test Coverage Information	290
30.2.7	Visit the Application in a Browser	290
30.2.8	Decisions the <code>pyramid_zodb</code> Template Has Made For You	290
30.3	Basic Layout	291
30.3.1	App Startup with <code>__init__.py</code>	291
30.3.2	Resources and Models with <code>models.py</code>	292
30.3.3	Views With <code>views.py</code>	293
30.3.4	The WSGI Pipeline in <code>development.ini</code>	294

30.4	Defining the Domain Model	296
30.4.1	Deleting the Database	297
30.4.2	Adding Model Classes	297
30.4.3	Looking at the Result of Our Edits to <code>models.py</code>	298
30.4.4	Removing View Configuration	298
30.4.5	Testing the Models	299
30.4.6	Declaring Dependencies in Our <code>setup.py</code> File	301
30.4.7	Running the Tests	302
30.5	Defining Views	302
30.5.1	Adding View Functions	303
30.5.2	Viewing the Result of Our Edits to <code>views.py</code>	306
30.5.3	Adding Templates	307
30.5.4	Testing the Views	311
30.5.5	Running the Tests	314
30.5.6	Viewing the Application in a Browser	315
30.6	Adding Authorization	315
30.6.1	Configuring a <code>pyramid</code> Authentication Policy	316
30.6.2	Giving Our Root Resource an ACL	321
30.6.3	Adding <code>permission</code> Declarations to our <code>view_config</code> Decorators	322
30.6.4	Viewing the Application in a Browser	323
30.6.5	Seeing Our Changes To <code>views.py</code> and our Templates	324
30.6.6	Revisiting the Application	328
30.7	Distributing Your Application	329
31	SQLAlchemy + URL Dispatch Wiki Tutorial	331
31.1	Background	331
31.2	Installation	332
31.2.1	Preparation	332
31.2.2	Making a Project	334
31.2.3	Installing the Project in “Development Mode”	334
31.2.4	Running the Tests	335
31.2.5	Starting the Application	335
31.2.6	Exposing Test Coverage Information	336
31.2.7	Visit the Application in a Browser	336
31.2.8	Decisions the <code>pyramid_routesalchemy</code> Template Has Made For You	337
31.3	Basic Layout	337
31.3.1	App Startup with <code>__init__.py</code>	337
31.3.2	Content Models with <code>models.py</code>	339
31.4	Defining the Domain Model	340
31.4.1	Making Edits to <code>models.py</code>	341
31.4.2	Looking at the Result of Our Edits to <code>models.py</code>	341
31.4.3	Viewing the Application in a Browser	342
31.5	Defining Views	343

31.5.1	Declaring Dependencies in Our <code>setup.py</code> File	343
31.5.2	Adding View Functions	345
31.5.3	Viewing the Result of Our Edits to <code>views.py</code>	347
31.5.4	Adding Templates	349
31.5.5	Mapping Views to URLs in <code>__init__.py</code>	352
31.5.6	Viewing the Application in a Browser	353
31.5.7	Adding Tests	354
31.6	Adding Authorization	358
31.6.1	Changing <code>__init__.py</code> For Authorization	358
31.6.2	Viewing the Application in a Browser	365
31.6.3	Seeing Our Changes To <code>views.py</code> and our Templates	365
31.6.4	Revisiting the Application	370
31.7	Distributing Your Application	370
32	Converting a <code>repoze.bfg</code> Application to Pyramid	371
33	Running Pyramid on Google's App Engine	375
33.1	Zippping Files Via Pip	378
34	Running a Pyramid Application under <code>mod_wsgi</code>	381
III	API Reference	385
35	<code>pyramid.authorization</code>	387
36	<code>pyramid.authentication</code>	389
36.1	Authentication Policies	389
36.2	Helper Classes	391
37	<code>pyramid.chameleon_text</code>	393
38	<code>pyramid.chameleon_zpt</code>	395
39	<code>pyramid.config</code>	397
40	<code>pyramid.events</code>	423
40.1	Functions	423
40.2	Event Types	424
41	<code>pyramid.exceptions</code>	427
42	<code>pyramid.httpexceptions</code>	429
42.1	HTTP Exception	429
42.2	Subclass usage notes:	432

43	pyramid.i18n	441
44	pyramid.interfaces	445
	44.1 Event-Related Interfaces	445
	44.2 Other Interfaces	447
45	pyramid.location	453
46	pyramid.paster	455
47	pyramid.registry	457
48	pyramid.renderers	459
49	pyramid.request	461
50	pyramid.response	479
51	pyramid.scripting	485
52	pyramid.security	487
	52.1 Authentication API Functions	487
	52.2 Authorization API Functions	488
	52.3 Constants	489
	52.4 Return Values	489
53	pyramid.settings	491
54	pyramid.testing	493
55	pyramid.threadlocal	497
56	pyramid.traversal	499
57	pyramid.url	507
58	pyramid.view	515
59	pyramid.wsgi	519
IV	Glossary and Index	521
	Glossary	523
	Index	535

Front Matter

COPYRIGHT, TRADEMARKS, AND ATTRIBUTIONS

The Pyramid Web Application Development Framework, Version 1.0

by Chris McDonough

Copyright © 2008-2011, Agendaless Consulting.

ISBN-10: 0615445675

ISBN-13: 978-0615445670

First print publishing: February, 2011

All rights reserved. This material may be copied or distributed only subject to the terms and conditions set forth in the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. You must give the original author credit. You may not use this work for commercial purposes. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.



While the Pyramid documentation is offered under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License, the Pyramid *software* is offered under a less restrictive (BSD-like) license .

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. However, use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as-is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book. No patent liability is assumed with respect to the use of the information contained herein.

Attributions

Editor: Casey Duncan

Contributors:

Ben Bangert, Blaise Laflamme, Rob Miller, Mike Orr, Carlos de la Guardia, Paul Everitt, Tres Seaver, John Shipman, Marius Gedminas, Chris Rossi, Joachim Krebs, Xavier Spriet, Reed O'Brien, William Chambers, Charlie Choiniere, Jamaludin Ahmad, Graham Higgins.

Used with permission:

The *Request and Response Objects* chapter is adapted, with permission, from documentation originally written by Ian Bicking.

The *Much Ado About Traversal* chapter is adapted, with permission, from an article written by Rob Miller.

Print Production

The print version of this book was produced using the Sphinx documentation generation system and the LaTeX typesetting system.

Contacting The Publisher

Please send documentation licensing inquiries, translation inquiries, and other business communications to Agendaless Consulting. Please send software and other technical queries to the Pylons-devel maillist.

HTML Version and Source Code

An HTML version of this book is freely available via <http://docs.pylonsproject.org>

The source code for the examples used in this book are available within the Pyramid software distribution, always available via <https://github.com/Pylons/pyramid>

TYPOGRAPHICAL CONVENTIONS

Literals, filenames and function arguments are presented using the following style:

```
argument1
```

Warnings, which represent limitations and need-to-know information related to a topic or concept are presented in the following style:



This is a warning.

Notes, which represent additional information related to a topic or concept are presented in the following style:



This is a note.

We present Python method names using the following style:

```
pyramid.config.Configurator.add_view()
```

We present Python class names, module names, attributes and global variables using the following style:

```
pyramid.config.Configurator.registry
```

References to glossary terms are presented using the following style:

Pylons

URLs are presented using the following style:

Pylons

References to sections and chapters are presented using the following style:

Traversal

Code and configuration file blocks are presented in the following style:

```
1 def foo(abc) :  
2     pass
```

When a command that should be typed on one line is too long to fit on a page, the backslash \ is used to indicate that the following printed line should actually be part of the command:

```
c:\bigfntut\tutorial> ..\Scripts\nosetests --cover-package=tutorial \  
--cover-erase --with-coverage
```

A sidebar, which presents a concept tangentially related to content discussed on a page, is rendered like so:

This is a sidebar

Sidebar information.

AUTHOR INTRODUCTION

Welcome to “The Pyramid Web Application Framework”. In this introduction, I’ll describe the audience for this book, I’ll describe the book content, I’ll provide some context regarding the genesis of Pyramid, and I’ll thank some important people.

I hope you enjoy both this book and the software it documents. I’ve had a blast writing both.

Audience

This book is aimed primarily at a reader that has the following attributes:

- At least a moderate amount of *Python* experience.
- A familiarity with web protocols such as HTTP and CGI.

If you fit into both of these categories, you’re in the direct target audience for this book. But don’t worry, even if you have no experience with Python or the web, both are easy to pick up “on the fly”.

Python is an *excellent* language in which to write applications; becoming productive in Python is almost mind-blowingly easy. If you already have experience in another language such as Java, Visual Basic, Perl, Ruby, or even C/C++, learning Python will be a snap; it should take you no longer than a couple of days to become modestly productive. If you don’t have previous programming experience, it will be slightly harder, and it will take a little longer, but you’d be hard-pressed to find a better “first language.”

Web technology familiarity is assumed in various places within the book. For example, the book doesn’t try to define common web-related concepts like “URL” or “query string.” Likewise, the book describes various interactions in terms of the HTTP protocol, but it does not describe how the HTTP protocol works in detail. Like any good web framework, though, Pyramid shields you from needing to know most of the gory details of web protocols and low-level data structures. As a result, you can usually avoid becoming “blocked” while you read this book even if you don’t yet deeply understand web technologies.

Book Content

This book is divided into three major parts:

Narrative Documentation

This is documentation which describes Pyramid concepts in narrative form, written in a largely conversational tone. Each narrative documentation chapter describes an isolated Pyramid concept. You should be able to get useful information out of the narrative chapters if you read them out-of-order, or when you need only a reminder about a particular topic while you're developing an application.

Tutorials

Each tutorial builds a sample application or implements a set of concepts with a sample; it then describes the application or concepts in terms of the sample. You should read the tutorials if you want a guided tour of Pyramid.

API Reference

Comprehensive reference material for every public API exposed by Pyramid. The API documentation is organized alphabetically by module name.

The Genesis of `repoze.bfg`

Before the end of 2010, Pyramid was known as `repoze.bfg`.

I wrote `repoze.bfg` after many years of writing applications using *Zope*. Zope provided me with a lot of mileage: it wasn't until almost a decade of successfully creating applications using it that I decided to write a different web framework. Although `repoze.bfg` takes inspiration from a variety of web frameworks, it owes more of its core design to Zope than any other.

The Repoze “brand” existed before `repoze.bfg` was created. One of the first packages developed as part of the Repoze brand was a package named `repoze.zope2`. This was a package that allowed Zope 2 applications to run under a *WSGI* server without modification. Zope 2 did not have reasonable *WSGI* support at the time.

During the development of the `repoze.zope2` package, I found that replicating the Zope 2 “publisher” – the machinery that maps URLs to code – was time-consuming and fiddly. Zope 2 had evolved over many years, and emulating all of its edge cases was extremely difficult. I finished the `repoze.zope2`

package, and it emulates the normal Zope 2 publisher pretty well. But during its development, it became clear that Zope 2 had simply begun to exceed my tolerance for complexity, and I began to look around for simpler options.

I considered using the Zope 3 application server machinery, but it turned out that it had become more indirect than the Zope 2 machinery it aimed to replace, which didn't fulfill the goal of simplification. I also considered using Django and Pylons, but neither of those frameworks offer much along the axes of traversal, contextual declarative security, or application extensibility; these were features I had become accustomed to as a Zope developer.

I decided that in the long term, creating a simpler framework that retained features I had become accustomed to when developing Zope applications was a more reasonable idea than continuing to use any Zope publisher or living with the limitations and unfamiliarities of a different framework. The result is what is now Pyramid.

The Genesis of Pyramid

What was `repoze.bfg` has become Pyramid as the result of a coalition built between the *Repoze* and *Pylons* community throughout the year 2010. By merging technology, we're able to reduce duplication of effort, and take advantage of more of each others' technology.

Thanks

This book is dedicated to my grandmother, who gave me my first typewriter (a Royal), and my mother, who bought me my first computer (a VIC-20).

Thanks to the following people for providing expertise, resources, and software. Without the help of these folks, neither this book nor the software which it details would exist: Paul Everitt, Tres Seaver, Andrew Sawyers, Malthe Borch, Carlos de la Guardia, Chris Rossi, Shane Hathaway, Daniel Holth, Wichert Akkerman, Georg Brandl, Blaise Laflamme, Ben Bangert, Casey Duncan, Mike Orr, John Shipman, Simon Oram and Nat Hardwick of Electrosoup, Ian Bicking of the Open Planning Project, Jim Fulton of Zope Corporation, Tom Moroz of the Open Society Institute, and Todd Koym of Environmental Health Sciences.

Thanks to Guido van Rossum and Tim Peters for Python.

Special thanks to Tricia for putting up with me.

Part I

Narrative Documentation

PYRAMID INTRODUCTION

Pyramid is a general, open source, Python web application development *framework*. Its primary goal is to make it easier for a developer to create web applications. The type of application being created could be a spreadsheet, a corporate intranet, or a social networking platform; Pyramid’s generality enables it to be used to build an unconstrained variety of web applications.

Frameworks vs. Libraries

A *framework* differs from a *library* in one very important way: library code is always *called* by code that you write, while a framework always *calls* code that you write. Using a set of libraries to create an application is usually easier than using a framework initially, because you can choose to cede control to library code you have not authored very selectively. But when you use a framework, you are required to cede a greater portion of control to code you have not authored: code that resides in the framework itself. You needn’t use a framework at all to create a web application using Python. A rich set of libraries already exists for the platform. In practice, however, using a framework to create an application is often more practical than rolling your own via a set of libraries if the framework provides a set of facilities that fits your application requirements.

The first release of Pyramid’s predecessor (named `repoze.bfg`) was made in July of 2008. We have worked hard to ensure that Pyramid continues to follow the design and engineering principles that we consider to be the core characteristics of a successful framework:

Simplicity Pyramid takes a “*pay only for what you eat*” approach. This means that you can get results even if you have only a partial understanding of Pyramid. It doesn’t force you to use any particular technology to produce an application, and we try to keep the core set of concepts that you need to understand to a minimum.

Minimalism Pyramid concentrates on providing fast, high-quality solutions to the fundamental problems of creating a web application: the mapping of URLs to code, templating, security and serving static assets. We consider these to be the core activities that are common to nearly all web applications.

Documentation Pyramid’s minimalism means that it is relatively easy for us to maintain extensive and up-to-date documentation. It is our goal that no aspect of Pyramid remains undocumented.

Speed Pyramid is designed to provide noticeably fast execution for common tasks such as templating and simple response generation. Although the “hardware is cheap” mantra may appear to offer a ready solution to speed problems, the limits of this approach become painfully evident when one finds him or herself responsible for managing a great many machines.

Reliability Pyramid is developed conservatively and tested exhaustively. Where Pyramid source code is concerned, our motto is: “If it ain’t tested, it’s broke”. Every release of Pyramid has 100% statement coverage via unit tests.

Openness As with Python, the Pyramid software is distributed under a permissive open source license.

1.1 What Is The Pylons Project?

Pyramid is a member of the collection of software published under the Pylons Project. Pylons software is written by a loose-knit community of contributors. The Pylons Project website includes details about how Pyramid relates to the Pylons Project.

1.2 Pyramid and Other Web Frameworks

Until the end of 2010, Pyramid was known as `repoze.bfg`; it was merged into the Pylons project as Pyramid in November of that year.

Pyramid was inspired by *Zope*, *Pylons* (version 1.0) and *Django*. As a result, Pyramid borrows several concepts and features from each, combining them into a unique web framework.

Many features of Pyramid trace their origins back to *Zope*. Like *Zope* applications, Pyramid applications can be configured via a set of declarative configuration files. Like *Zope* applications, Pyramid applications can be easily extended: if you obey certain constraints, the application you produce can be reused, modified, re-integrated, or extended by third-party developers without forking the original application. The concepts of *traversal* and declarative security in Pyramid were pioneered first in *Zope*.

The Pyramid concept of *URL dispatch* is inspired by the *Routes* system used by *Pylons* version 1.0. Like *Pylons* version 1.0, Pyramid is mostly policy-free. It makes no assertions about which database you should use, and its built-in templating facilities are included only for convenience. In essence, it only supplies a mechanism to map URLs to *view* code, along with a set of conventions for calling those views. You are free to use third-party components that fit your needs in your applications.

The concept of *view* is used by Pyramid mostly as it would be by Django. Pyramid has a documentation culture more like Django's than like Zope's.

Like *Pylons* version 1.0, but unlike *Zope*, a Pyramid application developer may use completely imperative code to perform common framework configuration tasks such as adding a view or a route. In *Zope*, *ZCML* is typically required for similar purposes. In *Grok*, a *Zope*-based web framework, *decorator* objects and class-level declarations are used for this purpose. Pyramid supports *ZCML* and decorator-based configuration, but does not require either. See *Application Configuration* for more information.

Also unlike *Zope* and unlike other “full-stack” frameworks such as *Django*, Pyramid makes no assumptions about which persistence mechanisms you should use to build an application. *Zope* applications are typically reliant on *ZODB*; Pyramid allows you to build *ZODB* applications, but it has no reliance on the *ZODB* software. Likewise, *Django* tends to assume that you want to store your application's data in a relational database. Pyramid makes no such assumption; it allows you to use a relational database but doesn't encourage or discourage the decision.

Other Python web frameworks advertise themselves as members of a class of web frameworks named model-view-controller frameworks. Insofar as this term has been claimed to represent a class of web frameworks, Pyramid also generally fits into this class.

You Say Pyramid is MVC, But Where's The Controller?

The Pyramid authors believe that the MVC pattern just doesn't really fit the web very well. In a Pyramid application, there is a resource tree, which represents the site structure, and views, which tend to present the data stored in the resource tree and a user-defined “domain model”. However, no facility provided by *the framework* actually necessarily maps to the concept of a “controller” or “model”. So if you had to give it some acronym, I guess you'd say Pyramid is actually an “RV” framework rather than an “MVC” framework. “MVC”, however, is close enough as a general classification moniker for purposes of comparison with other web frameworks.

INSTALLING PYRAMID

2.1 Before You Install

You will need Python version 2.4 or better to run Pyramid.

Python Versions

As of this writing, Pyramid has been tested under Python 2.4.6, Python 2.5.4 and Python 2.6.2, and Python 2.7. To ensure backwards compatibility, development of Pyramid is currently done primarily under Python 2.4 and Python 2.5. Pyramid does not run under any version of Python before 2.4, and does not yet run under Python 3.X.

Pyramid is known to run on all popular Unix-like systems such as Linux, MacOS X, and FreeBSD as well as on Windows platforms. It is also known to run on Google's App Engine and *Jython*.

Pyramid installation does not require the compilation of any C code, so you need only a Python interpreter that meets the requirements mentioned.

2.1.1 If You Don't Yet Have A Python Interpreter (UNIX)

If your system doesn't have a Python interpreter, and you're on UNIX, you can either install Python using your operating system's package manager *or* you can install Python from source fairly easily on any UNIX system that has development tools.

2. INSTALLING PYRAMID

Package Manager Method

You can use your system’s “package manager” to install Python. Every system’s package manager is slightly different, but the “flavor” of them is usually the same.

For example, on an Ubuntu Linux system, to use the system package manager to install a Python 2.6 interpreter, use the following command:

```
$ sudo apt-get install python2.6-dev
```

Once these steps are performed, the Python interpreter will usually be invocable via `python2.6` from a shell prompt.

Source Compile Method

It’s useful to use a Python interpreter that *isn’t* the “system” Python interpreter to develop your software. The authors of Pyramid tend not to use the system Python for development purposes; always a self-compiled one. Compiling Python is usually easy, and often the “system” Python is compiled with options that aren’t optimal for web development.

To compile software on your UNIX system, typically you need development tools. Often these can be installed via the package manager. For example, this works to do so on an Ubuntu Linux system:

```
$ sudo apt-get install build-essential
```

On Mac OS X, installing XCode has much the same effect.

Once you’ve got development tools installed on your system, On the same system, to install a Python 2.6 interpreter from *source*, use the following commands:

```
[chrism@vitaminf ~]$ cd ~
[chrism@vitaminf ~]$ mkdir tmp
[chrism@vitaminf ~]$ mkdir opt
[chrism@vitaminf ~]$ cd tmp
[chrism@vitaminf tmp]$ wget \
    http://www.python.org/ftp/python/2.6.4/Python-2.6.4.tgz
[chrism@vitaminf tmp]$ tar xvzf Python-2.6.4.tgz
[chrism@vitaminf tmp]$ cd Python-2.6.4
[chrism@vitaminf Python-2.6.4]$ ./configure \
    --prefix=$HOME/opt/Python-2.6.4
[chrism@vitaminf Python-2.6.4]$ make; make install
```

Once these steps are performed, the Python interpreter will be invocable via `$HOME/opt/Python-2.6.4/bin/python` from a shell prompt.

2.1.2 If You Don't Yet Have A Python Interpreter (Windows)

If your Windows system doesn't have a Python interpreter, you'll need to install it by downloading a Python 2.6-series interpreter executable from python.org's download section (the files labeled "Windows Installer"). Once you've downloaded it, double click on the executable and accept the defaults during the installation process. You may also need to download and install the Python for Windows extensions.



After you install Python on Windows, you may need to add the `C:\Python26` directory to your environment's Path in order to make it possible to invoke Python from a command prompt by typing `python`. To do so, right click `My Computer`, select `Properties` → `Advanced Tab` → `Environment Variables` and add that directory to the end of the Path environment variable.

2.2 Installing Pyramid on a UNIX System

It is best practice to install Pyramid into a "virtual" Python environment in order to obtain isolation from any "system" packages you've got installed in your Python version. This can be done by using the *virtualenv* package. Using a virtualenv will also prevent Pyramid from globally installing versions of packages that are not compatible with your system Python.

To set up a virtualenv in which to install Pyramid, first ensure that *setuptools* is installed. Invoke `import setuptools` within the Python interpreter you'd like to run Pyramid under:

```
[chris@vitaminf pyramid]$ python
Python 2.4.5 (#1, Aug 29 2008, 12:27:37)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import setuptools
```

If running `import setuptools` does not raise an `ImportError`, it means that *setuptools* is already installed into your Python interpreter. If `import setuptools` fails, you will need to install *setuptools* manually. Note that above we're using a Python 2.4-series interpreter on Mac OS X; your output may differ if you're using a later Python version or a different platform.

If you are using a "system" Python (one installed by your OS distributor or a 3rd-party packager such as Fink or MacPorts), you can usually install the *setuptools* package by using your system's package manager. If you cannot do this, or if you're using a self-installed version of Python, you will need to install *setuptools* "by hand". Installing *setuptools* "by hand" is always a reasonable thing to do, even if your package manager already has a pre-chewed version of *setuptools* for installation.

To install *setuptools* by hand, first download `ez_setup.py` then invoke it using the Python interpreter into which you want to install *setuptools*.

2. INSTALLING PYRAMID

```
$ python ez_setup.py
```

Once this command is invoked, `setuptools` should be installed on your system. If the command fails due to permission errors, you may need to be the administrative user on your system to successfully invoke the script. To remediate this, you may need to do:

```
$ sudo python ez_setup.py
```

2.2.1 Installing the `virtualenv` Package

Once you've got `setuptools` installed, you should install the `virtualenv` package. To install the `virtualenv` package into your `setuptools`-enabled Python interpreter, use the `easy_install` command.

```
$ easy_install virtualenv
```

This command should succeed, and tell you that the `virtualenv` package is now installed. If it fails due to permission errors, you may need to install it as your system's administrative user. For example:

```
$ sudo easy_install virtualenv
```

2.2.2 Creating the Virtual Python Environment

Once the `virtualenv` package is installed in your Python, you can then create a virtual environment. To do so, invoke the following:

```
$ virtualenv --no-site-packages env
New python executable in env/bin/python
Installing setuptools.....done.
```



Using `--no-site-packages` when generating your `virtualenv` is *very important*. This flag provides the necessary isolation for running the set of packages required by Pyramid. If you do not specify `--no-site-packages`, it's possible that Pyramid will not install properly into the `virtualenv`, or, even if it does, may not run properly, depending on the packages you've already got installed into your Python's "main" site-packages dir.



If you're on UNIX, *do not* use `sudo` to run the `virtualenv` script. It's perfectly acceptable (and desirable) to create a `virtualenv` as a normal user.

You should perform any following commands that mention a “bin” directory from within the `env virtualenv` dir.

2.2.3 Installing Pyramid Into the Virtual Python Environment

After you've got your `env virtualenv` installed, you may install Pyramid itself using the following commands from within the `virtualenv (env)` directory:

```
$ bin/easy_install pyramid
```

This command will take longer than the previous ones to complete, as it downloads and installs a number of dependencies.

2.3 Installing Pyramid on a Windows System

1. Install, or find Python 2.6 for your system.
2. Install the Python for Windows extensions. Make sure to pick the right download for Python 2.6 and install it using the same Python installation from the previous step.
3. Install latest *setuptools* distribution into the Python you obtained/installed/found in the step above: download `ez_setup.py` and run it using the `python` interpreter of your Python 2.6 installation using a command prompt:

```
c:\> c:\Python26\python ez_setup.py
```

4. Use that Python's `bin/easy_install` to install `virtualenv`:

```
c:\> c:\Python26\Scripts\easy_install virtualenv
```

5. Use that Python's `virtualenv` to make a workspace:

2. INSTALLING PYRAMID

```
c:\> c:\Python26\Scripts\virtualenv --no-site-packages env
```

6. Switch to the `env` directory:

```
c:\> cd env
```

7. (Optional) Consider using `Scripts\activate.bat` to make your shell environment wired to use the `virtualenv`.
8. Use `easy_install` pointed at the “current” index to get Pyramid and its direct dependencies installed:

```
c:\env> Scripts\easy_install pyramid
```

2.4 Installing Pyramid on Google App Engine

Running Pyramid on Google’s App Engine documents the steps required to install a Pyramid application on Google App Engine.

2.5 Installing Pyramid on Jython

Pyramid is known to work under *Jython* version 2.5.1. Install *Jython*, and then follow the installation steps for Pyramid on your platform described in one of the sections entitled *Installing Pyramid on a UNIX System* or *Installing Pyramid on a Windows System* above, replacing the `python` command with `jython` as necessary. The steps are exactly the same except you should use the `jython` command name instead of the `python` command name.

One caveat exists to using Pyramid under Jython: the *Chameleon* templating engine does not work on Jython. However, the *Mako* templating system, which is also included with Pyramid, does work under Jython; use it instead.

2.6 What Gets Installed

When you `easy_install` Pyramid, various Zope libraries, various Chameleon libraries, WebOb, Paste, PasteScript, and PasteDeploy libraries are installed.

Additionally, as chronicled in *Creating a Pyramid Project*, PasteScript (aka *paster*) templates will be registered that make it easy to start a new Pyramid project.

APPLICATION CONFIGURATION

Each deployment of an application written using Pyramid implies a specific *configuration* of the framework itself. For example, an application which serves up MP3 files for your listening enjoyment might plug code into the framework that manages song files, while an application that manages corporate data might plug in code that manages accounting information. The way in which code is plugged in to Pyramid for a specific application is referred to as “configuration”.

Most people understand “configuration” as coarse settings that inform the high-level operation of a specific application deployment. For instance, it’s easy to think of the values implied by a `.ini` file parsed at application startup time as “configuration”. Pyramid extends this pattern to application development, using the term “configuration” to express standardized ways that code gets plugged into a deployment of the framework itself. When you plug code into the Pyramid framework, you are “configuring” Pyramid to create a particular application.

3.1 Imperative Configuration

Here’s one of the simplest Pyramid applications, configured imperatively:

```
1 from paste.httpserver import serve
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 if __name__ == '__main__':
```

3. APPLICATION CONFIGURATION

```
9 config = Configurator()
10 config.add_view(hello_world)
11 app = config.make_wsgi_app()
12 serve(app, host='0.0.0.0')
```

We won't talk much about what this application does yet. Just note that the “configuration” statements take place underneath the `if __name__ == '__main__':` stanza in the form of method calls on a *Configurator* object (e.g. `config.add_view(...)`). These statements take place one after the other, and are executed in order, so the full power of Python, including conditionals, can be employed in this mode of configuration.

3.2 Configuration Decorations and Code Scanning

A different mode of configuration gives more *locality of reference* to a *configuration declaration*. It's sometimes painful to have all configuration done in imperative code, because often the code for a single application may live in many files. If the configuration is centralized in one place, you'll need to have at least two files open at once to see the “big picture”: the file that represents the configuration, and the file that contains the implementation objects referenced by the configuration. To avoid this, Pyramid allows you to insert *configuration decoration* statements very close to code that is referred to by the declaration itself. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(name='hello', request_method='GET')
5 def hello(request):
6     return Response('Hello')
```

The mere existence of configuration decoration doesn't cause any configuration registration to be performed. Before it has any effect on the configuration of a Pyramid application, a configuration decoration within application code must be found through a process known as a *scan*.

For example, the `pyramid.view.view_config` decorator in the code example above adds an attribute to the `hello` function, making it available for a *scan* to find it later.

A *scan* of a *module* or a *package* and its subpackages for decorations happens when the `pyramid.config.Configurator.scan()` method is invoked: scanning implies searching for configuration declarations in a package and its subpackages. For example:

Starting A Scan

```

1 from paste.httpserver import serve
2 from pyramid.response import Response
3 from pyramid.view import view_config
4
5 @view_config()
6 def hello(request):
7     return Response('Hello')
8
9 if __name__ == '__main__':
10    from pyramid.config import Configurator
11    config = Configurator()
12    config.scan()
13    app = config.make_wsgi_app()
14    serve(app, host='0.0.0.0')
```

The scanning machinery imports each module and subpackage in a package or module recursively, looking for special attributes attached to objects defined within a module. These special attributes are typically attached to code via the use of a *decorator*. For example, the `view_config` decorator can be attached to a function or instance method.

Once scanning is invoked, and *configuration decoration* is found by the scanner, a set of calls are made to a *Configurator* on your behalf: these calls replace the need to add imperative configuration statements that don't live near the code being configured.

In the example above, the scanner translates the arguments to `view_config` into a call to the `pyramid.config.Configurator.add_view()` method, effectively:

```
1 config.add_view(hello)
```

3.3 Declarative Configuration

A third mode of configuration can be employed when you create a Pyramid application named *declarative configuration*. This mode uses an XML language known as *ZCML* to represent configuration statements rather than Python. *ZCML* is not built-in to Pyramid, but almost everything that can be configured imperatively can also be configured via *ZCML* if you install the *pyramid_zcml* package.

CREATING YOUR FIRST PYRAMID APPLICATION

In this chapter, we will walk through the creation of a tiny Pyramid application. After we're finished creating the application, we'll explain in more detail how it works.

4.1 Hello World, Goodbye World

Here's one of the very simplest Pyramid applications, configured imperatively:

```
1 from pyramid.config import Configurator
2 from pyramid.response import Response
3 from paste.httpserver import serve
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 def goodbye_world(request):
9     return Response('Goodbye world!')
10
11 if __name__ == '__main__':
12     config = Configurator()
13     config.add_view(hello_world)
14     config.add_view(goodbye_world, name='goodbye')
15     app = config.make_wsgi_app()
16     serve(app, host='0.0.0.0')
```

4. CREATING YOUR FIRST PYRAMID APPLICATION

When this code is inserted into a Python script named `helloworld.py` and executed by a Python interpreter which has the Pyramid software installed, an HTTP server is started on TCP port 8080:

```
$ python helloworld.py
serving on 0.0.0.0:8080 view at http://127.0.0.1:8080
```

When port 8080 is visited by a browser on the root URL (`/`), the server will simply serve up the text “Hello world!” When visited by a browser on the URL `/goodbye`, the server will serve up the text “Goodbye world!”

Press `Ctrl-C` to stop the application.

Now that we have a rudimentary understanding of what the application does, let’s examine it piece-by-piece.

4.1.1 Imports

The above `helloworld.py` script uses the following set of import statements:

```
1 from pyramid.config import Configurator
2 from pyramid.response import Response
3 from paste.httpserver import serve
```

The script imports the `Configurator` class from the `pyramid.config` module. An instance of the `Configurator` class is later used to configure your Pyramid application.

The script uses the `pyramid.response.Response` class later in the script to create a *response* object.

Like many other Python web frameworks, Pyramid uses the *WSGI* protocol to connect an application and a web server together. The `paste.httpserver` server is used in this example as a WSGI server for convenience, as the `paste` package is a dependency of Pyramid itself.

4.1.2 View Callable Declarations

The above script, beneath its set of imports, defines two functions: one named `hello_world` and one named `goodbye_world`.

```
1 def hello_world(request):
2     return Response('Hello world!')
3
4 def goodbye_world(request):
5     return Response('Goodbye world!')
```

These functions don't do anything very difficult. Both functions accept a single argument (*request*). The `hello_world` function does nothing but return a response instance with the body `Hello world!`. The `goodbye_world` function returns a response instance with the body `Goodbye world!`.

Each of these functions is known as a *view callable*. A view callable accepts a single argument, *request*. It is expected to return a *response* object. A view callable doesn't need to be a function; it can be represented via another type of object, like a class or an instance, but for our purposes here, a function serves us well.

A view callable is always called with a *request* object. A request object is a representation of an HTTP request sent to Pyramid via the active *WSGI* server.

A view callable is required to return a *response* object because a response object has all the information necessary to formulate an actual HTTP response; this object is then converted to text by the upstream *WSGI* server and sent back to the requesting browser. To return a response, each view callable creates an instance of the `Response` class. In the `hello_world` function, the string `'Hello world!'` is passed to the `Response` constructor as the *body* of the response. In the `goodbye_world` function, the string `'Goodbye world!'` is passed.



As we'll see in later chapters, returning a literal *response* object from a view callable is not always required; we can instead use a *renderer* in our view configurations. If we use a renderer, our view callable is allowed to return a value that the renderer understands, and the renderer generates a response on our behalf.

4.1.3 Application Configuration

In the above script, the following code represents the *configuration* of this simple application. The application is configured using the previously defined imports and function definitions, placed within the confines of an `if` statement:

4. CREATING YOUR FIRST PYRAMID APPLICATION

```
1 if __name__ == '__main__':
2     config = Configurator()
3     config.add_view(hello_world)
4     config.add_view(goodbye_world, name='goodbye')
5     app = config.make_wsgi_app()
6     serve(app, host='0.0.0.0')
```

Let's break this down this piece-by-piece.

4.1.4 Configurator Construction

```
1 if __name__ == '__main__':
2     config = Configurator()
```

The `if __name__ == '__main__':` line in the code sample above represents a Python idiom: the code inside this `if` clause is not invoked unless the script containing this code is run directly from the command line. For example, if the file named `helloworld.py` contains the entire script body, the code within the `if` statement will only be invoked when `python helloworld.py` is executed from the operating system command line.

`helloworld.py` in this case is a Python *module*. Using the `if` clause is necessary – or at least best practice – because code in any Python module may be imported by another Python module. By using this idiom, the script is indicating that it does not want the code within the `if` statement to execute if this module is imported; the code within the `if` block should only be run during a direct script execution.

The `config = Configurator()` line above creates an instance of the `Configurator` class. The resulting `config` object represents an API which the script uses to configure this particular Pyramid application. Methods called on the `Configurator` will cause registrations to be made in a *application registry* associated with the application.

4.1.5 Adding Configuration

```
1 config.add_view(hello_world)
2 config.add_view(goodbye_world, name='goodbye')
```

Each of these lines calls the `pyramid.config.Configurator.add_view()` method. The `add_view` method of a configurator registers a *view configuration* within the *application registry*. A *view configuration* represents a set of circumstances related to the *request* that will cause a specific *view callable* to be invoked. This “set of circumstances” is provided as one or more keyword arguments to the `add_view` method. Each of these keyword arguments is known as a view configuration *predicate*.

The line `config.add_view(hello_world)` registers the `hello_world` function as a view callable. The `add_view` method of a `Configurator` must be called with a view callable object or a *dotted Python name* as its first argument, so the first argument passed is the `hello_world` function. This line calls `add_view` with a *default* value for the *predicate* argument, named `name`. The `name` predicate defaults to a value equalling the empty string (`''`). This means that we’re instructing Pyramid to invoke the `hello_world` view callable when the *view name* is the empty string. We’ll learn in later chapters what a *view name* is, and under which circumstances a request will have a view name that is the empty string; in this particular application, it means that the `hello_world` view callable will be invoked when the root URL `/` is visited by a browser.

The line `config.add_view(goodbye_world, name='goodbye')` registers the `goodbye_world` function as a view callable. The line calls `add_view` with the view callable as the first required positional argument, and a *predicate* keyword argument `name` with the value `'goodbye'`. The `name` argument supplied in this *view configuration* implies that only a request that has a *view name* of `goodbye` should cause the `goodbye_world` view callable to be invoked. In this particular application, this means that the `goodbye_world` view callable will be invoked when the URL `/goodbye` is visited by a browser.

Each invocation of the `add_view` method registers a *view configuration*. Each *predicate* provided as a keyword argument to the `add_view` method narrows the set of circumstances which would cause the view configuration’s callable to be invoked. In general, a greater number of predicates supplied along with a view configuration will more strictly limit the applicability of its associated view callable. When Pyramid processes a request, the view callable with the *most specific* view configuration (the view configuration that matches the most specific set of predicates) is always invoked.

In this application, Pyramid chooses the most specific view callable based only on view *predicate* applicability. The ordering of calls to `add_view()` is never very important. We can register `goodbye_world` first and `hello_world` second; Pyramid will still give us the most specific callable when a request is dispatched to it.

4.1.6 WSGI Application Creation

```
1 app = config.make_wsgi_app()
```

4. CREATING YOUR FIRST PYRAMID APPLICATION

After configuring views and ending configuration, the script creates a WSGI *application* via the `pyramid.config.Configurator.make_wsgi_app()` method. A call to `make_wsgi_app` implies that all configuration is finished (meaning all method calls to the configurator which set up views, and various other configuration settings have been performed). The `make_wsgi_app` method returns a WSGI application object that can be used by any WSGI server to present an application to a requestor. WSGI is a protocol that allows servers to talk to Python applications. We don't discuss WSGI in any depth within this book, however, you can learn more about it by visiting wsgi.org.

The Pyramid application object, in particular, is an instance of a class representing a Pyramid *router*. It has a reference to the *application registry* which resulted from method calls to the configurator used to configure it. The *router* consults the registry to obey the policy choices made by a single application. These policy choices were informed by method calls to the *Configurator* made earlier; in our case, the only policy choices made were implied by two calls to its `add_view` method.

4.1.7 WSGI Application Serving

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

Finally, we actually serve the application to requestors by starting up a WSGI server. We happen to use the `paste.httpserver.serve()` WSGI server runner, passing it the `app` object (a *router*) as the application we wish to serve. We also pass in an argument `host=='0.0.0.0'`, meaning “listen on all TCP interfaces.” By default, the Paste HTTP server listens only on the `127.0.0.1` interface, which is problematic if you're running the server on a remote system and you wish to access it with a web browser from a local system. We don't specify a TCP port number to listen on; this means we want to use the default TCP port, which is 8080.

When this line is invoked, it causes the server to start listening on TCP port 8080. It will serve requests forever, or at least until we stop it by killing the process which runs it (usually by pressing `Ctrl-C` in the terminal we used to start it).

4.1.8 Conclusion

Our hello world application is one of the simplest possible Pyramid applications, configured “imperatively”. We can see that it's configured imperatively because the full power of Python is available to us as we perform configuration tasks.

4.2 References

For more information about the API of a *Configurator* object, see `Configurator` .

For more information about *view configuration*, see *View Configuration*.

An example of using *declarative* configuration (*ZCML*) instead of imperative configuration to create a similar “hello world” is available within the documentation for *pyramid_zcml*.

CREATING A PYRAMID PROJECT

As we saw in *Creating Your First Pyramid Application*, it's possible to create a Pyramid application completely manually. However, it's usually more convenient to use a *template* to generate a basic Pyramid *project*.

A project is a directory that contains at least one Python *package*. You'll use a template to create a project, and you'll create your application logic within a package that lives inside the project. Even if your application is extremely simple, it is useful to place code that drives the application within a package, because a package is more easily extended with new code. An application that lives inside a package can also be distributed more easily than one which does not live within a package.

Pyramid comes with a variety of templates that you can use to generate a project. Each template makes different configuration assumptions about what type of application you're trying to construct.

These templates are rendered using the *PasteDeploy* `paster` script, and so therefore they are often referred to as "paster templates".

5.1 Paster Templates Included with Pyramid

The convenience `paster` templates included with Pyramid differ from each other on a number of axes:

- the persistence mechanism they offer (no persistence mechanism, *ZODB*, or *SQLAlchemy*).
- the mechanism they use to map URLs to code (*traversal* or *URL dispatch*).

5. CREATING A PYRAMID PROJECT

- whether or not the `pyramid_beaker` library is relied upon as the sessioning implementation (as opposed to no sessioning or default sessioning).

The included templates are these:

pyramid_starter URL mapping via *traversal* and no persistence mechanism.

pyramid_zodb URL mapping via *traversal* and persistence via *ZODB*.

pyramid_routesalchemy URL mapping via *URL dispatch* and persistence via *SQLAlchemy*

pyramid_alchemy URL mapping via *traversal* and persistence via *SQLAlchemy*

i At this time, each of these templates uses the *Chameleon* templating system, which is incompatible with both Jython and PyPy. To use paster templates to build applications which will run on Jython and PyPy, you can try the `pyramid_jinja2_starter` template which ships as part of the *pyramid_jinja2* package or the `pyramid_sqla` paster template which ships with the *pyramid_sqla* package (it uses Mako), both available from *PyPI*. You can also just use the above paster templates to build a skeleton and replace the Chameleon template it includes with a *Mako* analogue.

Rather than use any of the above templates, Pylons 1 users may feel more comfortable installing the *pyramid_sqla* add-on package, which provides a paster template named `pyramid_sqla`. This paster template configures a Pyramid application in a “Pylons-esque” way, including the use of a *view handler* to map URLs to code (it’s much like a Pylons “controller”).

5.2 Creating the Project

In *Installing Pyramid*, you created a virtual Python environment via the `virtualenv` command. To start a Pyramid *project*, use the `paster` facility installed within the `virtualenv`. In *Installing Pyramid* we called the `virtualenv` directory `env`; the following command assumes that our current working directory is that directory.

We’ll choose the `pyramid_starter` template for this purpose.

```
$ bin/paster create -t pyramid_starter
```


The above command uses the `paster` command to create a project using the `pyramid_starter` template. The `paster create` command creates project from a template. To use a different template, such as `pyramid_routesalchemy`, you’d just change the last argument. For example:


```
$ bin/paster create -t pyramid_routesalchemy
```

`paster create` will ask you a single question: the *name* of the project. You should use a string without spaces and with only letters in it. Here's sample output from a run of `paster create` for a project we name `MyProject`:

```
$ bin/paster create -t pyramid_starter
Selected and implied templates:
  pyramid#pyramid_starter  pyramid starter project

Enter project name: MyProject
Variables:
  egg:      MyProject
  package: myproject
  project:  MyProject
Creating template pyramid
Creating directory ./MyProject
# ... more output ...
Running /Users/chris/projects/pyramid/bin/python setup.py egg_info
```

 You can skip the interrogative question about a project name during `paster create` by adding the project name to the command line, e.g. `paster create -t pyramid_starter MyProject`.

 You may encounter an error when using `paster create` if a dependent Python package is not installed. This will result in a traceback ending in `pkg_resources.DistributionNotFound: <package name>`. Simply run `bin/easy_install`, with the missing package name from the error message to work around this issue.

As a result of invoking the `paster create` command, a project is created in a directory named `MyProject`. That directory is a *project* directory. The `setup.py` file in that directory can be used to distribute your application, or install your application for deployment or development.

A `PasteDeploy .ini` file named `development.ini` will be created in the project directory. You will use this `.ini` file to configure a server, to run your application, and to and debug your application. It sports configuration that enables an interactive debugger and settings optimized for development.

5. CREATING A PYRAMID PROJECT

Another *PasteDeploy* `.ini` file named `production.ini` will also be created in the project directory. It sports configuration that disables any interactive debugger (to prevent inappropriate access and disclosure), and turns off a number of debugging settings. You can use this file to put your application into production, and you can modify it to do things like send email when an exception occurs.

The `MyProject` project directory contains an additional subdirectory named `myproject` (note the case difference) representing a Python *package* which holds very simple Pyramid sample code. This is where you'll edit your application's Python code and templates.

5.3 Installing your Newly Created Project for Development

To install a newly created project for development, you should `cd` to the newly created project directory and use the Python interpreter from the *virtualenv* you created during *Installing Pyramid* to invoke the command `python setup.py develop`

The file named `setup.py` will be in the root of the paster-generated project directory. The `python` you're invoking should be the one that lives in the `bin` directory of your virtual Python environment. Your terminal's current working directory *must* be the newly created project directory. For example:

```
$ ../bin/python setup.py develop
```

Elided output from a run of this command is shown below:

```
$ ../bin/python setup.py develop
...
Finished processing dependencies for MyProject==0.0
```

This will install a *distribution* representing your project into the interpreter's library set so it can be found by `import` statements and by *PasteDeploy* commands such as `paster serve` and `paster pshell`.

5.4 Running The Tests For Your Application

To run unit tests for your application, you should invoke them using the Python interpreter from the *virtualenv* you created during *Installing Pyramid* (the `python` command that lives in the `bin` directory of your *virtualenv*):

```
$ ../bin/python setup.py test -q
```

Here's sample output from a test run:

```
$ python setup.py test -q
running test
running egg_info
writing requirements to MyProject.egg-info/requires.txt
writing MyProject.egg-info/PKG-INFO
writing top-level names to MyProject.egg-info/top_level.txt
writing dependency_links to MyProject.egg-info/dependency_links.txt
writing entry points to MyProject.egg-info/entry_points.txt
reading manifest file 'MyProject.egg-info/SOURCES.txt'
writing manifest file 'MyProject.egg-info/SOURCES.txt'
running build_ext
..
-----
Ran 1 test in 0.108s

OK
```

i The `-q` option is passed to the `setup.py test` command to limit the output to a stream of dots. If you don't pass `-q`, you'll see more verbose test result output (which normally isn't very useful).

The tests themselves are found in the `tests.py` module in your `paster create`-generated project. Within a project generated by the `pyramid_starter` template, a single sample test exists.

5.5 The Interactive Shell

Once you've installed your program for development using `setup.py develop`, you can use an interactive Python shell to examine your Pyramid project's *resource* and *view* objects from a Python prompt. To do so, use your `virtualenv`'s `paster pshell` command.

The first argument to `pshell` is the path to your application's `.ini` file. The second is the `app` section name inside the `.ini` file which points to *your application* as opposed to any other section within the `.ini` file. For example, if your application `.ini` file might have a `[app:MyProject]` section that looks like so:

5. CREATING A PYRAMID PROJECT

```
1 [app:MyProject]
2 use = egg:MyProject
3 reload_templates = true
4 debug_authorization = false
5 debug_notfound = false
6 debug_templates = true
7 default_locale_name = en
```

If so, you can use the following command to invoke a debug shell using the name `MyProject` as a section name:

```
[chris@vitaminf shellenv]$ ../bin/paster pshell development.ini MyProject
Python 2.4.5 (#1, Aug 29 2008, 12:27:37)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help" for more information. "root" is the Pyramid app root object,
"registry" is the Pyramid registry object.
>>> root
<myproject.resources.MyResource object at 0x445270>
>>> registry
<Registry myproject>
>>> registry.settings['debug_notfound']
False
>>> from myproject.views import my_view
>>> from pyramid.request import Request
>>> r = Request.blank('/')
>>> my_view(r)
{'project': 'myproject'}
```

Two names are made available to the pshell user as globals: `root` and `registry`. `root` is the the object returned by the default *root factory* in your application. `registry` is the *application registry* object associated with your project's application (often accessed within view code as `request.registry`).

If you have IPython installed in the interpreter you use to invoke the `paster` command, the `pshell` command will use an IPython interactive shell instead of a standard Python interpreter shell. If you don't want this to happen, even if you have IPython installed, you can pass the `--disable-ipython` flag to the `pshell` command to use a standard Python interpreter shell unconditionally.

```
[chris@vitaminf shellenv]$ ../bin/paster pshell --disable-ipython \
development.ini MyProject
```

You should always use a section name argument that refers to the actual app section within the Paste configuration file that points at your Pyramid application *without any middleware wrapping*. In particular, a section name is inappropriate as the second argument to `pshell` if the configuration section it names is a pipeline rather than an app. For example, if you have the following `.ini` file content:

```
1 [app:MyProject]
2 use = egg:MyProject
3 reload_templates = true
4 debug_authorization = false
5 debug_notfound = false
6 debug_templates = true
7 default_locale_name = en
8
9 [pipeline:main]
10 pipeline =
11     egg:WebError#evalerror
12     MyProject
```

Use `MyProject` instead of `main` as the section name argument to `pshell` against the above `.ini` file (e.g. `paster pshell development.ini MyProject`). If you use `main` instead, an error will occur. Use the most specific reference to your application within the `.ini` file possible as the section name argument.

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

5.6 Running The Project Application

Once a project is installed for development, you can run the application it represents using the `paster serve` command against the generated configuration file. In our case, this file is named `development.ini`:

```
$ ../bin/paster serve development.ini
```

Here's sample output from a run of `paster serve`:

```
$ ../bin/paster serve development.ini
Starting server in PID 16601.
serving on 0.0.0.0:6543 view at http://127.0.0.1:6543
```

By default, Pyramid applications generated from a `paster` template will listen on TCP port 6543. You can shut down a server started this way by pressing `Ctrl-C`.

During development, it's often useful to run `paster serve` using its `--reload` option. When `--reload` is passed to `paster serve`, changes to any Python module your project uses will cause

5. CREATING A PYRAMID PROJECT

the server to restart. This typically makes development easier, as changes to Python code made within a Pyramid application is not put into effect until the server restarts.

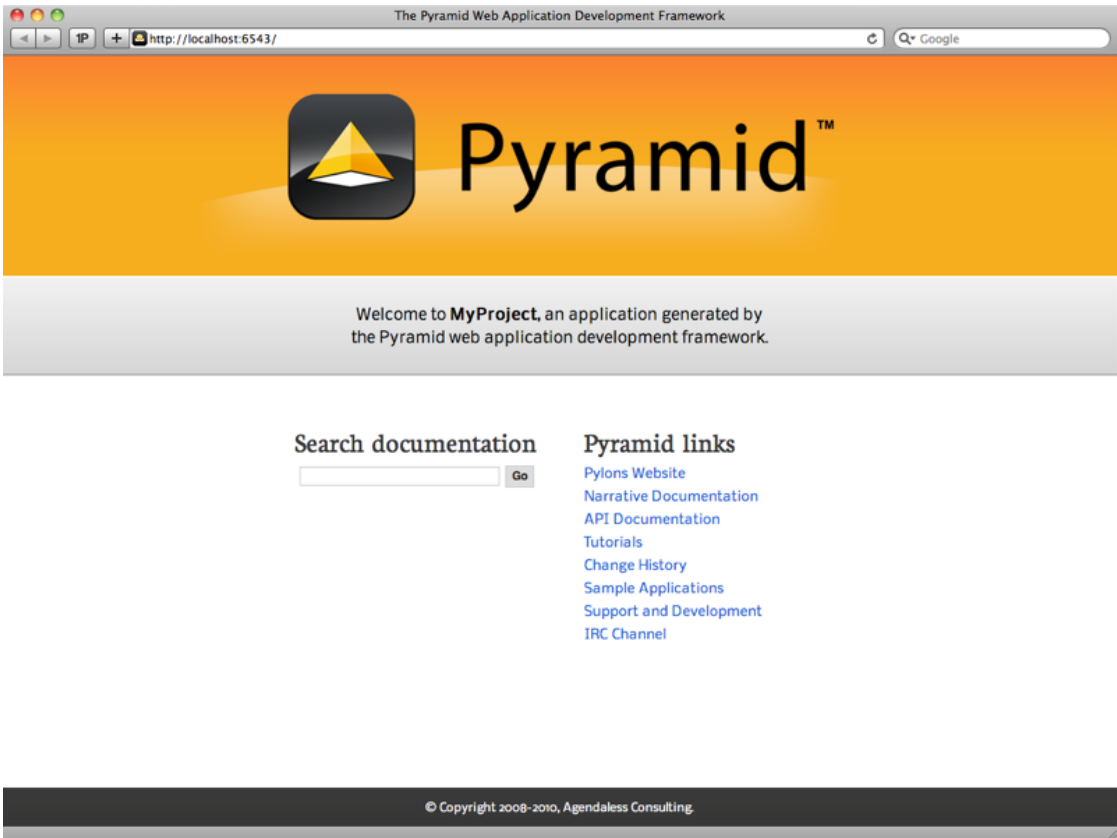
For example:

```
$ ../bin/paster serve development.ini --reload
Starting subprocess with file monitor
Starting server in PID 16601.
serving on 0.0.0.0:6543 view at http://127.0.0.1:6543
```

For more detailed information about the startup process, see *Startup*. For more information about environment variables and configuration file settings that influence startup and runtime behavior, see *Environment Variables and .ini File Settings*.

5.7 Viewing the Application

Once your application is running via `paster serve`, you may visit `http://localhost:6543/` in your browser. You will see something in your browser like what is displayed in the following image:



This is the page shown by default when you visit an unmodified `paste create -generated pyramid_starter` application in a browser.

Using an Alternate WSGI Server

The code generated by Pyramid `paster` templates assumes that you will be using the `paster serve` command to start your application while you do development. However, `paster serve` is by no means the only way to start up and serve a Pyramid application. As we saw in *Creating Your First Pyramid Application*, `paster serve` needn't be invoked at all to run a Pyramid application. The use of `paster serve` to run a Pyramid application is purely conventional based on the output of its `paster` templates.

Any WSGI server is capable of running a Pyramid application. Some WSGI servers don't require the *PasteDeploy* framework's `paster serve` command to do server process management at all. Each WSGI server has its own documentation about how it creates a process to run an application, and there are many of them, so we cannot provide the details for each here. But the concepts are largely the same, whatever server you happen to use.

One popular production alternative to a `paster`-invoked server is `mod_wsgi`. You can also use `mod_wsgi` to serve your Pyramid application using the Apache web server rather than any "pure-Python" server that is started as a result of `paster serve`. See *Running a Pyramid Application under mod_wsgi* for details. However, it is usually easier to *develop* an application using a `paster serve`-invoked webserver, as exception and debugging output will be sent to the console.

5.8 The Project Structure

The `pyramid_starter` template generated a *project* (named `MyProject`), which contains a Python *package*. The package is *also* named `myproject`, but it's lowercased; the `paster` template generates a project which contains a package that shares its name except for case.

All Pyramid `paster`-generated projects share a similar structure. The `MyProject` project we've generated has the following directory structure:

```
MyProject/
|-- CHANGES.txt
|-- development.ini
|-- MANIFEST.in
|-- myproject
|   |-- __init__.py
|   |-- resources.py
|   |-- static
|   |   |-- favicon.ico
|   |   |-- logo.png
|   |   `-- pylons.css
|   |-- templates
|   `-- mytemplate.pt
```

```
| |-- tests.py  
| |-- views.py  
|-- production.ini  
|-- README.txt  
|-- setup.cfg  
|-- setup.py
```

5.9 The MyProject *Project*

The `MyProject project` directory is the distribution and deployment wrapper for your application. It contains both the `myproject package` representing your application as well as files used to describe, run, and test your application.

1. `CHANGES.txt` describes the changes you’ve made to the application. It is conventionally written in *ReStructuredText* format.
2. `README.txt` describes the application in general. It is conventionally written in *ReStructuredText* format.
3. `development.ini` is a *PasteDeploy* configuration file that can be used to execute your application during development.
4. `production.ini` is a *PasteDeploy* configuration file that can be used to execute your application in a production configuration.
5. `setup.cfg` is a *setuptools* configuration file used by `setup.py`.
6. `MANIFEST.in` is a *distutils* “manifest” file, naming which files should be included in a source distribution of the package when `python setup.py sdist` is run.
7. `setup.py` is the file you’ll use to test and distribute your application. It is a standard *setuptools* `setup.py` file.

5.9.1 `development.ini`

The `development.ini` file is a *PasteDeploy* configuration file. Its purpose is to specify an application to run when you invoke `paster serve`, as well as the deployment settings provided to that application.

The generated `development.ini` file looks like so:

5. CREATING A PYRAMID PROJECT

```
1  [app:MyProject]
2  use = egg:MyProject
3  reload_templates = true
4  debug_authorization = false
5  debug_notfound = false
6  debug_routematch = false
7  debug_templates = true
8  default_locale_name = en
9
10 [pipeline:main]
11 pipeline =
12     egg:WebError#evalerror
13     MyProject
14
15 [server:main]
16 use = egg:Paste#http
17 host = 0.0.0.0
18 port = 6543
19
20 # Begin logging configuration
21
22 [loggers]
23 keys = root, myproject
24
25 [handlers]
26 keys = console
27
28 [formatters]
29 keys = generic
30
31 [logger_root]
32 level = INFO
33 handlers = console
34
35 [logger_myproject]
36 level = DEBUG
37 handlers =
38 qualname = myproject
39
40 [handler_console]
41 class = StreamHandler
42 args = (sys.stderr,)
43 level = NOTSET
44 formatter = generic
45
46 [formatter_generic]
```

```

47 format = %(asctime)s %(levelname)-5.5s [% (name) s] %(message) s
48
49 # End logging configuration

```

This file contains several “sections” including `[app:MyProject]`, `[pipeline:main]`, and `[server:main]`.

The `[app:MyProject]` section represents configuration for your application. This section name represents the `MyProject` application (and it’s an app -lication, thus `app:MyProject`)

The `use` setting is required in the `[app:MyProject]` section. The `use` setting points at a *setuptools* entry point named `MyProject` (the `egg:` prefix in `egg:MyProject` indicates that this is an entry point *URI* specifier, where the “scheme” is “egg”). `egg:MyProject` is actually shorthand for a longer spelling: `egg:MyProject#main`. The `#main` part is omitted for brevity, as it is the default.

setuptools Entry Points and PasteDeploy .ini Files

This part of configuration can be confusing so let’s try to clear things up a bit. Take a look at the generated `setup.py` file for this project. Note that the `entry_point` line in `setup.py` points at a string which looks a lot like an `.ini` file. This string representation of an `.ini` file has a section named `[paste.app_factory]`. Within this section, there is a key named `main` (the entry point name) which has a value `myproject:main`. The *key* `main` is what our `egg:MyProject#main` value of the `use` section in our config file is pointing at (although it is actually shortened to `egg:MyProject` there). The value represents a *dotted Python name* path, which refers to a callable in our `myproject` package’s `__init__.py` module. In English, this entry point can thus be referred to as a “Paste application factory in the `MyProject` project which has the entry point named `main` where the entry point refers to a `main` function in the `mypackage` module”. If indeed if you open up the `__init__.py` module generated within the `myproject` package, you’ll see a `main` function. This is the function called by *PasteDeploy* when the `paster serve` command is invoked against our application. It accepts a global configuration object and *returns* an instance of our application.

The `use` setting is the only setting *required* in the `[app:MyProject]` section unless you’ve changed the callable referred to by the `egg:MyProject` entry point to accept more arguments: other settings you add to this section are passed as keywords arguments to the callable represented by this entry point (`main` in our `__init__.py` module). You can provide startup-time configuration parameters to your application by adding more settings to this section.

The `reload_templates` setting in the `[app:MyProject]` section is a Pyramid -specific setting which is passed into the framework. If it exists, and its value is `true`, *Chameleon* and *Mako* template

5. CREATING A PYRAMID PROJECT

changes will not require an application restart to be detected. See *Automatically Reloading Templates* for more information.



The `reload_templates` option should be turned off for production applications, as template rendering is slowed when it is turned on.

The `debug_templates` setting in the `[app:MyProject]` section is a Pyramid -specific setting which is passed into the framework. If it exists, and its value is `true`, *Chameleon* template exceptions will contained more detailed and helpful information about the error than when this value is `false`. See *Nicer Exceptions in Chameleon Templates* for more information.



The `debug_templates` option should be turned off for production applications, as template rendering is slowed when it is turned on.

Various other settings may exist in this section having to do with debugging or influencing runtime behavior of a Pyramid application. See *Environment Variables and .ini File Settings* for more information about these settings.

`[pipeline:main]`, has the name `main` signifying that this is the default ‘application’ (although it’s actually a pipeline of middleware and an application) run by `paste serve` when it is invoked against this configuration file. The name `main` is a convention used by PasteDeploy signifying that it is the default application.

The `[server:main]` section of the configuration file configures a WSGI server which listens on TCP port 6543. It is configured to listen on all interfaces (`0.0.0.0`). The `Paste#http` server will create a new thread for each request.



In general, Pyramid applications generated from `paste` templates should be threading-aware. It is not required that a Pyramid application be nonblocking as all application code will run in its own thread, provided by the server you’re using.

See the *PasteDeploy* documentation for more information about other types of things you can put into this `.ini` file, such as other applications, *middleware* and alternate WSGI server implementations.




You can add a `[DEFAULT]` section to your `development.ini` file. Such a section should consists of global parameters that are shared by all the applications, servers and *middleware* defined within the configuration file. The values in a `[DEFAULT]` section will be passed to your application’s `main` function as `global_values`.

5.9.2 production.ini

The `development.ini` file is a *PasteDeploy* configuration file with a purpose much like that of `development.ini`. However, it disables the `WebError` interactive debugger, replacing it with a logger which outputs exception messages to `stderr` by default. It also turns off template development options such that templates are not automatically reloaded when changed, and turns off all debugging options. You can use this file instead of `development.ini` when you put your application into production.

5.9.3 setup.py

The `setup.py` file is a *setuptools* setup file. It is meant to be run directly from the command line to perform a variety of functions, such as testing your application, packaging, and distributing your application.

 `setup.py` is the defacto standard which Python developers use to distribute their reusable code. You can read more about `setup.py` files and their usage in the *Setuptools* documentation.

Our generated `setup.py` looks like this:

```

1  import os
2
3  from setuptools import setup, find_packages
4
5  here = os.path.abspath(os.path.dirname(__file__))
6  README = open(os.path.join(here, 'README.txt')).read()
7  CHANGES = open(os.path.join(here, 'CHANGES.txt')).read()
8
9  requires = ['pyramid', 'WebError']
10
11  setup(name='MyProject',
12        version='0.0',
13        description='MyProject',
14        long_description=README + '\n\n' + CHANGES,
15        classifiers=[
16            "Programming Language :: Python",
17            "Framework :: Pylons",
18            "Topic :: Internet :: WWW/HTTP",
19            "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
20        ],
21        author='',
22        author_email='',
23        url='',

```

5. CREATING A PYRAMID PROJECT

```
24     keywords='web pyramid pylons',
25     packages=find_packages(),
26     include_package_data=True,
27     zip_safe=False,
28     install_requires=requires,
29     tests_require=requires,
30     test_suite="myproject",
31     entry_points = """\
32     [paste.app_factory]
33     main = myproject:main
34     """,
35     paster_plugins=['pyramid'],
36 )
```

The `setup.py` file calls the `setuptools` `setup` function, which does various things depending on the arguments passed to `setup.py` on the command line.

Within the arguments to this function call, information about your application is kept. While it's beyond the scope of this documentation to explain everything about `setuptools` `setup` files, we'll provide a whirlwind tour of what exists in this file in this section.

Your application's name can be any string; it is specified in the `name` field. The version number is specified in the `version` value. A short description is provided in the `description` field. The `long_description` is conventionally the content of the `README` and `CHANGES` file appended together. The `classifiers` field is a list of Trove classifiers describing your application. `author` and `author_email` are text fields which probably don't need any description. `url` is a field that should point at your application project's URL (if any). `packages=find_packages()` causes all packages within the project to be found when packaging the application. `include_package_data` will include non-Python files when the application is packaged if those files are checked into version control. `zip_safe` indicates that this package is not safe to use as a zipped egg; instead it will always unpack as a directory, which is more convenient. `install_requires` and `tests_require` indicate that this package depends on the `pyramid` package. `test_suite` points at the package for our application, which means all tests found in the package will be run when `setup.py test` is invoked. We examined `entry_points` in our discussion of the `development.ini` file; this file defines the `main` entry point that represents our project's application.

Usually you only need to think about the contents of the `setup.py` file when distributing your application to other people, or when versioning your application for your own use. For fun, you can try this command now:

```
$ python setup.py sdist
```


This will create a tarball of your application in a `dist` subdirectory named `MyProject-0.1.tar.gz`. You can send this tarball to other people who want to use your application.



Without the presence of a `MANIFEST.in` file or without checking your source code into a version control repository, `setup.py sdist` places only *Python source files* (files ending with a `.py` extension) into tarballs generated by `python setup.py sdist`. This means, for example, if your project was not checked into a `setuptools`-compatible source control system, and your project directory didn't contain a `MANIFEST.in` file that told the `sdist` machinery to include `*.pt` files, the `myproject/templates/mytemplate.pt` file would not be included in the generated tarball. Projects generated by Pyramid paster templates include a default `MANIFEST.in` file. The `MANIFEST.in` file contains declarations which tell it to include files like `*.pt`, `*.css` and `*.js` in the generated tarball. If you include files with extensions other than the files named in the project's `MANIFEST.in` and you don't make use of a `setuptools`-compatible version control system, you'll need to edit the `MANIFEST.in` file and include the statements necessary to include your new files. See <http://docs.python.org/distutils/sourcedist.html#principle> for more information about how to do this.

You can also delete `MANIFEST.in` from your project and rely on a `setuptools` feature which simply causes all files checked into a version control system to be put into the generated tarball. To allow this to happen, check all the files that you'd like to be distributed along with your application's Python files into Subversion. After you do this, when you rerun `setup.py sdist`, all files checked into the version control system will be included in the tarball. If you don't use Subversion, and instead use a different version control system, you may need to install a `setuptools` add-on such as `setuptools-git` or `setuptools-hg` for this behavior to work properly.

5.9.4 setup.cfg

The `setup.cfg` file is a *setuptools* configuration file. It contains various settings related to testing and internationalization:

Our generated `setup.cfg` looks like this:

```

1  [nosetests]
2  match = ^test
3  nocapture = 1
4  cover-package = myproject
5  with-coverage = 1
6  cover-erase = 1
7
8  [compile_catalog]
9  directory = myproject/locale

```

```
10 domain = MyProject
11 statistics = true
12
13 [extract_messages]
14 add_comments = TRANSLATORS:
15 output_file = myproject/locale/MyProject.pot
16 width = 80
17
18 [init_catalog]
19 domain = MyProject
20 input_file = myproject/locale/MyProject.pot
21 output_dir = myproject/locale
22
23 [update_catalog]
24 domain = MyProject
25 input_file = myproject/locale/MyProject.pot
26 output_dir = myproject/locale
27 previous = true
```

The values in the default setup file allow various commonly-used internationalization commands and testing commands to work more smoothly.

5.10 The `myproject` Package

The `myproject` package lives inside the `MyProject` project. It contains:

1. An `__init__.py` file signifies that this is a Python *package*. It also contains code that helps users run the application, including a `main` function which is used as a Paste entry point.
2. A `resources.py` module, which contains *resource* code.
3. A `templates` directory, which contains *Chameleon* (or other types of) templates.
4. A `tests.py` module, which contains unit test code for the application.
5. A `views.py` module, which contains view code for the application.

These are purely conventions established by the `paste` template: Pyramid doesn't insist that you name things in any particular way. However, it's generally a good idea to follow Pyramid standards for naming, so that other Pyramid developers can get up to speed quickly on your code when you need help.

5.10.1 `__init__.py`

We need a small Python module that configures our application and which advertises an entry point for use by our *PasteDeploy* `.ini` file. This is the file named `__init__.py`. The presence of an `__init__.py` also informs Python that the directory which contains it is a *package*.

```

1 from pyramid.config import Configurator
2 from myproject.resources import Root
3
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6     """
7     config = Configurator(root_factory=Root, settings=settings)
8     config.add_view('myproject.views.my_view',
9                     context='myproject.resources.Root',
10                    renderer='myproject:templates/mytemplate.pt')
11    config.add_static_view('static', 'myproject:static')
12    return config.make_wsgi_app()

```

1. Line 1 imports the *Configurator* class from `pyramid.config` that we use later.
2. Line 2 imports the *Root* class from `myproject.resources` that we use later.
3. Lines 4-12 define a function that returns a Pyramid WSGI application. This function is meant to be called by the *PasteDeploy* framework as a result of running `paster serve`.

Within this function, application configuration is performed.

Lines 8-10 register a “default view” (a view that has no name attribute). It is registered so that it will be found when the *context* of the request is an instance of the `myproject.resources.Root` class. The first argument to `add_view` points at a Python function that does all the work for this view, also known as a *view callable*, via a *dotted Python name*. The view declaration also names a *renderer*, which in this case is a template that will be used to render the result of the view callable. This particular view declaration points at `myproject:templates/mytemplate.pt`, which is a *asset specification* that specifies the `mytemplate.pt` file within the `templates` directory of the `myproject` package. The template file it actually points to is a *Chameleon ZPT* template file.

Line 11 registers a static view, which will serve up the files from the `mypackage:static` *asset specification* (the `static` directory of the `mypackage` package).

Line 12 returns a *WSGI* application to the caller of the function (Paste).

5.10.2 `views.py`

Much of the heavy lifting in a Pyramid application is done by *view callables*. A *view callable* is the main tool of a Pyramid web application developer; it is a bit of code which accepts a *request* and which returns a *response*.

```
1 def my_view(request):
2     return {'project': 'MyProject' }
```

This bit of code was registered as the view callable within `__init__.py` (via `add_view`). `add_view` said that the default URL for instances that are of the class `myproject.resources.Root` should run this `myproject.views.my_view()` function.

This view callable function is handed a single piece of information: the *request*. The *request* is an instance of the *WebOb* `Request` class representing the browser's request to our server.

This view returns a dictionary. When this view is invoked, a *renderer* converts the dictionary returned by the view into HTML, and returns the result as the *response*. This view is configured to invoke a renderer which uses a *Chameleon* ZPT template (`mypackage:templates/my_template.pt`, as specified in the `__init__.py` file call to `add_view`).

See *Writing View Callables Which Use a Renderer* for more information about how views, renderers, and templates relate and cooperate.



Because our `development.ini` has a `reload_templates = true` directive indicating that templates should be reloaded when they change, you won't need to restart the application server to see changes you make to templates. During development, this is handy. If this directive had been `false` (or if the directive did not exist), you would need to restart the application server for each template change. For production applications, you should set your project's `reload_templates` to `false` to increase the speed at which templates may be rendered.

5.10.3 `resources.py`

The `resources.py` module provides the *resource* data and behavior for our application. Resources are objects which exist to provide site structure in applications which use *traversal* to map URLs to code. We write a class named `Root` that provides the behavior for the root resource.

```
1 class Root(object):
2     def __init__(self, request):
3         self.request = request
```

1. Lines 1-3 define the `Root` class. The `Root` class is a “root resource factory” function that will be called by the Pyramid *Router* for each request when it wants to find the root of the resource tree.

In a “real” application, the `Root` object would likely not be such a simple object. Instead, it might be an object that could access some persistent data store, such as a database. Pyramid doesn’t make any assumption about which sort of data storage you’ll want to use, so the sample application uses an instance of `myproject.resources.Root` to represent the root.

5.10.4 static

This directory contains static assets which support the `mytemplate.pt` template. It includes CSS and images.

5.10.5 templates/mytemplate.pt

The single *Chameleon* template exists in the project. Its contents are too long to show here, but it displays a default page when rendered. It is referenced by the call to `add_view` as the `renderer` attribute in the `__init__` file. See *Writing View Callables Which Use a Renderer* for more information about renderers.

Templates are accessed and used by view configurations and sometimes by view functions themselves. See *Using Templates Directly* and *Templates Used as Renderers via Configuration*.

5.10.6 tests.py

The `tests.py` module includes unit tests for your application.

```
1 import unittest
2
3 from pyramid import testing
4
5 class ViewTests(unittest.TestCase):
6     def setUp(self):
7         self.config = testing.setUp()
8
9     def tearDown(self):
10        testing.tearDown()
11
12    def test_my_view(self):
13        from myproject.views import my_view
14        request = testing.DummyRequest()
15        info = my_view(request)
16        self.assertEqual(info['project'], 'MyProject')
```

This sample `tests.py` file has a single unit test defined within it. This test is executed when you run `python setup.py test`. You may add more tests here as you build your application. You are not required to write tests to use Pyramid, this file is simply provided as convenience and example.

See *Unit, Integration, and Functional Testing* for more information about writing Pyramid unit tests.

5.11 Modifying Package Structure

It is best practice for your application's code layout to not stray too much from accepted Pyramid paster template defaults. If you refrain from changing things very much, other Pyramid coders will be able to more quickly understand your application. However, the code layout choices made for you by a paster template are in no way magical or required. Despite the choices made for you by any paster template, you can decide to lay your code out any way you see fit.

For example, the configuration method named `add_view()` requires you to pass a *dotted Python name* or a direct object reference as the class or function to be used as a view. By default, the `pyramid_starter` paster template would have you add view functions to the `views.py` module in your package. However, you might be more comfortable creating a *views directory*, and adding a single file for each view.

If your project package name was `myproject` and you wanted to arrange all your views in a Python subpackage within the `myproject` package named `views` instead of within a single `views.py` file, you might:

- Create a `views` directory inside your `mypackage` package directory (the same directory which holds `views.py`).
- *Move* the existing `views.py` file to a file inside the new `views` directory named, say, `blog.py`.
- Create a file within the new `views` directory named `__init__.py` (it can be empty, this just tells Python that the `views` directory is a *package*).

Then change the `__init__.py` of your `myproject` project (*not* the `__init__.py` you just created in the `views` directory, the one in its parent directory). For example, from something like:

```
1 config.add_view('myproject.views.my_view',  
2                 renderer='myproject:templates/mytemplate.pt')
```

To this:

```
1 config.add_view('myproject.views.blog.my_view',  
2                 renderer='myproject:templates/mytemplate.pt')
```

You can then continue to add files to the `views` directory, and refer to view classes or functions within those files via the dotted name passed as the first argument to `add_view`. For example, if you added a file named `anothermodule.py` to the `views` subdirectory, and added a view callable named `my_view` to it:

```
1 config.add_view('myproject.views.anothermodule.my_view',  
2                 renderer='myproject:templates/anothertemplate.pt')
```

This pattern can be used to rearrange code referred to by any Pyramid API argument which accepts a *dotted Python name* or direct object reference.

URL DISPATCH

URL dispatch provides a simple way to map URLs to *view* code using a simple pattern matching language. An ordered set of patterns is checked one-by-one. If one of the patterns matches the path information associated with a request, a particular *view callable* is invoked.

URL dispatch is one of two ways to perform *resource location* in Pyramid; the other way is using *traversal*. If no route is matched using *URL dispatch*, Pyramid falls back to *traversal* to handle the *request*.

It is the responsibility of the *resource location* subsystem (i.e., *URL dispatch* or *traversal*) to find the resource object that is the *context* of the *request*. Once the *context* is determined, *view lookup* is then responsible for finding and invoking a *view callable*. A view callable is a specific bit of code, defined in your application, that receives the *request* and returns a *response* object.

Where appropriate, we will describe how view lookup interacts with *resource location*. The *View Configuration* chapter describes the details of *view lookup*.

6.1 High-Level Operational Overview

If route configuration is present in an application, the Pyramid *Router* checks every incoming request against an ordered set of URL matching patterns present in a *route map*.

If any route pattern matches the information in the *request*, Pyramid will invoke *view lookup* using a *context* resource generated by the route match.

However, if no route pattern matches the information in the *request* provided to Pyramid, it will fail over to using *traversal* to perform resource location and view lookup.

Technically, URL dispatch is a *resource location* mechanism (it finds a context object). But ironically, using URL dispatch (instead of *traversal*) allows you to avoid thinking about your application in terms of “resources” entirely, because it allows you to directly map a *view callable* to a route.

6.2 Route Configuration

Route configuration is the act of adding a new *route* to an application. A route has a *pattern*, representing a pattern meant to match against the `PATH_INFO` portion of a URL (the portion following the scheme and port, e.g. `/foo/bar` in the URL `http://localhost:8080/foo/bar`), and a *route name*, which is used by developers within a Pyramid application to uniquely identify a particular route when generating a URL. It also optionally has a *factory*, a set of *route predicate* parameters, and a set of *view* parameters.

6.2.1 Configuring a Route via The `add_route` Configurator Method

The `pyramid.config.Configurator.add_route()` method adds a single *route configuration* to the *application registry*. Here's an example:

```
# "config" below is presumed to be an instance of the
# pyramid.config.Configurator class; "myview" is assumed
# to be a "view callable" function
from views import myview
config.add_route('myroute', '/prefix/{one}/{two}', view=myview)
```

Changed in version 1.0a4: Prior to 1.0a4, routes allow for a marker starting with a `:`, for example `/prefix/:one/:two`. This style is now deprecated in favor of `{}` usage which allows for additional functionality.

6.2.2 Route Configuration That Names a View Callable

When a route configuration declaration names a `view` attribute, the value of the attribute will reference a *view callable*. This view callable will be invoked when the route matches. A view callable, as described in *Views*, is developer-supplied code that “does stuff” as the result of a request.

Here's an example route configuration that references a view callable:

```
1 # "config" below is presumed to be an instance of the
2 # pyramid.config.Configurator class; "myview" is assumed
3 # to be a "view callable" function
4 from myproject.views import myview
5 config.add_route('myroute', '/prefix/{one}/{two}', view=myview)
```

You can also pass a *dotted Python name* as the `view` argument rather than an actual callable:

```

1 # "config" below is presumed to be an instance of the
2 # pyramid.config.Configurator class; "myview" is assumed
3 # to be a "view callable" function
4 config.add_route('myroute', '/prefix/{one}/{two}',
5                  view='myproject.views.myview')
```

When a route configuration names a `view` attribute, the *view callable* named as that `view` attribute will always be found and invoked when the associated route pattern matches during a request.

6.2.3 Route Pattern Syntax

The syntax of the pattern matching language used by Pyramid URL dispatch in the *pattern* argument is straightforward; it is close to that of the *Routes* system used by *Pylons*.

The *pattern* used in route configuration may start with a slash character. If the pattern does not start with a slash character, an implicit slash will be prepended to it at matching time. For example, the following patterns are equivalent:

```
{foo}/bar/baz
```

and:

```
/ {foo}/bar/baz
```

A pattern segment (an individual item between `/` characters in the pattern) may either be a literal string (e.g. `foo`) or it may be a replacement marker (e.g. `{foo}`) or a certain combination of both. A replacement marker does not need to be preceded by a `/` character.

A replacement marker is in the format `{name}`, where this means “accept any characters up to the next slash character and use this as the *name* *matchdict* value.” A *matchdict* is the dictionary representing the dynamic parts extracted from a URL based on the routing pattern. It is available as `request.matchdict`. For example, the following pattern defines one literal segment (`foo`) and two replacement markers (`baz`, and `bar`):

```
foo/{baz}/{bar}
```

The above pattern will match these URLs, generating the following *matchdicts*:

6. URL DISPATCH

```
foo/1/2      -> {'baz':u'1', 'bar':u'2'}
foo/abc/def  -> {'baz':u'abc', 'bar':u'def'}
```

It will not match the following patterns however:

```
foo/1/2/    -> No match (trailing slash)
bar/abc/def -> First segment literal mismatch
```

The match for a segment replacement marker in a segment will be done only up to the first non-alphanumeric character in the segment in the pattern. So, for instance, if this route pattern was used:

```
foo/{name}.html
```

The literal path `/foo/biz.html` will match the above route pattern, and the match result will be `{'name':u'biz'}`. However, the literal path `/foo/biz` will not match, because it does not contain a literal `.html` at the end of the segment represented by `{name}.html` (it only contains `biz`, not `biz.html`).

To capture both segments, two replacement markers can be used:

```
foo/{name}.{ext}
```

The literal path `/foo/biz.html` will match the above route pattern, and the match result will be `{'name': 'biz', 'ext': 'html'}`. This occurs because there is a literal part of `.` (period) between the two replacement markers `{name}` and `{ext}`.

Replacement markers can optionally specify a regular expression which will be used to decide whether a path segment should match the marker. To specify that a replacement marker should match only a specific set of characters as defined by a regular expression, you must use a slightly extended form of replacement marker syntax. Within braces, the replacement marker name must be followed by a colon, then directly thereafter, the regular expression. The *default* regular expression associated with a replacement marker `[^/]+` matches one or more characters which are not a slash. For example, under the hood, the replacement marker `{foo}` can more verbosely be spelled as `{foo:[^/]+}`. You can change this to be an arbitrary regular expression to match an arbitrary sequence of characters, such as `{foo:\d+}` to match only digits.

It is possible to use two replacement markers without any literal characters between them, for instance `/{{foo}}{bar}`. However, this would be a nonsensical pattern without specifying a custom regular expression to restrict what each marker captures.

Segments must contain at least one character in order to match a segment replacement marker. For example, for the URL `/abc/:`

- `/abc/{foo}` will not match.
- `{foo}/` will match.

Note that values representing matched path segments will be url-unquoted and decoded from UTF-8 into Unicode within the matchdict. So for instance, the following pattern:

```
foo/{bar}
```

When matching the following URL:

```
foo/La%20Pe%C3%B1a
```

The matchdict will look like so (the value is URL-decoded / UTF-8 decoded):

```
{'bar':u'La Pe\xfla'}
```

If the pattern has a `*` in it, the name which follows it is considered a “remainder match”. A remainder match *must* come at the end of the pattern. Unlike segment replacement markers, it does not need to be preceded by a slash. For example:

```
foo/{baz}/{bar}*fizzle
```

The above pattern will match these URLs, generating the following matchdicts:

```
foo/1/2/          ->
    {'baz':u'1', 'bar':u'2', 'fizzle':()}

foo/abc/def/a/b/c ->
    {'baz':u'abc', 'bar':u'def', 'fizzle':(u'a', u'b', u'c')}
```

Note that when a `*stararg` remainder match is matched, the value put into the matchdict is turned into a tuple of path segments representing the remainder of the path. These path segments are url-unquoted and decoded from UTF-8 into Unicode. For example, for the following pattern:

```
foo/*fizzle
```

When matching the following path:

6. URL DISPATCH

```
/foo/La%20Pe%C3%B1a/a/b/c
```

Will generate the following matchdict:

```
{'fizzle':(u'La Pe\xfla', u'a', u'b', u'c')}
```

By default, the `*stararg` will parse the remainder sections into a tuple split by segment. Changing the regular expression used to match a marker can also capture the remainder of the URL, for example:

```
foo/{baz}/{bar}{fizzle:.*}
```

The above pattern will match these URLs, generating the following matchdicts:

```
foo/1/2/          -> {'baz':u'1', 'bar':u'2', 'fizzle':()}\nfoo/abc/def/a/b/c -> {'baz':u'abc', 'bar':u'def', 'fizzle': u'a/b/c'}
```

This occurs because the default regular expression for a marker is `[^/]+` which will match everything up to the first `/`, while `{fizzle:.*}` will result in a regular expression match of `.*` capturing the remainder into a single value.

6.2.4 Route Declaration Ordering

Route configuration declarations are evaluated in a specific order when a request enters the system. As a result, the order of route configuration declarations is very important.

The order that routes declarations are evaluated is the order in which they are added to the application at startup time. This is unlike *traversal*, which depends on emergent behavior which happens as a result of traversing a resource tree.

For routes added via the `add_route` method, the order that routes are evaluated is the order in which they are added to the configuration imperatively.

For example, route configuration statements with the following patterns might be added in the following order:

```
members/{def}\nmembers/abc
```

In such a configuration, the `members/abc` pattern would *never* be matched. This is because the match ordering will always match `members/{def}` first; the route configuration with `members/abc` will never be evaluated.

6.2.5 Route Factories

A “route” configuration declaration can mention a “factory”. When that route matches a request, and a factory is attached to a route, the *root factory* passed at startup time to the *Configurator* is ignored; instead the factory associated with the route is used to generate a *root* object. This object will usually be used as the *context* resource of the view callable ultimately found via *view lookup*.

```
1 config.add_route('abc', '/abc', view='myproject.views.theview',
2                 factory='myproject.resources.root_factory')
```

The factory can either be a Python object or a *dotted Python name* (a string) which points to such a Python object, as it is above.

In this way, each route can use a different factory, making it possible to supply a different *context* resource object to the view related to each particular route.

Supplying a different resource factory each route is useful when you’re trying to use a Pyramid *authorization policy* to provide declarative, “context sensitive” security checks; each resource can maintain a separate *ACL*, as documented in *Using Pyramid Security With URL Dispatch*. It is also useful when you wish to combine URL dispatch with *traversal* as documented within *Combining Traversal and URL Dispatch*.

6.2.6 Route Configuration Arguments

Route configuration `add_route` statements may specify a large number of arguments. They are documented as part of the API documentation at `pyramid.config.Configurator.add_route()`.

Many of these arguments are *route predicate* arguments. A route predicate argument specifies that some aspect of the request must be true for the associated route to be considered a match during the route matching process. Examples of route predicate arguments are `pattern`, `xhr`, and `request_method`.

Other arguments are view configuration related arguments. These only have an effect when the route configuration names a `view`.

Other arguments are `name` and `factory`. These arguments represent neither predicates nor view configuration information.

6.2.7 Custom Route Predicates

Each of the predicate callables fed to the `custom_predicates` argument of `add_route()` must be a callable accepting two arguments. The first argument passed to a custom predicate is a dictionary conventionally named `info`. The second argument is the current *request* object.

The `info` dictionary has a number of contained values: `match` is a dictionary: it represents the arguments matched in the URL by the route. `route` is an object representing the route which was matched (see `pyramid.interfaces.IRoute` for the API of such a route object).

`info['match']` is useful when predicates need access to the route match. For example:

```
1 def any_of(segment_name, *allowed):
2     def predicate(info, request):
3         if info['match'][segment_name] in allowed:
4             return True
5     return predicate
6
7 num_one_two_or_three = any_of('num', 'one', 'two', 'three')
8
9 config.add_route('num', '/{num}',
10                 custom_predicates=(num_one_two_or_three,))
```

The above `any_of` function generates a predicate which ensures that the match value named `segment_name` is in the set of allowable values represented by `allowed`. We use this `any_of` function to generate a predicate function named `num_one_two_or_three`, which ensures that the `num` segment is one of the values `one`, `two`, or `three`, and use the result as a custom predicate by feeding it inside a tuple to the `custom_predicates` argument to `add_route()`.

A custom route predicate may also *modify* the `match` dictionary. For instance, a predicate might do some type conversion of values:

```
1 def integers(*segment_names):
2     def predicate(info, request):
3         match = info['match']
4         for segment_name in segment_names:
5             try:
6                 match[segment_name] = int(match[segment_name])
7             except (TypeError, ValueError):
8                 pass
9         return True
10    return predicate
11
12 ymd_to_int = integers('year', 'month', 'day')
```



```

13 |
14 | config.add_route('num', '/{year}/{month}/{day}',
15 |                 custom_predicates=(ymd_to_int,))

```

Note that a conversion predicate is still a predicate so it must return `True` or `False`; a predicate that does *only* conversion, such as the one we demonstrate above should unconditionally return `True`.

To avoid the try/except uncertainty, the route pattern can contain regular expressions specifying requirements for that marker. For instance:

```

1 | def integers(*segment_names):
2 |     def predicate(info, request):
3 |         match = info['match']
4 |         for segment_name in segment_names:
5 |             match[segment_name] = int(match[segment_name])
6 |         return True
7 |     return predicate
8 |
9 | ymd_to_int = integers('year', 'month', 'day')
10 |
11 | config.add_route('num', '/{year:\d+}/{month:\d+}/{day:\d+}',
12 |                 custom_predicates=(ymd_to_int,))

```

Now the try/except is no longer needed because the route will not match at all unless these markers match `\d+` which requires them to be valid digits for an `int` type conversion.

The `match` dictionary passed within `info` to each predicate attached to a route will be the same dictionary. Therefore, when registering a custom predicate which modifies the `match` dict, the code registering the predicate should usually arrange for the predicate to be the *last* custom predicate in the custom predicate list. Otherwise, custom predicates which fire subsequent to the predicate which performs the `match` modification will receive the *modified* match dictionary.



It is a poor idea to rely on ordering of custom predicates to build a conversion pipeline, where one predicate depends on the side effect of another. For instance, it's a poor idea to register two custom predicates, one which handles conversion of a value to an int, the next which handles conversion of that integer to some custom object. Just do all that in a single custom predicate.

The `route` object in the `info` dict is an object that has two useful attributes: `name` and `pattern`. The `name` attribute is the route name. The `pattern` attribute is the route pattern. An example of using the route in a set of route predicates:

```
1 def twenty_ten(info, request):
2     if info['route'].name in ('y', 'ym', 'y'):
3         return info['match']['year'] == '2010'
4
5 config.add_route('y', '/{year}', custom_predicates=(twenty_ten,))
6 config.add_route('ym', '/{year}/{month}', custom_predicates=(twenty_ten,))
7 config.add_route('y', '/{year}/{month}/{day}',
8                 custom_predicates=(twenty_ten,))
```

The above predicate, when added to a number of route configurations ensures that the year match argument is '2010' if and only if the route name is 'y', 'ym', or 'y'.

See also `pyramid.interfaces.IRoute` for more API documentation about route objects.

6.3 Route Matching

The main purpose of route configuration is to match (or not match) the `PATH_INFO` present in the WSGI environment provided during a request against a URL path pattern.

The way that Pyramid does this is very simple. When a request enters the system, for each route configuration declaration present in the system, Pyramid checks the `PATH_INFO` against the pattern declared.

If any route matches, the route matching process stops. The *request* is decorated with a special *interface* which describes it as a “route request”, the *context* resource is generated, and the context and the resulting request are handed off to *view lookup*. During view lookup, if any *view* argument was provided within the matched route configuration, the *view callable* it points to is called.


When a route configuration is declared, it may contain *route predicate* arguments. All route predicates associated with a route declaration must be `True` for the route configuration to be used for a given request.

If any predicate in the set of *route predicate* arguments provided to a route configuration returns `False`, that route is skipped and route matching continues through the ordered set of routes.

If no route matches after all route patterns are exhausted, Pyramid falls back to *traversal* to do *resource location* and *view lookup*.


6.3.1 The Matchdict

When the URL pattern associated with a particular route configuration is matched by a request, a dictionary named `matchdict` is added as an attribute of the `request` object. Thus, `request.matchdict` will contain the values that match replacement patterns in the `pattern` element. The keys in a `matchdict` will be strings. The values will be Unicode objects.

 If no route URL pattern matches, the `matchdict` object attached to the request will be `None`.

6.3.2 The Matched Route

When the URL pattern associated with a particular route configuration is matched by a request, an object named `matched_route` is added as an attribute of the `request` object. Thus, `request.matched_route` will be an object implementing the `IRoute` interface which matched the request. The most useful attribute of the route object is `name`, which is the name of the route that matched.

 If no route URL pattern matches, the `matched_route` object attached to the request will be `None`.

6.4 Routing Examples

Let's check out some examples of how route configuration statements might be commonly declared, and what will happen if they are matched by the information present in a request.

6.4.1 Example 1

The simplest route declaration which configures a route match to *directly* result in a particular view callable being invoked:

```
1 config.add_route('idea', 'site/{id}', view='mypackage.views.site_view')
```

6. URL DISPATCH

When a route configuration with a `view` attribute is added to the system, and an incoming request matches the *pattern* of the route configuration, the *view callable* named as the `view` attribute of the route configuration will be invoked.

In the case of the above example, when the URL of a request matches `/site/{id}`, the view callable at the Python dotted path name `mypackage.views.site_view` will be called with the request. In other words, we've associated a view callable directly with a route pattern.

When the `/site/{id}` route pattern matches during a request, the `site_view` view callable is invoked with that request as its sole argument. When this route matches, a `matchdict` will be generated and attached to the request as `request.matchdict`. If the specific URL matched is `/site/1`, the `matchdict` will be a dictionary with a single key, `id`; the value will be the string `'1'`, ex.: `{'id': '1'}`.

The `mypackage.views` module referred to above might look like so:

```
1 from pyramid.response import Response
2
3 def site_view(request):
4     return Response(request.matchdict['id'])
```

The view has access to the `matchdict` directly via the request, and can access variables within it that match keys present as a result of the route pattern.

See *Views*, and *View Configuration* for more information about views.

6.4.2 Example 2

Below is an example of a more complicated set of route statements you might add to your application:

```
1 config.add_route('idea', 'ideas/{idea}', view='mypackage.views.idea_view')
2 config.add_route('user', 'users/{user}', view='mypackage.views.user_view')
3 config.add_route('tag', 'tags/{tag}', view='mypackage.views.tag_view')
```

The above configuration will allow Pyramid to service URLs in these forms:

```
/ideas/{idea}
/users/{user}
/tags/{tag}
```

- When a URL matches the pattern `/ideas/{idea}`, the view callable available at the dotted Python pathname `mypackage.views.idea_view` will be called. For the specific URL `/ideas/1`, the `matchdict` generated and attached to the `request` will consist of `{'idea': '1'}`.
- When a URL matches the pattern `/users/{user}`, the view callable available at the dotted Python pathname `mypackage.views.user_view` will be called. For the specific URL `/users/1`, the `matchdict` generated and attached to the `request` will consist of `{'user': '1'}`.
- When a URL matches the pattern `/tags/{tag}`, the view callable available at the dotted Python pathname `mypackage.views.tag_view` will be called. For the specific URL `/tags/1`, the `matchdict` generated and attached to the `request` will consist of `{'tag': '1'}`.

In this example we've again associated each of our routes with a *view callable* directly. In all cases, the request, which will have a `matchdict` attribute detailing the information found in the URL by the process will be passed to the view callable.

6.4.3 Example 3

The *context* resource object passed in to a view found as the result of URL dispatch will, by default, be an instance of the object returned by the *root factory* configured at startup time (the `root_factory` argument to the *Configurator* used to configure the application).

You can override this behavior by passing in a *factory* argument to the `add_route()` method for a particular route. The *factory* should be a callable that accepts a *request* and returns an instance of a class that will be the context resource used by the view.

An example of using a route with a factory:

```
1 config.add_route('idea', 'ideas/{idea}',
2                 view='myproject.views.idea_view',
3                 factory='myproject.resources.Idea')
```

The above route will manufacture an *Idea* resource as a *context*, assuming that `mypackage.resources.Idea` resolves to a class that accepts a request in its `__init__`. For example:

```
1 class Idea(object):
2     def __init__(self, request):
3         pass
```

In a more complicated application, this root factory might be a class representing a *SQLAlchemy* model.

6.4.4 Example 4

It is possible to create a route declaration without a `view` attribute, but associate the route with a *view callable* using a `view` declaration.

```
1 config.add_route('idea', 'site/{id}')
2 config.add_view(route_name='idea', view='mypackage.views.site_view')
```

This set of configuration parameters creates a configuration completely equivalent to this example provided in *Example 1*:

```
1 config.add_route('idea', 'site/{id}', view='mypackage.views.site_view')
```

In fact, the spelling which names a `view` attribute is just syntactic sugar for the more verbose spelling which contains separate view and route registrations.

More uses for this style of associating views with routes are explored in *Combining Traversal and URL Dispatch*.

6.5 Matching the Root URL

It's not entirely obvious how to use a route pattern to match the root URL ("`/`"). To do so, give the empty string as a pattern in a call to `add_route()`:

```
1 config.add_route('root', '', view='mypackage.views.root_view')
```

Or provide the literal string `/` as the pattern:

```
1 config.add_route('root', '/', view='mypackage.views.root_view')
```

6.6 Generating Route URLs

Use the `pyramid.url.route_url()` function to generate URLs based on route patterns. For example, if you've configured a route with the name "foo" and the pattern "`{a}/{b}/{c}`", you might do this.

```

1 from pyramid.url import route_url
2 url = route_url('foo', request, a='1', b='2', c='3')

```

This would return something like the string `http://example.com/1/2/3` (at least if the current protocol and hostname implied `http://example.com`). See the `route_url()` API documentation for more information.

6.7 Redirecting to Slash-Appended Routes

For behavior like Django’s `APPEND_SLASH=True`, use the `append_slash_notfound_view()` view as the *Not Found* view in your application. Defining this view as the *Not Found* view is a way to automatically redirect requests where the URL lacks a trailing slash, but requires one to match the proper route. When configured, along with at least one other route in your application, this view will be invoked if the value of `PATH_INFO` does not already end in a slash, and if the value of `PATH_INFO` *plus* a slash matches any route’s pattern. In this case it does an HTTP redirect to the slash-appended `PATH_INFO`.

Let’s use an example, because this behavior is a bit magical. If the `append_slash_notfound_view` is configured in your application and your route configuration looks like so:

```

1 config.add_route('noslash', 'no_slash',
2                 view='myproject.views.no_slash')
3 config.add_route('haslash', 'has_slash/',
4                 view='myproject.views.has_slash')

```

If a request enters the application with the `PATH_INFO` value of `/has_slash/`, the second route will match. If a request enters the application with the `PATH_INFO` value of `/has_slash`, a route *will* be found by the slash-appending not found view. An HTTP redirect to `/has_slash/` will be returned to the user’s browser.

If a request enters the application with the `PATH_INFO` value of `/no_slash`, the first route will match. However, if a request enters the application with the `PATH_INFO` value of `/no_slash/`, *no* route will match, and the slash-appending not found view will *not* find a matching route with an appended slash.



You *should not* rely on this mechanism to redirect POST requests. The redirect of the slash-appending not found view will turn a POST request into a GET, losing any POST data in the original request.

To configure the slash-appending not found view in your application, change the application’s startup configuration, adding the following stanza:

```
1 config.add_view(context='pyramid.exceptions.NotFound',
2                 view='pyramid.view.append_slash_notfound_view')
```

See *pyramid.view* and *Changing the Not Found View* for more information about the slash-appending not found view and for a more general description of how to configure a not found view.

6.7.1 Custom Not Found View With Slash Appended Routes

There can only be one *Not Found view* in any Pyramid application. Even if you use `append_slash_notfound_view()` as the Not Found view, Pyramid still must generate a 404 Not Found response when it cannot redirect to a slash-appended URL; this not found response will be visible to site users.

If you don't care what this 404 response looks like, and only you need redirections to slash-appended route URLs, you may use the `append_slash_notfound_view()` object as the Not Found view as described above. However, if you wish to use a *custom* notfound view callable when a URL cannot be redirected to a slash-appended URL, you may wish to use an instance of the `AppendSlashNotFoundViewFactory` class as the Not Found view, supplying a *view callable* to be used as the custom notfound view as the first argument to its constructor. For instance:

```
1 from pyramid.exceptions import NotFound
2 from pyramid.view import AppendSlashNotFoundViewFactory
3
4 def notfound_view(context, request):
5     return HTTPNotFound('It aint there, stop trying!')
6
7 custom_append_slash = AppendSlashNotFoundViewFactory(notfound_view)
8 config.add_view(custom_append_slash, context=NotFound)
```

The `notfound_view` supplied must adhere to the two-argument view callable calling convention of `(context, request)` (context will be the exception object).

6.8 Cleaning Up After a Request

Sometimes it's required that some cleanup be performed at the end of a request when a database connection is involved.


For example, let's say you have a `mypackage` Pyramid application package that uses SQLAlchemy, and you'd like the current SQLAlchemy database session to be removed after each request. Put the following in the `mypackage.__init__` module:


```

1 from mypackage.models import DBSession
2
3 from pyramid.events import subscriber
4 from pyramid.events import NewRequest
5
6 def cleanup_callback(request):
7     DBSession.remove()
8
9 @subscriber(NewRequest)
10 def add_cleanup_callback(event):
11     event.request.add_finished_callback(cleanup_callback)

```

Registering the `cleanup_callback` finished callback at the start of a request (by causing the `add_cleanup_callback` to receive a `pyramid.events.NewRequest` event at the start of each request) will cause the `DBSession` to be removed whenever request processing has ended. Note that in the example above, for the `pyramid.events.subscriber` decorator to “work”, the `pyramid.config.Configurator.scan()` method must be called against your `mypackage` package during application initialization.

 This is only an example. In particular, it is not necessary to cause `DBSession.remove` to be called in an application generated from any Pyramid paster template, because these all use the `repoze.tm2` middleware. The cleanup done by `DBSession.remove` is unnecessary when `repoze.tm2` middleware is in the WSGI pipeline.

6.9 Using Pyramid Security With URL Dispatch

Pyramid provides its own security framework which consults an *authorization policy* before allowing any application code to be called. This framework operates in terms of an access control list, which is stored as an `__acl__` attribute of a resource object. A common thing to want to do is to attach an `__acl__` to the resource object dynamically for declarative security purposes. You can use the `factory` argument that points at a factory which attaches a custom `__acl__` to an object at its creation time.

Such a factory might look like so:

```

1 class Article(object):
2     def __init__(self, request):
3         matchdict = request.matchdict
4         article = matchdict.get('article', None)
5         if article == '1':
6             self.__acl__ = [ (Allow, 'editor', 'view') ]

```

6. URL DISPATCH

If the route `archives/{article}` is matched, and the article number is 1, Pyramid will generate an `Article context` resource with an ACL on it that allows the `editor` principal the `view` permission. Obviously you can do more generic things than inspect the routes match dict to see if the `article` argument matches a particular string; our sample `Article` factory class is not very ambitious.



See *Security* for more information about Pyramid security and ACLs.

6.10 Debugging Route Matching

It's useful to be able to take a peek under the hood when requests that enter your application aren't matching your routes as you expect them to. To debug route matching, use the `PYRAMID_DEBUG_ROUTE_MATCH` environment variable or the `debug_routematch` configuration file setting (set either to `true`). Details of the route matching decision for a particular request to the Pyramid application will be printed to the `stderr` of the console which you started the application from. For example:

```
1 [chrism@thinko pylonsbasic]$ PYRAMID_DEBUG_ROUTE_MATCH=true \  
2 bin/paster serve development.ini  
3 Starting server in PID 13586.  
4 serving on 0.0.0.0:6543 view at http://127.0.0.1:6543  
5 2010-12-16 14:45:19,956 no route matched for url \  
6 http://localhost:6543/wontmatch  
7 2010-12-16 14:45:20,010 no route matched for url \  
8 http://localhost:6543/favicon.ico  
9 2010-12-16 14:41:52,084 route matched for url \  
10 http://localhost:6543/static/logo.png; \  
11 route_name: 'static/', ....
```

See *Environment Variables and .ini File Settings* for more information about how, and where to set these values.

6.11 Displaying All Application Routes

You can use the `paster proutes` command in a terminal window to print a summary of routes related to your application. Much like the `paster pshell` command (see *The Interactive Shell*), the `paster proutes` command accepts two arguments. The first argument to `proutes` is the path to

your application's `.ini` file. The second is the `app` section name inside the `.ini` file which points to your application.

For example:

```

1 [chris@thinko MyProject]$ ../bin/paster proutes development.ini MyProject
2 Name          Pattern          View
3 ----          -
4 home          /                <function my_view>
5 home2         /                <function my_view>
6 another       /another         None
7 static/       static/*subpath <static_view object>
8 catchall      /*subpath       <function static_view>

```

`paster proutes` generates a table. The table has three columns: a `Name` name column, a `Pattern` column, and a `View` column. The items listed in the `Name` column are route names, the items listed in the `Pattern` column are route patterns, and the items listed in the `View` column are representations of the view callable that will be invoked when a request matches the associated route pattern. The view column may show `None` if no associated view callable could be found. If no routes are configured within your application, nothing will be printed to the console when `paster proutes` is executed.

6.12 Route View Callable Registration and Lookup Details

The purpose of making it possible to specify a view callable within a route configuration is to prevent developers from needing to deeply understand the details of *resource location* and *view lookup*. When a route names a view callable as a `view` argument, and a request enters the system which matches the pattern of the route, the result is simple: the view callable associated with the route is invoked with the request that caused the invocation.

For most usage, you needn't understand more than this; how it works is an implementation detail. In the interest of completeness, however, we'll explain how it *does* work in this section. You can skip it if you're uninterested.

When a `view` attribute is attached to a route configuration, Pyramid ensures that a *view configuration* is registered that will always be found when the route pattern is matched during a request. To do so:

- A special route-specific *interface* is created at startup time for each route configuration declaration.
- When a route configuration declaration mentions a `view` attribute, a *view configuration* is registered at startup time. This view configuration uses the route-specific interface as a *request* type.


- At runtime, when a request causes any route to match, the *request* object is decorated with the route-specific interface.
- The fact that the request is decorated with a route-specific interface causes the view lookup machinery to always use the view callable registered using that interface by the route configuration to service requests that match the route pattern.

In this way, we supply a shortcut to the developer. Under the hood, the *resource location* and *view lookup* subsystems provided by Pyramid are still being utilized, but in a way which does not require a developer to understand either of them in detail. It also means that we can allow a developer to combine *URL dispatch* and *traversal* in various exceptional cases as documented in *Combining Traversal and URL Dispatch*.


6.13 References

A tutorial showing how *URL dispatch* can be used to create a Pyramid application exists in *SQLAlchemy + URL Dispatch Wiki Tutorial*.

MUCH ADO ABOUT TRAVERSAL

 This chapter was adapted, with permission, from a blog post by Rob Miller, originally published at <http://blog.nonsequitarian.org/2010/much-ado-about-traversal/>.

Traversal is an alternative to *URL dispatch* which allows Pyramid applications to map URLs to code.

 Ex-Zope users whom are already familiar with traversal and view lookup conceptually may want to skip directly to the *Traversal* chapter, which discusses technical details. This chapter is mostly aimed at people who have previous *Pylons* experience or experience in another framework which does not provide traversal, and need an introduction to the “why” of traversal.

Some folks who have been using Pylons and its Routes-based URL matching for a long time are being exposed for the first time, via Pyramid, to new ideas such as “*traversal*” and “*view lookup*” as a way to route incoming HTTP requests to callable code. Some of the same folks believe that traversal is hard to understand. Others question its usefulness; URL matching has worked for them so far, why should they even consider dealing with another approach, one which doesn’t fit their brain and which doesn’t provide any immediately obvious value?

You can be assured that if you don’t want to understand traversal, you don’t have to. You can happily build Pyramid applications with only *URL dispatch*. However, there are some straightforward, real-world use cases that are much more easily served by a traversal-based approach than by a pattern-matching mechanism. Even if you haven’t yet hit one of these use cases yourself, understanding these new ideas is worth the effort for any web developer so you know when you might want to use them. *Traversal* is actually a straightforward metaphor easily comprehended by anyone who’s ever used a run-of-the-mill file system with folders and files.

7.1 URL Dispatch

Let's step back and consider the problem we're trying to solve. An HTTP request for a particular path has been routed to our web application. The requested path will possibly invoke a specific *view callable* function defined somewhere in our app. We're trying to determine *which* callable function, if any, should be invoked for a given requested URL.

Many systems, including Pyramid, offer a simple solution. They offer the concept of "URL matching". URL matching approaches this problem by parsing the URL path and comparing the results to a set of registered "patterns", defined by a set of regular expressions, or some other URL path templating syntax. Each pattern is mapped to a callable function somewhere; if the request path matches a specific pattern, the associated function is called. If the request path matches more than one pattern, some conflict resolution scheme is used, usually a simple order precedence so that the first match will take priority over any subsequent matches. If a request path doesn't match any of the defined patterns, a "404 Not Found" response is returned.

In Pyramid, we offer an implementation of URL matching which we call *URL dispatch*. Using Pyramid syntax, we might have a match pattern such as `{userid}/photos/{photoid}`, mapped to a `photo_view()` function defined somewhere in our code. Then a request for a path such as `/joeschmoe/photos/photo1` would be a match, and the `photo_view()` function would be invoked to handle the request. Similarly, `{userid}/blog/{year}/{month}/{postid}` might map to a `blog_post_view()` function, so `/joeschmoe/blog/2010/12/urlmatching` would trigger the function, which presumably would know how to find and render the `urlmatching` blog post.

7.2 Historical Refresher

Now that we've refreshed our understanding of *URL dispatch*, we'll dig in to the idea of traversal. Before we do, though, let's take a trip down memory lane. If you've been doing web work for a while, you may remember a time when we didn't have fancy web frameworks like *Pylons* and Pyramid. Instead, we had general purpose HTTP servers that primarily served files off of a file system. The "root" of a given site mapped to a particular folder somewhere on the file system. Each segment of the request URL path represented a subdirectory. The final path segment would be either a directory or a file, and once the server found the right file it would package it up in an HTTP response and send it back to the client. So serving up a request for `/joeschmoe/photos/photo1` literally meant that there was a `joeschmoe` folder somewhere, which contained a `photos` folder, which in turn contained a `photo1` file. If at any point along the way we find that there is not a folder or file matching the requested path, we return a 404 response.

As the web grew more dynamic, however, a little bit of extra complexity was added. Technologies such as CGI and HTTP server modules were developed. Files were still looked up on the file system, but if the

file ended with (for example) `.cgi` or `.php`, or if it lived in a special folder, instead of simply sending the file to the client the server would read the file, execute it using an interpreter of some sort, and then send the output from this process to the client as the final result. The server configuration specified which files would trigger some dynamic code, with the default case being to just serve the static file.

7.3 Traversal (aka Resource Location)

Believe it or not, if you understand how serving files from a file system works, you understand traversal. And if you understand that a server might do something different based on what type of file a given request specifies, then you understand view lookup.

The major difference between file system lookup and traversal is that a file system lookup steps through nested directories and files in a file system tree, while traversal steps through nested dictionary-type objects in a *resource tree*. Let's take a detailed look at one of our example paths, so we can see what I mean:

The path `/joeschmoe/photos/photo1`, has four segments: `/`, `joeschmoe`, `photos` and `photo1`. With file system lookup we might have a root folder (`/`) containing a nested folder (`joeschmoe`), which contains another nested folder (`photos`), which finally contains a JPG file (`photo1`). With traversal, we instead have a dictionary-like root object. Asking for the `joeschmoe` key gives us another dictionary-like object. Asking this in turn for the `photos` key gives us yet another mapping object, which finally (hopefully) contains the resource that we're looking for within its values, referenced by the `photo1` key.

In pure Python terms, then, the traversal or “resource location” portion of satisfying the `/joeschmoe/photos/photo1` request will look something like this pseudocode:

```
get_root() ['joeschmoe'] ['photos'] ['photo1']
```

`get_root()` is some function that returns a root traversal *resource*. If all of the specified keys exist, then the returned object will be the resource that is being requested, analogous to the JPG file that was retrieved in the file system example. If a `KeyError` is generated anywhere along the way, Pyramid will return 404. (This isn't precisely true, as you'll see when we learn about view lookup below, but the basic idea holds.)

7.4 What Is a “Resource”?

“Files on a file system I understand”, you might say. “But what are these nested dictionary things? Where do these objects, these ‘resources’, live? What *are* they?”

Since Pyramid is not a highly opinionated framework, it makes no restriction on how a *resource* is implemented; a developer can implement them as he wishes. One common pattern used is to persist all of the resources, including the root, in a database as a graph. The root object is a dictionary-like object. Dictionary-like objects in Python supply a `__getitem__` method which is called when key lookup is done. Under the hood, when `adict` is a dictionary-like object, Python translates `adict['a']` to `adict.__getitem__('a')`. Try doing this in a Python interpreter prompt if you don't believe us:

```
1 Python 2.4.6 (#2, Apr 29 2010, 00:31:48)
2 [GCC 4.4.3] on linux2
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>> adict = {}
5 >>> adict['a'] = 1
6 >>> adict['a']
7 1
8 >>> adict.__getitem__('a')
9 1
```

The dictionary-like root object stores the ids of all of its subresources as keys, and provides a `__getitem__` implementation that fetches them. So `get_root()` fetches the unique root object, while `get_root()['joeschmoe']` returns a different object, also stored in the database, which in turn has its own subresources and `__getitem__` implementation, etc. These resources might be persisted in a relational database, one of the many “NoSQL” solutions that are becoming popular these days, or anywhere else, it doesn't matter. As long as the returned objects provide the dictionary-like API (i.e. as long as they have an appropriately implemented `__getitem__` method) then traversal will work.

In fact, you don't need a “database” at all. You could use plain dictionaries, with your site's URL structure hard-coded directly in the Python source. Or you could trivially implement a set of objects with `__getitem__` methods that search for files in specific directories, and thus precisely recreate the traditional mechanism of having the URL path mapped directly to a folder structure on the file system. Traversal is in fact a superset of file system lookup.



See the chapter entitled *Resources* for a more technical overview of resources.

7.5 View Lookup

At this point we're nearly there. We've covered traversal, which is the process by which a specific resource is retrieved according to a specific URL path. But what is "view lookup"?

The need for view lookup is simple: there is more than one possible action that you might want to take after finding a *resource*. With our photo example, for instance, you might want to view the photo in a page, but you might also want to provide a way for the user to edit the photo and any associated metadata. We'll call the former the `view` view, and the latter will be the `edit` view. (Original, I know.) Pyramid has a centralized view *application registry* where named views can be associated with specific resource types. So in our example, we'll assume that we've registered `view` and `edit` views for `photo` objects, and that we've specified the `view` view as the default, so that `/joeschmoe/photos/photo1/view` and `/joeschmoe/photos/photo1` are equivalent. The `edit` view would sensibly be provided by a request for `/joeschmoe/photos/photo1/edit`.

Hopefully it's clear that the first portion of the `edit` view's URL path is going to resolve to the same resource as the non-`edit` version, specifically the resource returned by `get_root() ['joeschmoe'] ['photos'] ['photo1']`. But traversal ends there; the `photo1` resource doesn't have an `edit` key. In fact, it might not even be a dictionary-like object, in which case `photo1['edit']` would be meaningless. When the Pyramid resource location has been resolved to a *leaf* resource, but the entire request path has not yet been expended, the *very next* path segment is treated as a *view name*. The registry is then checked to see if a view of the given name has been specified for a resource of the given type. If so, the view callable is invoked, with the resource passed in as the related `context` object (also available as `request.context`). If a view callable could not be found, Pyramid will return a "404 Not Found" response.

You might conceptualize a request for `/joeschmoe/photos/photo1/edit` as ultimately converted into the following piece of Pythonic pseudocode:

```
context = get_root() ['joeschmoe'] ['photos'] ['photo1']
view_callable = get_view(context, 'edit')
request.context = context
view_callable(request)
```

The `get_root` and `get_view` functions don't really exist. Internally, Pyramid does something more complicated. But the example above is a reasonable approximation of the view lookup algorithm in pseudocode.

7.6 Use Cases

Why should we care about traversal? URL matching is easier to explain, and it's good enough, right?

In some cases, yes, but certainly not in all cases. So far we've had very structured URLs, where our paths have had a specific, small number of pieces, like this:

```
/{userid}/{typename}/{objectid}[/{view_name}]
```

In all of the examples thus far, we've hard coded the `typename` value, assuming that we'd know at development time what names were going to be used ("photos", "blog", etc.). But what if we don't know what these names will be? Or, worse yet, what if we don't know *anything* about the structure of the URLs inside a user's folder? We could be writing a CMS where we want the end user to be able to arbitrarily add content and other folders inside his folder. He might decide to nest folders dozens of layers deep. How will you construct matching patterns that could account for every possible combination of paths that might develop?


It might be possible, but it certainly won't be easy. The matching patterns are going to become complex quickly as you try to handle all of the edge cases.

With traversal, however, it's straightforward. Twenty layers of nesting would be no problem. Pyramid will happily call `__getitem__` as many times as it needs to, until it runs out of path segments or until a resource raises a `KeyError`. Each resource only needs to know how to fetch its immediate children, the traversal algorithm takes care of the rest. Also, since the structure of the resource tree can live in the database and not in the code, it's simple to let users modify the tree at runtime to set up their own personalized "directory" structures.

Another use case in which traversal shines is when there is a need to support a context-dependent security policy. One example might be a document management infrastructure for a large corporation, where members of different departments have varying access levels to the various other departments' files. Reasonably, even specific files might need to be made available to specific individuals. Traversal does well here if your resources actually represent the data objects related to your documents, because the idea of a resource authorization is baked right into the code resolution and calling process. Resource objects can store ACLs, which can be inherited and/or overridden by the subresources.

If each resource can thus generate a context-based ACL, then whenever view code is attempting to perform a sensitive action, it can check against that ACL to see whether the current user should be allowed to perform the action. In this way you achieve so called "instance based" or "row level" security which is considerably harder to model using a traditional tabular approach. Pyramid actively supports such a scheme, and in fact if you register your views with guard permissions and use an authorization policy, Pyramid can check against a resource's ACL when deciding whether or not the view itself is available to the current user.

In summary, there are entire classes of problems that are more easily served by traversal and view lookup than by *URL dispatch*. If your problems don't require it, great: stick with *URL dispatch*. But if you're using Pyramid and you ever find that you *do* need to support one of these use cases, you'll be glad you have traversal in your toolkit.

 It is even possible to mix and match *traversal* with *URL dispatch* in the same Pyramid application. See the *Combining Traversal and URL Dispatch* chapter for details.

TRAVERSAL

A *traversal* uses the URL (Universal Resource Locator) to find a *resource* located in a *resource tree*, which is a set of nested dictionary-like objects. Traversal is done by using each segment of the path portion of the URL to navigate through the *resource tree*. You might think of this as looking up files and directories in a file system. Traversal walks down the path until it finds a published resource, analogous to a file system “directory” or “file”. The resource found as the result of a traversal becomes the *context* of the *request*. Then, the *view lookup* subsystem is used to find some view code willing “publish” this resource by generating a *response*.

Using *Traversal* to map a URL to code is optional. It is often less easy to understand than *URL dispatch*, so if you’re a rank beginner, it probably makes sense to use URL dispatch to map URLs to code instead of traversal. In that case, you can skip this chapter.

8.1 Traversal Details

Traversal is dependent on information in a *request* object. Every *request* object contains URL path information in the `PATH_INFO` portion of the *WSGI* environment. The `PATH_INFO` string is the portion of a request’s URL following the hostname and port number, but before any query string elements or fragment element. For example the `PATH_INFO` portion of the URL `http://example.com:8080/a/b/c?foo=1` is `/a/b/c`.

Traversal treats the `PATH_INFO` segment of a URL as a sequence of path segments. For example, the `PATH_INFO` string `/a/b/c` is converted to the sequence `['a', 'b', 'c']`.

This path sequence is then used to descend through the *resource tree*, looking up a resource for each path segment. Each lookup uses the `__getitem__` method of a resource in the tree.

For example, if the path info sequence is `['a', 'b', 'c']`:

- *Traversal* starts by acquiring the *root* resource of the application by calling the *root factory*. The *root factory* can be configured to return whatever object is appropriate as the traversal root of your application.
- Next, the first element (a) is popped from the path segment sequence and is used as a key to lookup the corresponding resource in the root. This invokes the root resource's `__getitem__` method using that value (a) as an argument.
- If the root resource “contains” a resource with key a, its `__getitem__` method will return it. The *context* temporarily becomes the “A” resource.
- The next segment (b) is popped from the path sequence, and the “A” resource's `__getitem__` is called with that value (b) as an argument; we'll presume it succeeds.
- The “A” resource's `__getitem__` returns another resource, which we'll call “B”. The *context* temporarily becomes the “B” resource.

Traversal continues until the path segment sequence is exhausted or a path element cannot be resolved to a resource. In either case, the *context* resource is the last object that the traversal successfully resolved. If any resource found during traversal lacks a `__getitem__` method, or if its `__getitem__` method raises a `KeyError`, traversal ends immediately, and that resource becomes the *context*.

The results of a *traversal* also include a *view name*. If traversal ends before the path segment sequence is exhausted, the *view name* is the *next* remaining path segment element. If the *traversal* expends all of the path segments, then the *view name* is the empty string ('').

The combination of the context resource and the *view name* found via traversal is used later in the same request by the *view lookup* subsystem to find a *view callable*. How Pyramid performs view lookup is explained within the *View Configuration* chapter.

8.2 The Resource Tree

The resource tree is a set of nested dictionary-like resource objects that begins with a *root* resource. In order to use *traversal* to resolve URLs to code, your application must supply a *resource tree* to Pyramid.

In order to supply a root resource for an application the Pyramid *Router* is configured with a call-back known as a *root factory*. The root factory is supplied by the application, at startup time, as the `root_factory` argument to the *Configurator*.

The root factory is a Python callable that accepts a *request* object, and returns the root object of the *resource tree*. A function, or class is typically used as an application's root factory. Here's an example of a simple root factory class:

```

1 class Root(dict):
2     def __init__(self, request):
3         pass

```

Here’s an example of using this root factory within startup configuration, by passing it to an instance of a *Configurator* named `config`:

```

1 config = Configurator(root_factory=Root)

```

The `root_factory` argument to the *Configurator* constructor registers this root factory to be called to generate a root resource whenever a request enters the application. The root factory registered this way is also known as the global root factory. A root factory can alternately be passed to the *Configurator* as a *dotted Python name* which can refer to a root factory defined in a different module.

If no *root factory* is passed to the Pyramid *Configurator* constructor, or if the `root_factory` value specified is `None`, a *default* root factory is used. The default root factory always returns a resource that has no child resources; it is effectively empty.

Usually a root factory for a traversal-based application will be more complicated than the above `Root` class; in particular it may be associated with a database connection or another persistence mechanism.

Emulating the Default Root Factory

For purposes of understanding the default root factory better, we’ll note that you can emulate the default root factory by using this code as an explicit root factory in your application setup:

```

1 class Root(object):
2     def __init__(self, request):
3         pass
4
5 config = Configurator(root_factory=Root)

```

The default root factory is just a really stupid object that has no behavior or state. Using *traversal* against an application that uses the resource tree supplied by the default root resource is not very interesting, because the default root resource has no children. Its availability is more useful when you’re developing an application using *URL dispatch*.

i If the items contained within the resource tree are “persistent” (they have state that lasts longer than the execution of a single process), they become analogous to the concept of *domain model* objects used by many other frameworks.

The resource tree consists of *container* resources and *leaf* resources. There is only one difference between a *container* resource and a *leaf* resource: *container* resources possess a `__getitem__` method (making it “dictionary-like”) while *leaf* resources do not. The `__getitem__` method was chosen as the signifying difference between the two types of resources because the presence of this method is how Python itself typically determines whether an object is “containerish” or not (dictionary objects are “containerish”).

Each container resource is presumed to be willing to return a child resource or raise a `KeyError` based on a name passed to its `__getitem__`.

Leaf-level instances must not have a `__getitem__`. If instances that you’d like to be leaves already happen to have a `__getitem__` through some historical inequity, you should subclass these resource types and cause their `__getitem__` methods to simply raise a `KeyError`. Or just disuse them and think up another strategy.

Usually, the traversal root is a *container* resource, and as such it contains other resources. However, it doesn’t *need* to be a container. Your resource tree can be as shallow or as deep as you require.

In general, the resource tree is traversed beginning at its root resource using a sequence of path elements described by the `PATH_INFO` of the current request; if there are path segments, the root resource’s `__getitem__` is called with the next path segment, and it is expected to return another resource. The resulting resource’s `__getitem__` is called with the very next path segment, and it is expected to return another resource. This happens *ad infinitum* until all path segments are exhausted.

8.3 The Traversal Algorithm

This section will attempt to explain the Pyramid traversal algorithm. We’ll provide a description of the algorithm, a diagram of how the algorithm works, and some example traversal scenarios that might help you understand how the algorithm operates against a specific resource tree.

We’ll also talk a bit about *view lookup*. The *View Configuration* chapter discusses *view lookup* in detail, and it is the canonical source for information about views. Technically, *view lookup* is a Pyramid subsystem that is separated from traversal entirely. However, we’ll describe the fundamental behavior of view lookup in the examples in the next few sections to give you an idea of how traversal and view lookup cooperate, because they are almost always used together.

8.3.1 A Description of The Traversal Algorithm

When a user requests a page from your traversal-powered application, the system uses this algorithm to find a *context* resource and a *view name*.

1. The request for the page is presented to the Pyramid *router* in terms of a standard *WSGI* request, which is represented by a WSGI environment and a WSGI `start_response` callable.
2. The router creates a *request* object based on the WSGI environment.
3. The *root factory* is called with the *request*. It returns a *root* resource.
4. The router uses the WSGI environment's `PATH_INFO` information to determine the path segments to traverse. The leading slash is stripped off `PATH_INFO`, and the remaining path segments are split on the slash character to form a traversal sequence.

The traversal algorithm by default attempts to first URL-unquote and then Unicode-decode each path segment derived from `PATH_INFO` from its natural byte string (`str` type) representation. URL unquoting is performed using the Python standard library `urllib.unquote` function. Conversion from a URL-decoded string into Unicode is attempted using the UTF-8 encoding. If any URL-unquoted path segment in `PATH_INFO` is not decodeable using the UTF-8 decoding, a `TypeError` is raised. A segment will be fully URL-unquoted and UTF8-decoded before it is passed it to the `__getitem__` of any resource during traversal.

Thus, a request with a `PATH_INFO` variable of `/a/b/c` maps to the traversal sequence `[u' a' , u' b' , u' c']`.

5. *Traversal* begins at the root resource returned by the root factory. For the traversal sequence `[u' a' , u' b' , u' c']`, the root resource's `__getitem__` is called with the name `a`. Traversal continues through the sequence. In our example, if the root resource's `__getitem__` called with the name `a` returns a resource (aka "resource a"), that resource's `__getitem__` is called with the name `b`. If resource A returns a resource when asked for `b`, "resource b"'s `__getitem__` is then asked for the name `c`, and may return "resource c".
6. Traversal ends when a) the entire path is exhausted or b) when any resource raises a `KeyError` from its `__getitem__` or c) when any non-final path element traversal does not have a `__getitem__` method (resulting in a `NameError`) or d) when any path element is prefixed with the set of characters `@@` (indicating that the characters following the `@@` token should be treated as a *view name*).
7. When traversal ends for any of the reasons in the previous step, the last resource found during traversal is deemed to be the *context*. If the path has been exhausted when traversal ends, the *view name* is deemed to be the empty string (`''`). However, if the path was *not* exhausted before traversal terminated, the first remaining path segment is treated as the view name.

8. TRAVERSAL

8. Any subsequent path elements after the *view name* is found are deemed the *subpath*. The subpath is always a sequence of path segments that come from `PATH_INFO` that are “left over” after traversal has completed.

Once the *context* resource, the *view name*, and associated attributes such as the *subpath* are located, the job of *traversal* is finished. It passes back the information it obtained to its caller, the *Pyramid Router*, which subsequently invokes *view lookup* with the context and view name information.

The traversal algorithm exposes two special cases:

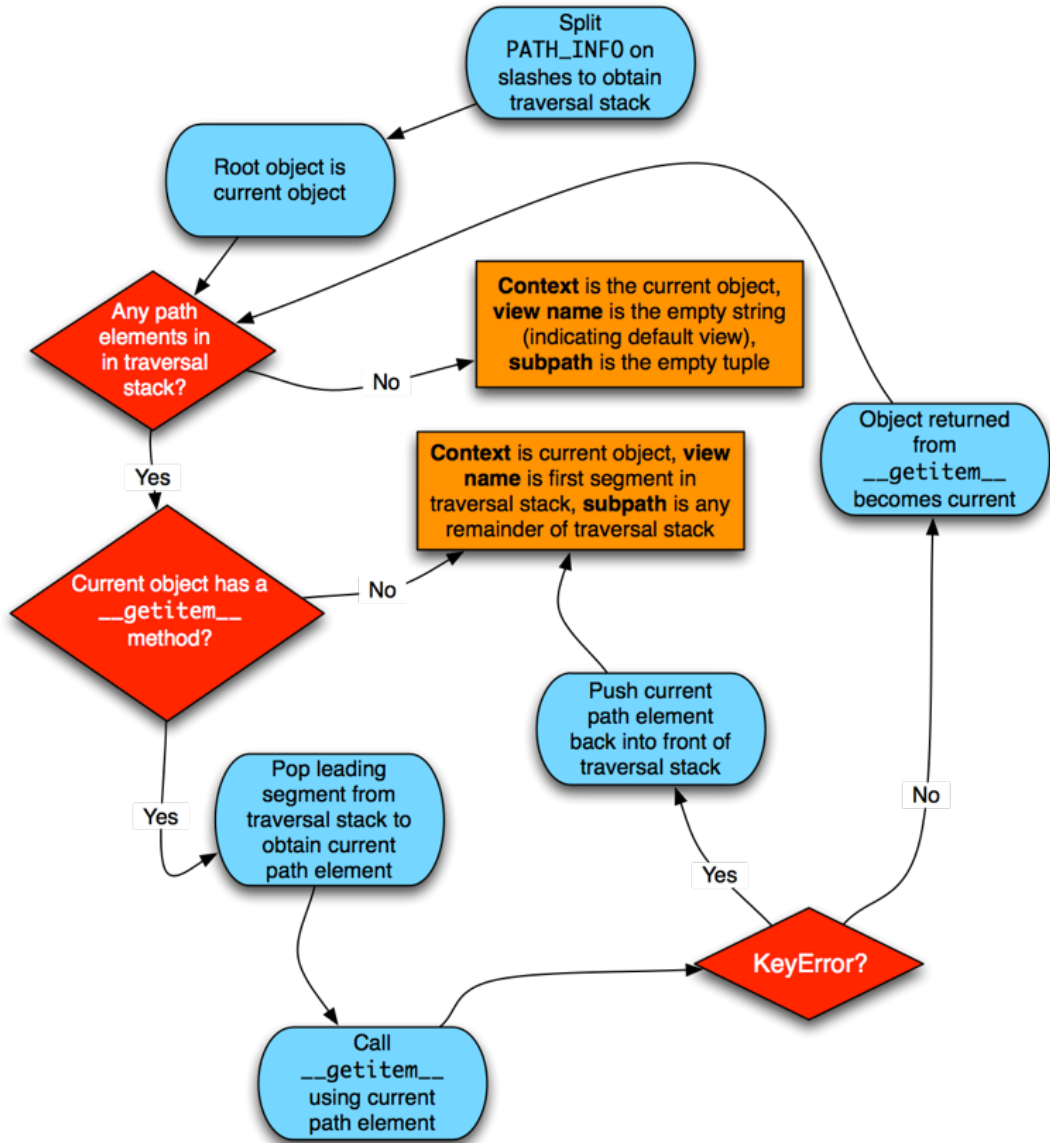
- You will often end up with a *view name* that is the empty string as the result of a particular traversal. This indicates that the view lookup machinery should look up the *default view*. The default view is a view that is registered with no name or a view which is registered with a name that equals the empty string.
- If any path segment element begins with the special characters @@ (think of them as goggles), the value of that segment minus the goggle characters is considered the *view name* immediately and traversal stops there. This allows you to address views that may have the same names as resource names in the tree unambiguously.

Finally, traversal is responsible for locating a *virtual root*. A virtual root is used during “virtual hosting”; see the *Virtual Hosting* chapter for information. We won’t speak more about it in this chapter.



Pyramid™

Model Graph Traversal



8.3.2 Traversal Algorithm Examples

No one can be expected to understand the traversal algorithm by analogy and description alone, so let's examine some traversal scenarios that use concrete URLs and resource tree compositions.

Let's pretend the user asks for `http://example.com/foo/bar/baz/biz/buz.txt`. The request's `PATH_INFO` in that case is `/foo/bar/baz/biz/buz.txt`. Let's further pretend that when this request comes in that we're traversing the following resource tree:

```
/--  
 |  
 |-- foo  
    |  
    ----bar
```

Here's what happens:

- `traversal` traverses the root, and attempts to find “foo”, which it finds.
- `traversal` traverses “foo”, and attempts to find “bar”, which it finds.
- `traversal` traverses bar, and attempts to find “baz”, which it does not find (the “bar” resource raises a `KeyError` when asked for “baz”).

The fact that it does not find “baz” at this point does not signify an error condition. It signifies that:

- the *context* is the “bar” resource (the context is the last resource found during traversal).
- the *view name* is `baz`
- the *subpath* is `('biz', 'buz.txt')`

At this point, traversal has ended, and *view lookup* begins.

Because it's the “context” resource, the view lookup machinery examines “bar” to find out what “type” it is. Let's say it finds that the context is a `Bar` type (because “bar” happens to be an instance of the class `Bar`). Using the *view name* (`baz`) and the type, view lookup asks the *application registry* this question:

- Please find me a *view callable* registered using a *view configuration* with the name “baz” that can be used for the class `Bar`.

Let's say that view lookup finds no matching view type. In this circumstance, the Pyramid *router* returns the result of the *not found view* and the request ends.

However, for this tree:



The user asks for `http://example.com/foo/bar/baz/biz/buz.txt`

- traversal traverses “foo”, and attempts to find “bar”, which it finds.
- traversal traverses “bar”, and attempts to find “baz”, which it finds.
- traversal traverses “baz”, and attempts to find “biz”, which it finds.
- traversal traverses “biz”, and attempts to find “buz.txt” which it does not find.

The fact that it does not find a resource related to “buz.txt” at this point does not signify an error condition. It signifies that:

- the *context* is the “biz” resource (the context is the last resource found during traversal).
- the *view name* is “buz.txt”
- the *subpath* is an empty sequence (()).

At this point, traversal has ended, and *view lookup* begins.

Because it’s the “context” resource, the view lookup machinery examines the “biz” resource to find out what “type” it is. Let’s say it finds that the resource is a `Biz` type (because “biz” is an instance of the Python class `Biz`). Using the *view name* (`buz.txt`) and the type, view lookup asks the *application registry* this question:

- Please find me a *view callable* registered with a *view configuration* with the name `buz.txt` that can be used for class `Biz`.

Let’s say that question is answered by the application registry; in such a situation, the application registry returns a *view callable*. The view callable is then called with the current *WebOb request* as the sole argument: `request`; it is expected to return a response.

The Example View Callables Accept Only a Request; How Do I Access the Context Resource?

Most of the examples in this book assume that a view callable is typically passed only a *request* object. Sometimes your view callables need access to the *context* resource, especially when you use *traversal*. You might use a supported alternate view callable argument list in your view callables such as the `(context, request)` calling convention described in *Alternate View Callable Argument/Calling Conventions*. But you don't need to if you don't want to. In view callables that accept only a request, the *context* resource found by traversal is available as the `context` attribute of the request object, e.g. `request.context`. The *view name* is available as the `view_name` attribute of the request object, e.g. `request.view_name`. Other Pyramid -specific request attributes are also available as described in *Special Attributes Added to the Request by Pyramid*.

8.4 References

A tutorial showing how *traversal* can be used within a Pyramid application exists in *ZODB + Traversal Wiki Tutorial*.

See the *View Configuration* chapter for detailed information about *view lookup*.

The `pyramid.traversal` module contains API functions that deal with traversal, such as traversal invocation from within application code.

The `pyramid.url.resource_url()` function generates a URL when given a resource retrieved from a resource tree.

VIEWS

One of the primary jobs of Pyramid is to find and invoke a *view callable* when a *request* reaches your application. View callables are bits of code which do something interesting in response to a request made to your application.



A Pyramid *view callable* is often referred to in conversational shorthand as a *view*. In this documentation, however, we need to use less ambiguous terminology because there are significant differences between *view configuration*, the code that implements a *view callable*, and the process of *view lookup*.

The *URL Dispatch*, and *Traversal* chapters describes how, using information from the *request*, a *context* resource is computed. But the context resource itself isn't very useful without an associated *view callable*. A view callable returns a response to a user, often using the context resource to do so.

The job of actually locating and invoking the “best” *view callable* is the job of the *view lookup* subsystem. The view lookup subsystem compares the resource supplied by *resource location* and information in the *request* against *view configuration* statements made by the developer to choose the most appropriate view callable for a specific set of circumstances.

This chapter describes how view callables work. In the *View Configuration* chapter, there are details about performing view configuration, and a detailed explanation of view lookup.

9.1 View Callables

View callables are, at the risk of sounding obvious, callable Python objects. Specifically, view callables can be functions, classes, or instances that implement an `__call__` method (making the instance callable).

View callables must, at a minimum, accept a single argument named `request`. This argument represents a Pyramid *Request* object. A request object encapsulates a WSGI environment provided to Pyramid by the upstream *WSGI* server. As you might expect, the request object contains everything your application needs to know about the specific HTTP request being made.

A view callable's ultimate responsibility is to create a Pyramid *Response* object. This can be done by creating the response object in the view callable code and returning it directly, as we will be doing in this chapter. However, if a view callable does not return a response itself, it can be configured to use a *renderer* that converts its return value into a *Response* object. Using renderers is the common way that templates are used with view callables to generate markup. See the *Renderers* chapter for details.

9.2 Defining a View Callable as a Function

One of the easiest way to define a view callable is to create a function that accepts a single argument named `request`, and which returns a *Response* object. For example, this is a “hello world” view callable implemented as a function:

```
1 from pyramid.response import Response
2
3 def hello_world(request):
4     return Response('Hello world!')
```

9.3 Defining a View Callable as a Class

A view callable may also be represented by a Python class instead of a function. When a view callable is a class, the calling semantics are slightly different than when it is a function or another non-class callable. When a view callable is a class, the class' `__init__` is called with a `request` parameter. As a result, an instance of the class is created. Subsequently, that instance's `__call__` method is invoked with no parameters. Views defined as classes must have the following traits:

- an `__init__` method that accepts a `request` argument.

- a `__call__` (or other) method that accepts no parameters and which returns a response.

For example:

```

1 from pyramid.response import Response
2
3 class MyView(object):
4     def __init__(self, request):
5         self.request = request
6
7     def __call__(self):
8         return Response('hello')
```

The request object passed to `__init__` is the same type of request object described in *Defining a View Callable as a Function*.

If you'd like to use a different attribute than `__call__` to represent the method expected to return a response, you can either:

- use an `attr` value as part of the configuration for the view. See *View Configuration Parameters*. The same view callable class can be used in different view configuration statements with different `attr` values, each pointing at a different method of the class if you'd like the class to represent a collection of related view callables.



A package named *pyramid_handlers* (available from PyPI) provides an analogue of *Pylons* -style “controllers”, which are a special kind of view class which provides more automation when your application uses *URL dispatch* solely.

9.4 Alternate View Callable Argument/Calling Conventions

Usually, view callables are defined to accept only a single argument: `request`. However, view callables may alternately be defined as classes, functions, or any callable that accept *two* positional arguments: a `context` resource as the first argument and a `request` as the second argument.

The `context` and `request` arguments passed to a view function defined in this style can be defined as follows:

context

9. VIEWS

The *resource* object found via tree *traversal* or *URL dispatch*.

request A Pyramid Request object representing the current WSGI request.

The following types work as view callables in this style:

1. Functions that accept two arguments: context, and request, e.g.:

```
1 from pyramid.response import Response
2
3 def view(context, request):
4     return Response('OK')
```

2. Classes that have an `__init__` method that accepts context, request and a `__call__` which accepts no arguments, e.g.:

```
1 from pyramid.response import Response
2
3 class view(object):
4     def __init__(self, context, request):
5         self.context = context
6         self.request = request
7
8     def __call__(self):
9         return Response('OK')
```

3. Arbitrary callables that have a `__call__` method that accepts context, request, e.g.:

```
1 from pyramid.response import Response
2
3 class View(object):
4     def __call__(self, context, request):
5         return Response('OK')
6 view = View() # this is the view callable
```

This style of calling convention is most useful for *traversal* based applications, where the context object is frequently used within the view callable code itself.

No matter which view calling convention is used, the view code always has access to the context via `request.context`.

9.5 View Callable Responses

A view callable may always return an object that implements the Pyramid *Response* interface. The easiest way to return something that implements the *Response* interface is to return a `pyramid.response.Response` object instance directly. For example:

```
1 from pyramid.response import Response
2
3 def view(request):
4     return Response('OK')
```

You don't need to always use `Response` to represent a response. Pyramid provides a range of different “exception” classes which can act as response objects too. For example, an instance of the class `pyramid.httpexceptions.HTTPFound` is also a valid response object (see *Using a View Callable to Do an HTTP Redirect*). A view can actually return any object that has the following attributes.

status The HTTP status code (including the name) for the response as a string. E.g. `200 OK` or `401 Unauthorized`.

headerlist A sequence of tuples representing the list of headers that should be set in the response. E.g. `[('Content-Type', 'text/html'), ('Content-Length', '412')]`

app_iter An iterable representing the body of the response. This can be a list, e.g. `['<html><head></head><body>Hello world!</body></html>']` or it can be a file-like object, or any other sort of iterable.

These attributes form the notional “Pyramid Response interface”.

9.6 Using a View Callable to Do an HTTP Redirect

You can issue an HTTP redirect from within a view by returning a particular kind of response.

```
1 from pyramid.httpexceptions import HTTPFound
2
3 def myview(request):
4     return HTTPFound(location='http://example.com')
```

All exception types from the `pyramid.httpexceptions` module implement the *Response* interface; any can be returned as the response from a view. See `pyramid.httpexceptions` for the documentation for the `HTTPFound` exception; it also includes other response types that imply other HTTP response codes, such as `HTTPUnauthorized` for 401 `Unauthorized`.



Although exception types from the `pyramid.httpexceptions` module are in fact bona fide Python `Exception` types, the Pyramid view machinery expects them to be *returned* by a view callable rather than *raised*.

It is possible, however, in Python 2.5 and above, to configure an *exception view* to catch these exceptions, and return an appropriate *Response*. The simplest such view could just catch and return the original exception. See *Exception Views* for more details.

9.7 Using Special Exceptions In View Callables

Usually when a Python exception is raised within a view callable, Pyramid allows the exception to propagate all the way out to the *WSGI* server which invoked the application.

However, for convenience, two special exceptions exist which are always handled by Pyramid itself. These are `pyramid.exceptions.NotFound` and `pyramid.exceptions.Forbidden`. Both are exception classes which accept a single positional constructor argument: a message.

If `NotFound` is raised within view code, the result of the *Not Found View* will be returned to the user agent which performed the request.

If `Forbidden` is raised within view code, the result of the *Forbidden View* will be returned to the user agent which performed the request.

In all cases, the message provided to the exception constructor is made available to the view which Pyramid invokes as `request.exception.args[0]`.

9.8 Exception Views

The machinery which allows the special `NotFound` and `Forbidden` exceptions to be caught by specialized views as described in *Using Special Exceptions In View Callables* can also be used by application developers to convert arbitrary exceptions to responses.

To register a view that should be called whenever a particular exception is raised from with Pyramid view code, use the exception class or one of its superclasses as the `context` of a view configuration which points at a view callable you'd like to generate a response.

For example, given the following exception class in a module named `helloworld.exceptions`:

```

1 class ValidationFailure(Exception):
2     def __init__(self, msg):
3         self.msg = msg

```

You can wire a view callable to be called whenever any of your *other* code raises a `helloworld.exceptions.ValidationFailure` exception:

```

1 from helloworld.exceptions import ValidationFailure
2
3 @view_config(context=ValidationFailure)
4 def failed_validation(exc, request):
5     response = Response('Failed validation: %s' % exc.msg)
6     response.status_int = 500
7     return response

```

Assuming that a *scan* was run to pick up this view registration, this view callable will be invoked whenever a `helloworld.exceptions.ValidationFailure` is raised by your application’s view code. The same exception raised by a custom root factory or a custom traverser is also caught and hooked.

Other normal view predicates can also be used in combination with an exception view registration:

```

1 from pyramid.view import view_config
2 from pyramid.exceptions import NotFound
3 from pyramid.httpexceptions import HTTPNotFound
4
5 @view_config(context=NotFound, route_name='home')
6 def notfound_view(request):
7     return HTTPNotFound()

```

The above exception view names the `route_name` of `home`, meaning that it will only be called when the route matched has a name of `home`. You can therefore have more than one exception view for any given exception in the system: the “most specific” one will be called when the set of request circumstances match the view registration.

The only view predicate that cannot be used successfully when creating an exception view configuration is `name`. The name used to look up an exception view is always the empty string. Views registered as exception views which have a name will be ignored.



Normal (i.e., non-exception) views registered against a context resource type which inherits from `Exception` will work normally. When an exception view configuration is processed, *two* views are registered. One as a “normal” view, the other as an “exception” view. This means that you can use an exception as `context` for a normal view.

Exception views can be configured with any view registration mechanism: `@view_config` decorator, ZCML, or imperative `add_view` styles.

9.9 Handling Form Submissions in View Callables (Unicode and Character Set Issues)

Most web applications need to accept form submissions from web browsers and various other clients. In Pyramid, form submission handling logic is always part of a *view*. For a general overview of how to handle form submission data using the *WebOb* API, see *Request and Response Objects* and “Query and POST variables” within the *WebOb* documentation. Pyramid defers to *WebOb* for its request and response implementations, and handling form submission data is a property of the request implementation. Understanding *WebOb*’s request API is the key to understanding how to process form submission data.

There are some defaults that you need to be aware of when trying to handle form submission data in a Pyramid view. Having high-order (i.e., non-ASCII) characters in data contained within form submissions is exceedingly common, and the UTF-8 encoding is the most common encoding used on the web for character data. Since Unicode values are much saner than working with and storing bytestrings, Pyramid configures the *WebOb* request machinery to attempt to decode form submission values into Unicode from UTF-8 implicitly. This implicit decoding happens when view code obtains form field values via the `request.params`, `request.GET`, or `request.POST` APIs (see *pyramid.request* for details about these APIs).



Many people find the difference between Unicode and UTF-8 confusing. Unicode is a standard for representing text that supports most of the world’s writing systems. However, there are many ways that Unicode data can be encoded into bytes for transit and storage. UTF-8 is a specific encoding for Unicode, that is backwards-compatible with ASCII. This makes UTF-8 very convenient for encoding data where a large subset of that data is ASCII characters, which is largely true on the web. UTF-8 is also the standard character encoding for URLs.

As an example, let’s assume that the following form page is served up to a browser client, and its `action` points at some Pyramid view code:

```
1 <html xmlns="http://www.w3.org/1999/xhtml">
2   <head>
3     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
4   </head>
5   <form method="POST" action="myview">
6     <div>
7       <input type="text" name="firstname"/>
```

```

8     </div>
9     <div>
10        <input type="text" name="lastname" />
11    </div>
12    <input type="submit" value="Submit" />
13 </form>
14 </html>

```

The `myview` view code in the Pyramid application *must* expect that the values returned by `request.params` will be of type `unicode`, as opposed to type `str`. The following will work to accept a form post from the above form:

```

1 def myview(request):
2     firstname = request.params['firstname']
3     lastname = request.params['lastname']

```

But the following `myview` view code *may not* work, as it tries to decode already-decoded (`unicode`) values obtained from `request.params`:

```

1 def myview(request):
2     # the .decode('utf-8') will break below if there are any high-order
3     # characters in the firstname or lastname
4     firstname = request.params['firstname'].decode('utf-8')
5     lastname = request.params['lastname'].decode('utf-8')


```

For implicit decoding to work reliably, you should ensure that every form you render that posts to a Pyramid view explicitly defines a charset encoding of UTF-8. This can be done via a response that has a `; charset=UTF-8` in its `Content-Type` header; or, as in the form above, with a `meta http-equiv` tag that implies that the charset is UTF-8 within the HTML head of the page containing the form. This must be done explicitly because all known browser clients assume that they should encode form data in the same character set implied by `Content-Type` value of the response containing the form when subsequently submitting that form. There is no other generally accepted way to tell browser clients which charset to use to encode form data. If you do not specify an encoding explicitly, the browser client will choose to encode form data in its default character set before submitting it, which may not be UTF-8 as the server expects. If a request containing form data encoded in a non-UTF8 charset is handled by your view code, eventually the request code accessed within your view will throw an error when it can't decode some high-order character encoded in another character set within form data, e.g., when `request.params['somename']` is accessed.

If you are using the `Response` class to generate a response, or if you use the `render_template_*` templating APIs, the UTF-8 charset is set automatically as the default via the `Content-Type` header. If you return a `Content-Type` header without an explicit charset, a request will add a

9. VIEWS

`; charset=utf-8` trailer to the `Content-Type` header value for you, for response content types that are textual (e.g. `text/html`, `application/xml`, etc) as it is rendered. If you are using your own response object, you will need to ensure you do this yourself.

 Only the *values* of request params obtained via `request.params`, `request.GET` or `request.POST` are decoded to Unicode objects implicitly in the Pyramid default configuration. The keys are still (byte) strings.

RENDERERS

A view needn't *always* return a *Response* object. If a view happens to return something which does not implement the Pyramid Response interface, Pyramid will attempt to use a *renderer* to construct a response. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(renderer='json')
5 def hello_world(request):
6     return {'content': 'Hello!' }
```

The above example returns a *dictionary* from the view callable. A dictionary does not implement the Pyramid response interface, so you might believe that this example would fail. However, since a *renderer* is associated with the view callable through its *view configuration* (in this case, using a *renderer* argument passed to `view_config()`), if the view does *not* return a Response object, the renderer will attempt to convert the result of the view to a response on the developer's behalf.

Of course, if no renderer is associated with a view's configuration, returning anything except an object which implements the Response interface will result in an error. And, if a renderer *is* used, whatever is returned by the view must be compatible with the particular kind of renderer used, or an error may occur during view invocation.

One exception exists: it is *always* OK to return a Response object, even when a *renderer* is configured. If a view callable returns a response object from a view that is configured with a *renderer*, the *renderer* is bypassed entirely.

Various types of renderers exist, including serialization renderers and renderers which use templating systems. See also *Writing View Callables Which Use a Renderer*.

10.1 Writing View Callables Which Use a Renderer

As we've seen, view callables needn't always return a `Response` object. Instead, they may return an arbitrary Python object, with the expectation that a *renderer* will convert that object into a response instance on your behalf. Some renderers use a templating system; other renderers use object serialization techniques.

View configuration can vary the renderer associated with a view callable via the `renderer` attribute. For example, this call to `add_view()` associates the `json` renderer with a view callable:

```
1 config.add_view('myproject.views.my_view', renderer='json')
```

When this configuration is added to an application, the `myproject.views.my_view` view callable will now use a `json` renderer, which renders view return values to a *JSON* response serialization.

Other built-in renderers include renderers which use the *Chameleon* templating language to render a dictionary to a response.

If the *view callable* associated with a *view configuration* returns a `Response` object directly (an object with the attributes `status`, `headerlist` and `app_iter`), any renderer associated with the view configuration is ignored, and the response is passed back to Pyramid unmolested. For example, if your view callable returns an instance of the `pyramid.httpexceptions.HTTPFound` class as a response, no renderer will be employed.

```
1 from pyramid.httpexceptions import HTTPFound
2
3 def view(request):
4     return HTTPFound(location='http://example.com') # any renderer avoided
```

Views which use a renderer can vary non-body response attributes (such as headers and the HTTP status code) by attaching properties to the request. See *Varying Attributes of Rendered Responses*.

Additional renderers can be added by developers to the system as necessary (see *Adding and Changing Renderers*).

10.2 Built-In Renderers

Several built-in renderers exist in Pyramid. These renderers can be used in the `renderer` attribute of view configurations.

10.2.1 string: String Renderer

The `string` renderer is a renderer which renders a view callable result to a string. If a view callable returns a non-Response object, and the `string` renderer is associated in that view's configuration, the result will be to run the object through the Python `str` function to generate a string. Note that if a Unicode object is returned by the view callable, it is not `str()`-ified.

Here's an example of a view that returns a dictionary. If the `string` renderer is specified in the configuration for this view, the view will render the returned dictionary to the `str()` representation of the dictionary:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(renderer='string')
5 def hello_world(request):
6     return {'content': 'Hello!'}
```

The body of the response returned by such a view will be a string representing the `str()` serialization of the return value:

```
1 {'content': 'Hello!'}
```

Views which use the `string` renderer can vary non-body response attributes by attaching properties to the request. See *Varying Attributes of Rendered Responses*.

10.2.2 json: JSON Renderer

The `json` renderer renders view callable results to *JSON*. It passes the return value through the `json.dumps` standard library function, and wraps the result in a response object. It also sets the response content-type to `application/json`.

Here's an example of a view that returns a dictionary. Since the `json` renderer is specified in the configuration for this view, the view will render the returned dictionary to a JSON serialization:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(renderer='json')
5 def hello_world(request):
6     return {'content': 'Hello!'}
```

10. RENDERERS

The body of the response returned by such a view will be a string representing the JSON serialization of the return value:

```
1 '{"content": "Hello!"}'
```

The return value needn't be a dictionary, but the return value must contain values serializable by `json.dumps()`.

You can configure a view to use the JSON renderer by naming `json` as the `renderer` argument of a view configuration, e.g. by using `add_view()`:

```
1 config.add_view('myproject.views.hello_world',
2                 name='hello',
3                 context='myproject.resources.Hello',
4                 renderer='json')
```

Views which use the JSON renderer can vary non-body response attributes by attaching properties to the request. See *Varying Attributes of Rendered Responses*.

10.2.3 *.pt or *.txt: Chameleon Template Renderers

Two built-in renderers exist for *Chameleon* templates.

If the `renderer` attribute of a view configuration is an absolute path, a relative path or *asset specification* which has a final path element with a filename extension of `.pt`, the Chameleon ZPT renderer is used. See *Chameleon ZPT Templates* for more information about ZPT templates.

If the `renderer` attribute of a view configuration is an absolute path or a *asset specification* which has a final path element with a filename extension of `.txt`, the *Chameleon* text renderer is used. See *Chameleon ZPT Templates* for more information about Chameleon text templates.

The behavior of these renderers is the same, except for the engine used to render the template.

When a `renderer` attribute that names a template path or *asset specification* (e.g. `myproject:templates/foo.pt` or `myproject:templates/foo.txt`) is used, the view must return a *Response* object or a Python *dictionary*. If the view callable with an associated template returns a Python dictionary, the named template will be passed the dictionary as its keyword arguments, and the template renderer implementation will return the resulting rendered template in a response to the user. If the view callable returns anything but a *Response* object or a dictionary, an error will be raised.

Before passing keywords to the template, the keyword arguments derived from the dictionary returned by the view are augmented. The callable object – whatever object was used to define the `view` – will be automatically inserted into the set of keyword arguments passed to the template as the `view` keyword. If the view callable was a class, the `view` keyword will be an instance of that class. Also inserted into the keywords passed to the template are `renderer_name` (the string used in the `renderer` attribute of the directive), `renderer_info` (an object containing renderer-related information), `context` (the context resource of the view used to render the template), and `request` (the request passed to the view used to render the template).

Here’s an example view configuration which uses a Chameleon ZPT renderer:

```

1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('myproject.views.hello_world',
4                 name='hello',
5                 context='myproject.resources.Hello',
6                 renderer='myproject:templates/foo.pt')
```

Here’s an example view configuration which uses a Chameleon text renderer:

```

1 config.add_view('myproject.views.hello_world',
2                 name='hello',
3                 context='myproject.resources.Hello',
4                 renderer='myproject:templates/foo.txt')
```

Views which use a Chameleon renderer can vary response attributes by attaching properties to the request. See *Varying Attributes of Rendered Responses*.

10.2.4 * .mak or * .mako: Mako Template Renderer

The Mako template renderer renders views using a Mako template. When used, the view must return a Response object or a Python *dictionary*. The dictionary items will then be used in the global template space. If the view callable returns anything but a Response object, or a dictionary, an error will be raised.

When using a `renderer` argument to a *view configuration* to specify a Mako template, the value of the `renderer` may be a path relative to the `mako.directories` setting (e.g. `some/template.mak`) or, alternately, it may be a *asset specification* (e.g. `apackage:templates/sometemplate.mak`). Mako templates may internally inherit other Mako templates using a relative filename or a *asset specification* as desired.

Here’s an example view configuration which uses a relative path:

10. RENDERERS

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('myproject.views.hello_world',
4                 name='hello',
5                 context='myproject.resources.Hello',
6                 renderer='foo.mak')
```

It's important to note that in Mako's case, the 'relative' path name `foo.mak` above is not relative to the package, but is relative to the directory (or directories) configured for Mako via the `mako.directories` configuration file setting.

The renderer can also be provided in *asset specification* format. Here's an example view configuration which uses one:

```
1 config.add_view('myproject.views.hello_world',
2                 name='hello',
3                 context='myproject.resources.Hello',
4                 renderer='mypackage:templates/foo.mak')
```

The above configuration will use the file named `foo.mak` in the `templates` directory of the `mypackage` package.

The Mako template renderer can take additional arguments beyond the standard `reload_templates` setting, see the *Environment Variables and .ini File Settings* for additional *Mako Template Render Settings*.

10.3 Varying Attributes of Rendered Responses

Before a response constructed by a *renderer* is returned to Pyramid, several attributes of the request are examined which have the potential to influence response behavior.

View callables that don't directly return a response should set these attributes on the `request` object via `setattr` during their execution, to influence associated response attributes.

response_content_type Defines the content-type of the resulting response, e.g. `text/xml`.

response_headerlist A sequence of tuples describing cookie values that should be set in the response, e.g. `[('Set-Cookie', 'abc=123'), ('X-My-Header', 'foo')]`.

response_status A WSGI-style status code (e.g. `200 OK`) describing the status of the response.

response_charset The character set (e.g. UTF-8) of the response.

response_cache_for A value in seconds which will influence `Cache-Control` and `Expires` headers in the returned response. The same can also be achieved by returning various values in the `response_headerlist`, this is purely a convenience.

For example, if you need to change the response status from within a view callable that uses a renderer, assign the `response_status` attribute to the request before returning a result:

```

1 from pyramid.view import view_config
2
3 @view_config(name='gone', renderer='templates/gone.pt')
4 def myview(request):
5     request.response_status = '404 Not Found'
6     return {'URL':request.URL}

```

For more information on attributes of the request, see the API documentation in *pyramid.request*.

10.4 Adding and Changing Renderers

New templating systems and serializers can be associated with Pyramid renderer names. To this end, configuration declarations can be made which change an existing *renderer factory*, and which add a new renderer factory.

Renderers can be registered imperatively using the `pyramid.config.Configurator.add_renderer()` API.

For example, to add a renderer which renders views which have a `renderer` attribute that is a path that ends in `.jinja2`:

```

1 config.add_renderer('.jinja2', 'mypackage.MyJinja2Renderer')

```

The first argument is the renderer name. The second argument is a reference to an implementation of a *renderer factory* or a *dotted Python name* referring to such an object.

10.4.1 Adding a New Renderer

You may add a new renderer by creating and registering a *renderer factory*.

A renderer factory implementation is typically a class with the following interface:

10. RENDERERS

```
1 class RendererFactory:
2     def __init__(self, info):
3         """ Constructor: info will be an object having the the
4           following attributes: name (the renderer name), package
5           (the package that was 'current' at the time the
6           renderer was registered), type (the renderer type
7           name), registry (the current application registry) and
8           settings (the deployment settings dictionary). """
9
10    def __call__(self, value, system):
11        """ Call a the renderer implementation with the value
12          and the system value passed in as arguments and return
13          the result (a string or unicode object). The value is
14          the return value of a view. The system value is a
15          dictionary containing available system values
16          (e.g. view, context, and request). """
```

The formal interface definition of the `info` object passed to a renderer factory constructor is available as `pyramid.interfaces.IRendererInfo`.

There are essentially two different kinds of renderer factories:

- A renderer factory which expects to accept a *asset specification*, or an absolute path, as the `name` attribute of the `info` object fed to its constructor. These renderer factories are registered with a `name` value that begins with a dot (`.`). These types of renderer factories usually relate to a file on the filesystem, such as a template.
- A renderer factory which expects to accept a token that does not represent a filesystem path or an asset specification in the `name` attribute of the `info` object fed to its constructor. These renderer factories are registered with a `name` value that does not begin with a dot. These renderer factories are typically object serializers.

Asset Specifications

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

Here's an example of the registration of a simple renderer factory via `add_renderer()`:


```

1 # config is an instance of pyramid.config.Configurator
2
3 config.add_renderer(name='amf', factory='my.package.MyAMFRenderer')

```

Adding the above code to your application startup configuration will allow you to use the `my.package.MyAMFRenderer` renderer factory implementation in view configurations. Your application can use this renderer by specifying `amf` in the `renderer` attribute of a *view configuration*:

```

1 from pyramid.view import view_config
2
3 @view_config(renderer='amf')
4 def myview(request):
5     return {'Hello': 'world'}

```

At startup time, when a *view configuration* is encountered, which has a `name` attribute that does not contain a dot, the full name value is used to construct a renderer from the associated renderer factory. In this case, the view configuration will create an instance of an `AMFRenderer` for each view configuration which includes `amf` as its `renderer` value. The name passed to the `AMFRenderer` constructor will always be `amf`.

Here's an example of the registration of a more complicated renderer factory, which expects to be passed a filesystem path:

```

1 config.add_renderer(name='.jinja2',
2                    factory='my.package.MyJinja2Renderer')

```

Adding the above code to your application startup will allow you to use the `my.package.MyJinja2Renderer` renderer factory implementation in view configurations by referring to any renderer which *ends in* `.jinja` in the `renderer` attribute of a *view configuration*:

```

1 from pyramid.view import view_config
2
3 @view_config(renderer='templates/mytemplate.jinja2')
4 def myview(request):
5     return {'Hello': 'world'}

```

When a *view configuration* is encountered at startup time, which has a `name` attribute that does contain a dot, the value of the `name` attribute is split on its final dot. The second element of the split is typically the filename extension. This extension is used to look up a renderer factory for the configured view. Then the value of `renderer` is passed to the factory to create a renderer for the view. In this case, the view configuration will create an instance of a `Jinja2Renderer` for each view configuration which includes anything ending with `.jinja2` in its `renderer` value. The name passed to the `Jinja2Renderer` constructor will be the full value that was set as `renderer=` in the view configuration.

10.4.2 Changing an Existing Renderer

You can associate more than one filename extension with the same existing renderer implementation as necessary if you need to use a different file extension for the same kinds of templates. For example, to associate the `.zpt` extension with the Chameleon ZPT renderer factory, use the `pyramid.config.Configurator.add_renderer()` method:

```
1 config.add_renderer('.zpt', 'pyramid.chameleon_zpt.renderer_factory')
```

After you do this, Pyramid will treat templates ending in both the `.pt` and `.zpt` filename extensions as Chameleon ZPT templates.

To change the default mapping in which files with a `.pt` extension are rendered via a Chameleon ZPT page template renderer, use a variation on the following in your application's startup code:

```
1 config.add_renderer('.pt', 'mypackage.pt_renderer')
```

After you do this, the *renderer factory* in `mypackage.pt_renderer` will be used to render templates which end in `.pt`, replacing the default Chameleon ZPT renderer.

To associate a *default* renderer with *all* view configurations (even ones which do not possess a `renderer` attribute), pass `None` as the name attribute to the renderer tag:

```
1 config.add_renderer(None, 'mypackage.json_renderer_factory')
```

10.5 Overriding A Renderer At Runtime



This is an advanced feature, not typically used by “civilians”.

In some circumstances, it is necessary to instruct the system to ignore the static renderer declaration provided by the developer in view configuration, replacing the renderer with another *after a request starts*. For example, an “omnipresent” XML-RPC implementation that detects that the request is from an XML-RPC client might override a view configuration statement made by the user instructing the view to use a template renderer with one that uses an XML-RPC renderer. This renderer would produce an XML-RPC representation of the data returned by an arbitrary view callable.

To use this feature, create a `NewRequest subscriber` which sniffs at the request data and which conditionally sets an `override_renderer` attribute on the request itself, which is the *name* of a registered renderer. For example:

```
1 from pyramid.event import subscriber
2 from pyramid.event import NewRequest
3
4 @subscriber(NewRequest)
5 def set_xmlrpc_params(event):
6     request = event.request
7     if (request.content_type == 'text/xml'
8         and request.method == 'POST'
9         and not 'soapaction' in request.headers
10        and not 'x-pyramid-avoid-xmlrpc' in request.headers):
11        params, method = parse_xmlrpc_request(request)
12        request.xmlrpc_params, request.xmlrpc_method = params, method
13        request.is_xmlrpc = True
14        request.override_renderer = 'xmlrpc'
15    return True
```

The result of such a subscriber will be to replace any existing static renderer configured by the developer with a (notional, nonexistent) XML-RPC renderer if the request appears to come from an XML-RPC client.

10. RENDERERS

TEMPLATES

A *template* is a file on disk which can be used to render dynamic data provided by a *view*. Pyramid offers a number of ways to perform templating tasks out of the box, and provides add-on templating support through a set of bindings packages.

Out of the box, Pyramid provides templating via the *Chameleon* and *Mako* templating libraries. *Chameleon* provides support for two different types of templates: *ZPT* templates, and text templates.

Before discussing how built-in templates are used in detail, we'll discuss two ways to render templates within Pyramid in general: directly, and via renderer configuration.

11.1 Using Templates Directly

The most straightforward way to use a template within Pyramid is to cause it to be rendered directly within a *view callable*. You may use whatever API is supplied by a given templating engine to do so.

Pyramid provides various APIs that allow you to render templates directly from within a view callable. For example, if there is a *Chameleon* ZPT template named `foo.pt` in a directory named `templates` in your application, you can render the template from within the body of a view callable like so:

```
1 from pyramid.renderers import render_to_response
2
3 def sample_view(request):
4     return render_to_response('templates/foo.pt',
5                               {'foo':1, 'bar':2},
6                               request=request)
```

11. TEMPLATES



Earlier iterations of this documentation (pre-version-1.3) encouraged the application developer to use ZPT-specific APIs such as `pyramid.chameleon_zpt.render_template_to_response()` and `pyramid.chameleon_zpt.render_template()` to render templates directly. This style of rendering still works, but at least for purposes of this documentation, those functions are deprecated. Application developers are encouraged instead to use the functions available in the `pyramid.renderers` module to perform rendering tasks. This set of functions works to render templates for all renderer extensions registered with Pyramid.

The `sample_view` *view callable* function above returns a *response* object which contains the body of the `templates/foo.pt` template. In this case, the `templates` directory should live in the same directory as the module containing the `sample_view` function. The template author will have the names `foo` and `bar` available as top-level names for replacement or comparison purposes.

In the example above, the path `templates/foo.pt` is relative to the directory containing the file which defines the view configuration. In this case, this is the directory containing the file that defines the `sample_view` function. Although a renderer path is usually just a simple relative pathname, a path named as a renderer can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows.



Only *Chameleon* templates support defining a renderer for a template relative to the location of the module where the view callable is defined. Mako templates, and other templating system bindings work differently. In particular, Mako templates use a “lookup path” as defined by the `mako.directories` configuration file instead of treating relative paths as relative to the current view module. See *Templating With Mako Templates*.

The path can alternately be a *asset specification* in the form `some.dotted.package_name:relative/path`. This makes it possible to address template assets which live in another package. For example:

```
1 from pyramid.renderers import render_to_response
2
3 def sample_view(request):
4     return render_to_response('mypackage:templates/foo.pt',
5                               {'foo':1, 'bar':2},
6                               request=request)
```

An asset specification points at a file within a Python *package*. In this case, it points at a file named `foo.pt` within the `templates` directory of the `mypackage` package. Using an asset specification instead of a relative template name is usually a good idea, because calls to `render_to_response` using asset specifications will continue to work properly if you move the code containing them around.

i Mako templating system bindings also respect absolute asset specifications as an argument to any of the `render*` commands. If a template name defines a `:` (colon) character and is not an absolute path, it is treated as an absolute asset specification.

In the examples above we pass in a keyword argument named `request` representing the current Pyramid request. Passing a request keyword argument will cause the `render_to_response` function to supply the renderer with more correct system values (see *System Values Used During Rendering*), because most of the information required to compose proper system values is present in the request. If your template relies on the name `request` or `context`, or if you've configured special *renderer globals*, make sure to pass `request` as a keyword argument in every call to a `pyramid.renderers.render_*` function.

Every view must return a *response* object, except for views which use a *renderer* named via view configuration (which we'll see shortly). The `pyramid.renderers.render_to_response()` function is a shortcut function that actually returns a response object. This allows the example view above to simply return the result of its call to `render_to_response()` directly.

Obviously not all APIs you might call to get response data will return a response object. For example, you might render one or more templates to a string that you want to use as response data. The `pyramid.renderers.render()` API renders a template to a string. We can manufacture a *response* object directly, and use that string as the body of the response:

```
1 from pyramid.renderers import render
2 from pyramid.response import Response
3
4 def sample_view(request):
5     result = render('mypackage:templates/foo.pt',
6                   {'foo':1, 'bar':2},
7                   request=request)
8     response = Response(result)
9     return response
```

Because *view callable* functions are typically the only code in Pyramid that need to know anything about templates, and because view functions are very simple Python, you can use whatever templating system you're most comfortable with within Pyramid. Install the templating system, import its API functions into your views module, use those APIs to generate a string, then return that string as the body of a Pyramid *Response* object.

For example, here's an example of using "raw" Mako from within a Pyramid *view*:

11. TEMPLATES

```
1 from mako.template import Template
2 from pyramid.response import Response
3
4 def make_view(request):
5     template = Template(filename='/templates/template.mak')
6     result = template.render(name=request.params['name'])
7     response = Response(result)
8     return response
```

You probably wouldn't use this particular snippet in a project, because it's easier to use the Mako renderer bindings which already exist in Pyramid. But if your favorite templating system is not supported as a renderer extension for Pyramid, you can create your own simple combination as shown above.



If you use third-party templating languages without cooperating Pyramid bindings directly within view callables, the auto-template-reload strategy explained in *Automatically Reloading Templates* will not be available, nor will the template asset overriding capability explained in *Overriding Assets* be available, nor will it be possible to use any template using that language as a *renderer*. However, it's reasonably easy to write custom templating system binding packages for use under Pyramid so that templates written in the language can be used as renderers. See *Adding and Changing Renderers* for instructions on how to create your own template renderer and *Available Add-On Template System Bindings* for example packages.

If you need more control over the status code and content-type, or other response attributes from views that use direct templating, you may set attributes on the response that influence these values.

Here's an example of changing the content-type and status of the response object returned by `render_to_response()`:

```
1 from pyramid.renderers import render_to_response
2
3 def sample_view(request):
4     response = render_to_response('templates/foo.pt',
5                                 {'foo':1, 'bar':2},
6                                 request=request)
7     response.content_type = 'text/plain'
8     response.status_int = 204
9     return response
```

Here's an example of manufacturing a response object using the result of `render()` (a string):


```
1 from pyramid.renderers import render
2 from pyramid.response import Response
3
4 def sample_view(request):
5     result = render('mypackage:templates/foo.pt',
6                   {'foo':1, 'bar':2},
7                   request=request)
8     response = Response(result)
9     response.content_type = 'text/plain'
10    return response
```

11.2 System Values Used During Rendering

When a template is rendered using `render_to_response()` or `render()`, the renderer representing the template will be provided with a number of *system* values. These values are provided in a dictionary to the renderer and include:

context The current Pyramid context if `request` was provided as a keyword argument, or `None`.

request The request provided as a keyword argument.

renderer_name The renderer name used to perform the rendering, e.g. `mypackage:templates/foo.pt`.

renderer_info An object implementing the `pyramid.interfaces.IRendererInfo` interface. Basically, an object with the following attributes: `name`, `package` and `type`.

You can define more values which will be passed to every template executed as a result of rendering by defining *renderer globals*.

What any particular renderer does with these system values is up to the renderer itself, but most template renderers, including Chameleon and Mako renderers, make these names available as top-level template variables.

11.3 Templates Used as Renderers via Configuration

An alternative to using `render_to_response()` to render templates manually in your view callable code, is to specify the template as a *renderer* in your *view configuration*. This can be done with any of the templating languages supported by Pyramid.

To use a renderer via view configuration, specify a template *asset specification* as the `renderer` argument, or attribute to the *view configuration* of a *view callable*. Then return a *dictionary* from that view callable. The dictionary items returned by the view callable will be made available to the renderer template as top-level names.

The association of a template as a renderer for a *view configuration* makes it possible to replace code within a *view callable* that handles the rendering of a template.

Here's an example of using a `view_config` decorator to specify a *view configuration* that names a template renderer:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='templates/foo.pt')
4 def my_view(request):
5     return {'foo':1, 'bar':2}
```



You do not need to supply the `request` value as a key in the dictionary result returned from a renderer-configured view callable. Pyramid automatically supplies this value for you so that the “most correct” system values are provided to the renderer.



The `renderer` argument to the `@view_config` configuration decorator shown above is the template *path*. In the example above, the path `templates/foo.pt` is *relative*. Relative to what, you ask? Because we're using a Chameleon renderer, it means “relative to the directory in which the file which defines the view configuration lives”. In this case, this is the directory containing the file that defines the `my_view` function. View-configuration-relative asset specifications work only in Chameleon, not in Mako templates.

Similar renderer configuration can be done imperatively and via *ZCML*. See *Writing View Callables Which Use a Renderer*. See also *Built-In Renderers*.

Although a renderer path is usually just a simple relative pathname, a path named as a renderer can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternately

be an *asset specification* in the form `some.dotted.package_name:relative/path`, making it possible to address template assets which live in another package.

Not just any template from any arbitrary templating system may be used as a renderer. Bindings must exist specifically for Pyramid to use a templating language template as a renderer. Currently, Pyramid has built-in support for two Chameleon templating languages: ZPT and text, and the Mako templating system. See *Built-In Renderers* for a discussion of their details. Pyramid also supports the use of *Jinja2* templates as renderers. See *Available Add-On Template System Bindings*.

Why Use A Renderer via View Configuration

Using a renderer in view configuration is usually a better way to render templates than using any rendering API directly from within a *view callable* because it makes the view callable more unit-testable. Views which use templating or rendering APIs directly must return a *Response* object. Making testing assertions about response objects is typically an indirect process, because it means that your test code often needs to somehow parse information out of the response body (often HTML). View callables configured with renderers externally via view configuration typically return a dictionary, as above. Making assertions about results returned in a dictionary is almost always more direct and straightforward than needing to parse HTML. Specifying a renderer from within *ZCML* (as opposed to imperatively or via a `view_config` decorator, or using a template directly from within a view callable) also makes it possible for someone to modify the template used to render a view without needing to fork your code to do so. See *Extending An Existing Pyramid Application* for more information.

By default, views rendered via a template renderer return a *Response* object which has a *status code* of 200 OK, and a *content-type* of `text/html`. To vary attributes of the response of a view that uses a renderer, such as the content-type, headers, or status attributes, you must set attributes on the *request* object within the view before returning the dictionary. See *Varying Attributes of Rendered Responses* for more information.

The same set of system values are provided to templates rendered via a renderer view configuration as those provided to templates rendered imperatively. See *System Values Used During Rendering*.

11.4 Chameleon ZPT Templates

Like *Zope*, Pyramid uses *ZPT* (Zope Page Templates) as its default templating language. However, Pyramid uses a different implementation of the *ZPT* specification than *Zope* does: the *Chameleon* templating engine. The Chameleon engine complies largely with the Zope Page Template template specification. However, it is significantly faster.

11. TEMPLATES

The language definition documentation for Chameleon ZPT-style templates is available from the Chameleon website.



Chameleon only works on CPython platforms and Google App Engine. On Jython and other non-CPython platforms, you should use Mako (see *Templating With Mako Templates*) or `pyramid_jinja2` instead. See *Available Add-On Template System Bindings*.

Given a *Chameleon* ZPT template named `foo.pt` in a directory in your application named `templates`, you can render the template as a *renderer* like so:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='templates/foo.pt')
4 def my_view(request):
5     return {'foo':1, 'bar':2}
```

See also *Built-In Renderers* for more general information about renderers, including Chameleon ZPT renderers.

11.4.1 A Sample ZPT Template

Here's what a simple *Chameleon* ZPT template used under Pyramid might look like:

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:tal="http://xml.zope.org/namespaces/tal">
5 <head>
6   <meta http-equiv="content-type" content="text/html; charset=utf-8" />
7   <title>${project} Application</title>
8 </head>
9 <body>
10  <h1 class="title">Welcome to <code>${project}</code>, an
11    application generated by the <a
12      href="http://docs.pyloproject.org/projects/pyramid/dev/"
13    >pyramid</a> web
14    application framework.</h1>
15 </body>
16 </html>
```

Note the use of *Genshi*-style `${replacements}` above. This is one of the ways that *Chameleon* ZPT differs from standard ZPT. The above template expects to find a `project` key in the set of keywords passed in to it via `render()` or `render_to_response()`. Typical ZPT attribute-based syntax (e.g. `tal:content` and `tal:replace`) also works in these templates.

11.4.2 Using ZPT Macros in Pyramid

When a *renderer* is used to render a template, Pyramid makes at least two top-level names available to the template by default: `context` and `request`. One of the common needs in ZPT-based templates is to use one template's "macros" from within a different template. In Zope, this is typically handled by retrieving the template from the `context`. But the context in Pyramid is a *resource* object, and templates cannot usually be retrieved from resources. To use macros in Pyramid, you need to make the macro template itself available to the rendered template by passing the macro template, or even the macro itself, into the rendered template. To do this you can use the `pyramid.renderers.get_renderer()` API to retrieve the macro template, and pass it into the template being rendered via the dictionary returned by the view. For example, using a *view configuration* via a `view_config` decorator that uses a *renderer*:

```

1 from pyramid.renderers import get_renderer
2 from pyramid.view import view_config
3
4 @view_config(renderer='templates/mytemplate.pt')
5 def my_view(request):
6     main = get_renderer('templates/master.pt').implementation()
7     return {'main':main}

```

Where `templates/master.pt` might look like so:

```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:tal="http://xml.zope.org/namespaces/tal"
3     xmlns:metal="http://xml.zope.org/namespaces/metal">
4   <span metal:define-macro="hello">
5     <h1>
6       Hello <span metal:define-slot="name">Fred</span>!
7     </h1>
8   </span>
9 </html>

```

And `templates/mytemplate.pt` might look like so:

```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:tal="http://xml.zope.org/namespaces/tal"
3     xmlns:metal="http://xml.zope.org/namespaces/metal">
4   <span metal:use-macro="main.macros['hello']">
5     <span metal:fill-slot="name">Chris</span>
6   </span>
7 </html>

```

11.5 Templating with *Chameleon* Text Templates

Pyramid also allows for the use of templates which are composed entirely of non-XML text via *Chameleon*. To do so, you can create templates that are entirely composed of text except for `${name}`-style substitution points.

Here's an example usage of a Chameleon text template. Create a file on disk named `mytemplate.txt` in your project's `templates` directory with the following contents:

```
Hello, ${name}!
```

Then in your project's `views.py` module, you can create a view which renders this template:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='templates/mytemplate.txt')
4 def my_view(request):
5     return {'name': 'world'}
```

When the template is rendered, it will show:

```
Hello, world!
```

If you'd rather use templates directly within a view callable (without the indirection of using a renderer), see `pyramid.chameleon_text` for the API description.

See also *Built-In Renderers* for more general information about renderers, including Chameleon text renderers.

11.6 Side Effects of Rendering a Chameleon Template

When a Chameleon template is rendered from a file, the templating engine writes a file in the same directory as the template file itself as a kind of cache, in order to do less work the next time the template needs to be read from disk. If you see “strange” `.py` files showing up in your `templates` directory (or otherwise directly “next” to your templates), it is due to this feature.

If you're using a version control system such as Subversion, you should configure it to ignore these files. Here's the contents of the author's `svn propedit svn:ignore .` in each of my `templates` directories.

```
*.pt.py
*.txt.py
```

Note that I always name my Chameleon ZPT template files with a `.pt` extension and my Chameleon text template files with a `.txt` extension so that these `svn:ignore` patterns work.

11.7 Nicier Exceptions in Chameleon Templates

The exceptions raised by Chameleon templates when a rendering fails are sometimes less than helpful. Pyramid allows you to configure your application development environment so that exceptions generated by Chameleon during template compilation and execution will contain nicer debugging information.



Template-debugging behavior is not recommended for production sites as it slows renderings; it's usually only desirable during development.

In order to turn on template exception debugging, you can use an environment variable setting or a configuration file setting.

To use an environment variable, start your application under a shell using the `PYRAMID_DEBUG_TEMPLATES` operating system environment variable set to 1, For example:

```
$ PYRAMID_DEBUG_TEMPLATES=1 bin/paster serve myproject.ini
```

To use a setting in the application `.ini` file for the same purpose, set the `debug_templates` key to `true` within the application's configuration section, e.g.:

```
1 [app:MyProject]
2 use = egg:MyProject#app
3 debug_templates = true
```

With template debugging off, a `NameError` exception resulting from rendering a template with an undefined variable (e.g. `${wrong}`) might end like this:

```
File "...", in __getitem__
    raise NameError(key)
NameError: wrong
```

11. TEMPLATES

Note that the exception has no information about which template was being rendered when the error occurred. But with template debugging on, an exception resulting from the same problem might end like so:

```
RuntimeError: Caught exception rendering template.
- Expression: ``wrong``
- Filename:   /home/fred/env/proj/proj/templates/mytemplate.pt
- Arguments: renderer_name: proj:templates/mytemplate.pt
              template: <PageTemplateFile - at 0x1d2ecf0>
              xincludes: <XIncludes - at 0x1d3a130>
              request: <Request - at 0x1d2ecd0>
              project: proj
              macros: <Macros - at 0x1d3aed0>
              context: <MyResource None at 0x1d39130>
              view: <function my_view at 0x1d23570>

NameError: wrong
```

The latter tells you which template the error occurred in, as well as displaying the arguments passed to the template itself.



Turning on `debug_templates` has the same effect as using the Chameleon environment variable `CHAMELEON_DEBUG`. See [Chameleon Environment Variables](#) for more information.

11.8 *Chameleon* Template Internationalization

See *Chameleon Template Support for Translation Strings* for information about supporting internationalized units of text within *Chameleon* templates.

11.9 Templating With Mako Templates

Mako is a templating system written by Mike Bayer. Pyramid has built-in bindings for the Mako templating system. The language definition documentation for Mako templates is available from the Mako website.

To use a Mako template, given a *Mako* ZPT template file named `foo.mak` in the `templates` subdirectory in your application package named `mypackage`, you can configure the template as a *renderer* like so:


```

1 from pyramid.view import view_config
2
3 @view_config(renderer='foo.mak')
4 def my_view(request):
5     return {'project': 'my project'}

```

For the above view callable to work, the following setting needs to be present in the application stanza of your configuration's ini file:

```
mako.directories = mypackage:templates
```

This lets the Mako templating system know that it should look for templates in the `templates` subdirectory of the `mypackage` Python package. See *Mako Template Render Settings* for more information about the `mako.directories` setting and other Mako-related settings that can be placed into the application's ini file.

11.9.1 A Sample Mako Template

Here's what a simple *Mako* template used under Pyramid might look like:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:tal="http://xml.zope.org/namespaces/tal">
5 <head>
6     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
7     <title>${project} Application</title>
8 </head>
9 <body>
10    <h1 class="title">Welcome to <code>${project}</code>, an
11    application generated by the <a
12    href="http://docs.pylonsproject.org/projects/pyramid/dev/"
13    >pyramid</a> web application framework.</h1>
14 </body>
15 </html>

```

This template doesn't use any advanced features of Mako, only the ``${squiggly}`` replacement syntax for names that are passed in as *renderer globals*. See the the Mako documentation to use more advanced features.

11.10 Automatically Reloading Templates

It's often convenient to see changes you make to a template file appear immediately without needing to restart the application process. Pyramid allows you to configure your application development environment so that a change to a template will be automatically detected, and the template will be reloaded on the next rendering.



Auto-template-reload behavior is not recommended for production sites as it slows rendering slightly; it's usually only desirable during development.

In order to turn on automatic reloading of templates, you can use an environment variable, or a configuration file setting.

To use an environment variable, start your application under a shell using the `PYRAMID_RELOAD_TEMPLATES` operating system environment variable set to 1. For example:

```
$ PYRAMID_RELOAD_TEMPLATES=1 bin/paster serve myproject.ini
```

To use a setting in the application `.ini` file for the same purpose, set the `reload_templates` key to `true` within the application's configuration section, e.g.:

```
1 [app:main]
2 use = egg:MyProject#app
3 reload_templates = true
```

11.11 Available Add-On Template System Bindings

Jinja2 template bindings are available for Pyramid in the `pyramid_jinja2` package. You can get the latest release of this package from the Python package index (pypi).

VIEW CONFIGURATION

View configuration controls how *view lookup* operates in your application. In earlier chapters, you have been exposed to a few simple view configuration declarations without much explanation. In this chapter we will explore the subject in detail.

12.1 View Lookup and Invocation

View lookup is the Pyramid subsystem responsible for finding and invoking a *view callable*. The view lookup subsystem is passed a *context* and a *request* object.

View configuration information stored within in the *application registry* is compared against the context and request by the view lookup subsystem in order to find the “best” view callable for the set of circumstances implied by the context and request.

View predicate attributes are an important part of view configuration that enables the *View lookup* subsystem to find and invoke the appropriate view. Predicate attributes can be thought of like “narrowers”. In general, the greater number of predicate attributes possessed by a view’s configuration, the more specific the circumstances need to be before the registered view callable will be invoked.

12.2 Mapping a Resource or URL Pattern to a View Callable

A developer makes a *view callable* available for use within a Pyramid application via *view configuration*. A view configuration associates a view callable with a set of statements that determine the set of circumstances which must be true for the view callable to be invoked.

A view configuration statement is made about information present in the *context* resource and the *request*.

View configuration is performed in one of these ways:

- by running a *scan* against application source code which has a `pyramid.view.view_config` decorator attached to a Python object as per *View Configuration Using the @view_config Decorator*.
- by using the `pyramid.config.Configurator.add_view()` method as per *View Registration Using add_view()*.
- By specifying a view within a *route configuration*. View configuration via a route configuration is performed by using the `pyramid.config.Configurator.add_route()` method, passing a `view` argument specifying a view callable.



A package named `pyramid_handlers` (available from PyPI) provides an analogue of *Pylons*-style “controllers”, which are a special kind of view class which provides more automation when your application uses *URL dispatch* solely.

12.2.1 View Configuration Parameters

All forms of view configuration accept the same general types of arguments.

Many arguments supplied during view configuration are *view predicate* arguments. View predicate arguments used during view configuration are used to narrow the set of circumstances in which *view lookup* will find a particular view callable.

In general, the fewer number of predicates which are supplied to a particular view configuration, the more likely it is that the associated view callable will be invoked. The greater the number supplied, the less likely. A view with five predicates will always be found and evaluated before a view with two, for example. All predicates must match for the associated view to be called.

This does not mean however, that Pyramid “stops looking” when it finds a view registration with predicates that don’t match. If one set of view predicates does not match, the “next most specific” view (if

any) is consulted for predicates, and so on, until a view is found, or no view can be matched up with the request. The first view with a set of predicates all of which match the request environment will be invoked.

If no view can be found with predicates which allow it to be matched up with the request, Pyramid will return an error to the user's browser, representing a "not found" (404) page. See *Changing the Not Found View* for more information about changing the default notfound view.

Some view configuration arguments are non-predicate arguments. These tend to modify the response of the view callable or prevent the view callable from being invoked due to an authorization policy. The presence of non-predicate arguments in a view configuration does not narrow the circumstances in which the view callable will be invoked.

Non-Predicate Arguments

permission The name of a *permission* that the user must possess in order to invoke the *view callable*. See *Configuring View Security* for more information about view security and permissions.

If `permission` is not supplied, no permission is registered for this view (it's accessible by any caller).

attr The view machinery defaults to using the `__call__` method of the *view callable* (or the function itself, if the view callable is a function) to obtain a response. The `attr` value allows you to vary the method attribute used to obtain the response. For example, if your view was a class, and the class has a method named `index` and you wanted to use this method instead of the class' `__call__` method to return the response, you'd say `attr="index"` in the view configuration for the view. This is most useful when the view definition is a class.

If `attr` is not supplied, `None` is used (implying the function itself if the view is a function, or the `__call__` callable attribute if the view is a class).

renderer Denotes the *renderer* implementation which will be used to construct a *response* from the associated view callable's return value. (see also *Renderers*).

This is either a single string term (e.g. `json`) or a string implying a path or *asset specification* (e.g. `templates/views.pt`) naming a *renderer* implementation. If the `renderer` value does not contain a dot (`.`), the specified string will be used to look up a *renderer* implementation, and that *renderer* implementation will be used to construct a response from the view return value. If the `renderer` value contains a dot (`.`), the specified term will be treated as a path, and the filename extension of the last element in the path will be used to look up the *renderer* implementation, which will be passed the full path.

12. VIEW CONFIGURATION

When the `renderer` is a path, although a path is usually just a simple relative pathname (e.g. `templates/foo.pt`, implying that a template named “foo.pt” is in the “templates” directory relative to the directory of the current *package*), a path can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternately be a *asset specification* in the form `some.dotted.package_name:relative/path`, making it possible to address template assets which live in a separate package.

The `renderer` attribute is optional. If it is not defined, the “null” renderer is assumed (no rendering is performed and the value is passed back to the upstream Pyramid machinery unmolested). Note that if the view callable itself returns a *response* (see *View Callable Responses*), the specified renderer implementation is never called.

wrapper The *view name* of a different *view configuration* which will receive the response body of this view as the `request.wrapped_body` attribute of its own *request*, and the *response* returned by this view as the `request.wrapped_response` attribute of its own *request*. Using a wrapper makes it possible to “chain” views together to form a composite response. The response of the outermost wrapper view will be returned to the user. The wrapper view will be found as any view is found: see *View Lookup and Invocation*. The “best” wrapper view will be found based on the lookup ordering: “under the hood” this wrapper view is looked up via `pyramid.view.render_view_to_response(context, request, 'wrapper_viewname')`. The context and request of a wrapper view is the same context and request of the inner view.

If `wrapper` is not supplied, no wrapper view is used.

decorator A *dotted Python name* to function (or the function itself) which will be used to decorate the registered *view callable*. The decorator function will be called with the view callable as a single argument. The view callable it is passed will accept `(context, request)`. The decorator must return a replacement view callable which also accepts `(context, request)`.

mapper A Python object or *dotted Python name* which refers to a *view mapper*, or `None`. By default it is `None`, which indicates that the view should use the default view mapper. This plug-point is useful for Pyramid extension developers, but it’s not very useful for ‘civilians’ who are just developing stock Pyramid applications. Pay no attention to the man behind the curtain.

Predicate Arguments

These arguments modify view lookup behavior. In general, the more predicate arguments that are supplied, the more specific, and narrower the usage of the configured view.

name The *view name* required to match this view callable. Read *Traversal* to understand the concept of a view name.

If `name` is not supplied, the empty string is used (implying the default view).

context An object representing a Python class that the *context* resource must be an instance of *or* the *interface* that the *context* resource must provide in order for this view to be found and called. This predicate is true when the *context* resource is an instance of the represented class or if the *context* resource provides the represented interface; it is otherwise false.

If `context` is not supplied, the value `None`, which matches any resource, is used.

route_name If `route_name` is supplied, the view callable will be invoked only when the named route has matched.

This value must match the `name` of a *route configuration* declaration (see *URL Dispatch*) that must match before this view will be called. Note that the `route` configuration referred to by `route_name` will usually have a `*traverse` token in the value of its `pattern`, representing a part of the path that will be used by *traversal* against the result of the route's *root factory*.

If `route_name` is not supplied, the view callable will have a chance of being invoked if no other route was matched. This is when the request/context pair found via *resource location* does not indicate it matched any configured route.

request_type This value should be an *interface* that the *request* must provide in order for this view to be found and called.

If `request_type` is not supplied, the value `None` is used, implying any request type.

This is an advanced feature, not often used by “civilians”.

request_method This value can either be one of the strings `GET`, `POST`, `PUT`, `DELETE`, or `HEAD` representing an `HTTP REQUEST_METHOD`. A view declaration with this argument ensures that the view will only be called when the request's `method` attribute (aka the `REQUEST_METHOD` of the WSGI environment) string matches the supplied value.

If `request_method` is not supplied, the view will be invoked regardless of the `REQUEST_METHOD` of the *WSGI* environment.

request_param This value can be any string. A view declaration with this argument ensures that the view will only be called when the *request* has a key in the `request.params` dictionary (an `HTTP GET` or `POST` variable) that has a name which matches the supplied value.

If the value supplied has a `=` sign in it, e.g. `request_params="foo=123"`, then the key (`foo`) must both exist in the `request.params` dictionary, *and* the value must match the right hand side of the expression (`123`) for the view to “match” the current request.

If `request_param` is not supplied, the view will be invoked without consideration of keys and values in the `request.params` dictionary.

12. VIEW CONFIGURATION

containment This value should be a reference to a Python class or *interface* that a parent object in the context resource's *lineage* must provide in order for this view to be found and called. The resources in your resource tree must be "location-aware" to use this feature.

If `containment` is not supplied, the interfaces and classes in the lineage are not considered when deciding whether or not to invoke the view callable.

See *Location-Aware Resources* for more information about location-awareness.

xhr This value should be either `True` or `False`. If this value is specified and is `True`, the *WSGI* environment must possess an `HTTP_X_REQUESTED_WITH` (aka X-Requested-With) header that has the value `XMLHttpRequest` for the associated view callable to be found and called. This is useful for detecting AJAX requests issued from jQuery, Prototype and other Javascript libraries.

If `xhr` is not specified, the `HTTP_X_REQUESTED_WITH` HTTP header is not taken into consideration when deciding whether or not to invoke the associated view callable.

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

If `accept` is not specified, the `HTTP_ACCEPT` HTTP header is not taken into consideration when deciding whether or not to invoke the associated view callable.

header This value represents an HTTP header name or a header name/value pair.

If `header` is specified, it must be a header name or a `headername:headervalue` pair.

If `header` is specified without a value (a bare header name only, e.g. `If-Modified-Since`), the view will only be invoked if the HTTP header exists with any value in the request.

If `header` is specified, and possesses a name/value pair (e.g. `User-Agent:Mozilla/.*`), the view will only be invoked if the HTTP header exists *and* the HTTP header matches the value requested. When the `headervalue` contains a `:` (colon), it will be considered a name/value pair (e.g. `User-Agent:Mozilla/.*` or `Host:localhost`). The value portion should be a regular expression.

Whether or not the value represents a header name or a header name/value pair, the case of the header name is not significant.

If `header` is not specified, the composition, presence or absence of HTTP headers is not taken into consideration when deciding whether or not to invoke the associated view callable.

path_info This value represents a regular expression pattern that will be tested against the `PATH_INFO` WSGI environment variable to decide whether or not to call the associated view callable. If the regex matches, this predicate will be `True`.

If `path_info` is not specified, the WSGI `PATH_INFO` is not taken into consideration when deciding whether or not to invoke the associated view callable.

custom_predicates If `custom_predicates` is specified, it must be a sequence of references to custom predicate callables. Use custom predicates when no set of predefined predicates do what you need. Custom predicates can be combined with predefined predicates as necessary. Each custom predicate callable should accept two arguments: `context` and `request` and should return either `True` or `False` after doing arbitrary evaluation of the context resource and/or the request. If all callables return `True`, the associated view callable will be considered viable for a given request.

If `custom_predicates` is not specified, no custom predicates are used.

12.2.2 View Configuration Using the `@view_config` Decorator

For better locality of reference, you may use the `pyramid.view.view_config` decorator to associate your view functions with URLs instead of using imperative configuration for the same purpose.



Using this feature tends to slow down application startup slightly, as more work is performed at application startup to scan for view declarations.

Usage of the `view_config` decorator is a form of *declarative configuration* in decorator form. `view_config` can be used to associate *view configuration* information – as done via the equivalent imperative code – with a function that acts as a Pyramid view callable. All arguments to the `pyramid.config.Configurator.add_view()` method (save for the `view` argument) are available in decorator form and mean precisely the same thing.

An example of the `view_config` decorator might reside in a Pyramid application module `views.py`:

```

1 from resources import MyResource
2 from pyramid.view import view_config
3 from pyramid.response import Response
4
5 @view_config(name='my_view', request_method='POST', context=MyResource,
6             permission='read')
7 def my_view(request):
8     return Response('OK')
```

Using this decorator as above replaces the need to add this imperative configuration stanza:

12. VIEW CONFIGURATION

```
1 config.add_view('.views.my_view', name='my_view', request_method='POST',
2                 context=MyResource, permission='read')
```

All arguments to `view_config` may be omitted. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config()
5 def my_view(request):
6     """ My view """
7     return Response()
```

Such a registration as the one directly above implies that the view name will be `my_view`, registered with a `context` argument that matches any resource type, using no permission, registered against requests with any request method, request type, request param, route name, or containment.

The mere existence of a `@view_config` decorator doesn't suffice to perform view configuration. All that the decorator does is “annotate” the function with your configuration declarations, it doesn't process them. To make Pyramid process your `view_config` declarations, you *must* do use the `scan` method of a `Configurator`:

```
1 # config is assumed to be an instance of the
2 # pyramid.config.Configurator class
3 config.scan()
```

Please see *Configuration Decorations and Code Scanning* for detailed information about what happens when code is scanned for configuration declarations resulting from use of decorators like `view_config`.

See `pyramid.config` for additional API arguments to the `scan()` method. For example, the method allows you to supply a `package` argument to better control exactly *which* code will be scanned.

@view_config Placement

A `view_config` decorator can be placed in various points in your application.

If your view callable is a function, it may be used as a function decorator:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(name='edit')
5 def edit(request):
6     return Response('edited!')
```

If your view callable is a class, the decorator can also be used as a class decorator in Python 2.6 and better (Python 2.5 and below do not support class decorators). All the arguments to the decorator are the same when applied against a class as when they are applied against a function. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config()
5 class MyView(object):
6     def __init__(self, request):
7         self.request = request
8
9     def __call__(self):
10        return Response('hello')
```

You can use the `view_config` decorator as a simple callable to manually decorate classes in Python 2.5 and below without the decorator syntactic sugar, if you wish:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 class MyView(object):
5     def __init__(self, request):
6         self.request = request
7
8     def __call__(self):
9         return Response('hello')
```

```
11 my_view = view_config()(MyView)
```

More than one `view_config` decorator can be stacked on top of any number of others. Each decorator creates a separate view registration. For example:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
```

12. VIEW CONFIGURATION

```
3
4 @view_config(name='edit')
5 @view_config(name='change')
6 def edit(request):
7     return Response('edited!')
```

This registers the same view under two different names.

The decorator can also be used against class methods:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 class MyView(object):
5     def __init__(self, request):
6         self.request = request
7
8     @view_config(name='hello')
9     def amethod(self):
10        return Response('hello')
```

When the decorator is used against a class method, a view is registered for the *class*, so the class constructor must accept an argument list in one of two forms: either it must accept a single argument *request* or it must accept two arguments, *context*, *request*.

The method which is decorated must return a *response*.

Using the decorator against a particular method of a class is equivalent to using the *attr* parameter in a decorator attached to the class itself. For example, the above registration implied by the decorator being used against the *amethod* method could be spelled equivalently as the below:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(attr='amethod', name='hello')
5 class MyView(object):
6     def __init__(self, request):
7         self.request = request
8
9     def amethod(self):
10        return Response('hello')
```

12.2.3 View Registration Using `add_view()`

The `pyramid.config.Configurator.add_view()` method within `pyramid.config` is used to configure a view imperatively. The arguments to this method are very similar to the arguments that you provide to the `@view_config` decorator. For example:

```
1 from pyramid.response import Response
2
3 def hello_world(request):
4     return Response('hello!')
5
6 # config is assumed to be an instance of the
7 # pyramid.config.Configurator class
8 config.add_view(hello_world, name='hello.html')
```

The first argument, `view`, is required. It must either be a Python object which is the view itself or a *dotted Python name* to such an object. All other arguments are optional. See `pyramid.config.Configurator.add_view()` for more information.

12.2.4 Using Resource Interfaces In View Configuration

Instead of registering your views with a `context` that names a Python resource *class*, you can optionally register a view callable with a `context` which is an *interface*. An interface can be attached arbitrarily to any resource object. View lookup treats context interfaces specially, and therefore the identity of a resource can be divorced from that of the class which implements it. As a result, associating a view with an interface can provide more flexibility for sharing a single view between two or more different implementations of a resource type. For example, if two resource objects of different Python class types share the same interface, you can use the same view configuration to specify both of them as a `context`.

In order to make use of interfaces in your application during view dispatch, you must create an interface and mark up your resource classes or instances with interface declarations that refer to this interface.

To attach an interface to a resource *class*, you define the interface and use the `zope.interface.implements()` function to associate the interface with the class.

```
1 from zope.interface import Interface
2 from zope.interface import implements
3
4 class IHello(Interface):
5     """ A marker interface """
6
7 class Hello(object):
8     implements(IHello)
```

12. VIEW CONFIGURATION

To attach an interface to a resource *instance*, you define the interface and use the `zope.interface.alsoProvides()` function to associate the interface with the instance. This function mutates the instance in such a way that the interface is attached to it.

```
1 from zope.interface import Interface
2 from zope.interface import alsoProvides
3
4 class IHello(Interface):
5     """ A marker interface """
6
7 class Hello(object):
8     pass
9
10 def make_hello():
11     hello = Hello()
12     alsoProvides(hello, IHello)
13     return hello
```

Regardless of how you associate an interface, with a resource instance, or a resource class, the resulting code to associate that interface with a view callable is the same. Assuming the above code that defines an `IHello` interface lives in the root of your application, and its module is named “resources.py”, the interface declaration below will associate the `mypackage.views.hello_world` view with resources that implement, or provide, this interface.

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('mypackage.views.hello_world', name='hello.html',
4                context='mypackage.resources.IHello')
```

Any time a resource that is determined to be the *context* provides this interface, and a view named `hello.html` is looked up against it as per the URL, the `mypackage.views.hello_world` view callable will be invoked.

Note, in cases where a view is registered against a resource class, and a view is also registered against an interface that the resource class implements, an ambiguity arises. Views registered for the resource class take precedence over any views registered for any interface the resource class implements. Thus, if one view configuration names a *context* of both the class type of a resource, and another view configuration names a *context* of interface implemented by the resource’s class, and both view configurations are otherwise identical, the view registered for the *context*’s class will “win”.

For more information about defining resources with interfaces for use within view configuration, see *Resources Which Implement Interfaces*.

12.2.5 Configuring View Security

If an *authorization policy* is active, any *permission* attached to a *view configuration* found during view lookup will be verified. This will ensure that the currently authenticated user possesses that permission against the *context* resource before the view function is actually called. Here's an example of specifying a permission in a view configuration using `add_view()`:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('myproject.views.add_entry', name='add.html',
4                context='myproject.resources.IBlog', permission='add')
```

When an *authorization policy* is enabled, this view will be protected with the `add` permission. The view will *not be called* if the user does not possess the `add` permission relative to the current *context*. Instead the *forbidden view* result will be returned to the client as per *Protecting Views with Permissions*.

12.2.6 NotFound Errors

It's useful to be able to debug `NotFound` error responses when they occur unexpectedly due to an application registry misconfiguration. To debug these errors, use the `PYRAMID_DEBUG_NOTFOUND` environment variable or the `debug_notfound` configuration file setting. Details of why a view was not found will be printed to `stderr`, and the browser representation of the error will include the same information. See *Environment Variables and .ini File Settings* for more information about how, and where to set these values.

12. VIEW CONFIGURATION

RESOURCES

A *resource* is an object that represents a “place” in a tree related to your application. Every Pyramid application has at least one resource object: the *root* resource. Even if you don’t define a root resource manually, a default one is created for you. The root resource is the root of a *resource tree*. A resource tree is a set of nested dictionary-like objects which you can use to represent your website’s structure.

In an application which uses *traversal* to map URLs to code, the resource tree structure is used heavily to map each URL to a *view callable*. When *traversal* is used, Pyramid will walk through the resource tree by traversing through its nested dictionary structure in order to find a *context* resource. Once a context resource is found, the context resource and data in the request will be used to find a *view callable*.

In an application which uses *URL dispatch*, the resource tree is only used indirectly, and is often “invisible” to the developer. In URL dispatch applications, the resource “tree” is often composed of only the root resource by itself. This root resource sometimes has security declarations attached to it, but is not required to have any. In general, the resource tree is much less important in applications that use URL dispatch than applications that use traversal.

In “Zope-like” Pyramid applications, resource objects also often store data persistently, and offer methods related to mutating that persistent data. In these kinds of applications, resources not only represent the site structure of your website, but they become the *domain model* of the application.

Also:

- The `context` and `containment` predicate arguments to `add_view()` (or a `view_config()` decorator) reference a resource class or resource *interface*.
- A *root factory* returns a resource.
- A resource is exposed to *view* code as the *context* of a view.
- Various helpful Pyramid API methods expect a resource as an argument (e.g. `resource_url()` and others).

13.1 Defining a Resource Tree

When *traversal* is used (as opposed to a purely *url dispatch* based application), Pyramid expects to be able to traverse a tree composed of resources (the *resource tree*). Traversal begins at a root resource, and descends into the tree recursively, trying each resource's `__getitem__` method to resolve a path segment to another resource object. Pyramid imposes the following policy on resource instances in the tree:

- A container resource (a resource which contains other resources) must supply a `__getitem__` method which is willing to resolve a unicode name to a sub-resource. If a sub-resource by a particular name does not exist in a container resource, `__getitem__` method of the container resource must raise a `KeyError`. If a sub-resource by that name *does* exist, the container's `__getitem__` should return the sub-resource.
- Leaf resources, which do not contain other resources, must not implement a `__getitem__`, or if they do, their `__getitem__` method must always raise a `KeyError`.

See *Traversal* for more information about how traversal works against resource instances.

Here's a sample resource tree, represented by a variable named `root`:

```
1 class Resource(dict):
2     pass
3
4 root = Resource({'a':Resource({'b':Resource({'c':Resource()})})})
```

The resource tree we've created above is represented by a dictionary-like root object which has a single child named `a`. `a` has a single child named `b`, and `b` has a single child named `c`, which has no children. It is therefore possible to access `c` like so:

```
1 root['a']['b']['c']
```

If you returned the above `root` object from a *root factory*, the path `/a/b/c` would find the `c` object in the resource tree as the result of *traversal*.

In this example, each of the resources in the tree is of the same class. This is not a requirement. Resource elements in the tree can be of any type. We used a single class to represent all resources in the tree for the sake of simplicity, but in a “real” app, the resources in the tree can be arbitrary.

Although the example tree above can service a traversal, the resource instances in the above example are not aware of *location*, so their utility in a “real” application is limited. To make best use of built-in Pyramid API facilities, your resources should be “location-aware”. The next section details how to make resources location-aware.

13.2 Location-Aware Resources

In order for certain Pyramid location, security, URL-generation, and traversal APIs to work properly against the resources in a resource tree, all resources in the tree must be *location*-aware. This means they must have two attributes: `__parent__` and `__name__`.

The `__parent__` attribute of a location-aware resource should be a reference to the resource’s parent resource instance in the tree. The `__name__` attribute should be the name with which a resource’s parent refers to the resource via `__getitem__`.

The `__parent__` of the root resource should be `None` and its `__name__` should be the empty string. For instance:

```
1 class MyRootResource(object):
2     __name__ = ''
3     __parent__ = None
```

A resource returned from the root resource’s `__getitem__` method should have a `__parent__` attribute that is a reference to the root resource, and its `__name__` attribute should match the name by which it is reachable via the root resource’s `__getitem__`. A container resource within the root resource should have a `__getitem__` that returns resources with a `__parent__` attribute that points at the container, and these subobjects should have a `__name__` attribute that matches the name by which they are retrieved from the container via `__getitem__`. This pattern continues recursively “up” the tree from the root.

The `__parent__` attributes of each resource form a linked list that points “downwards” toward the root. This is analogous to the `..` entry in filesystem directories. If you follow the `__parent__` values from any resource in the resource tree, you will eventually come to the root resource, just like if you keep executing the `cd ..` filesystem command, eventually you will reach the filesystem root directory.



If your root resource has a `__name__` argument that is not `None` or the empty string, URLs returned by the `resource_url()` function and paths generated by the `resource_path()` and `resource_path_tuple()` APIs will be generated improperly. The value of `__name__` will be prepended to every path and URL generated (as opposed to a single leading slash or empty tuple element).

Using `pyramid_traversalwrapper`

If you'd rather not manage the `__name__` and `__parent__` attributes of your resources “by hand”, an add-on package named `pyramid_traversalwrapper` can help.

In order to use this helper feature, you must first install the `pyramid_traversalwrapper` package (available via PyPI), then register its `ModelGraphTraverser` as the traversal policy, rather than the default `Pyramid` traverser. The package contains instructions for doing so.

Once `Pyramid` is configured with this feature, you will no longer need to manage the `__parent__` and `__name__` attributes on resource objects “by hand”. Instead, as necessary, during traversal `Pyramid` will wrap each resource (even the root resource) in a `LocationProxy` which will dynamically assign a `__name__` and a `__parent__` to the traversed resource (based on the last traversed resource and the name supplied to `__getitem__`). The root resource will have a `__name__` attribute of `None` and a `__parent__` attribute of `None`.

Applications which use tree-walking `Pyramid` APIs require location-aware resources. These APIs include (but are not limited to) `resource_url()`, `find_resource()`, `find_root()`, `find_interface()`, `resource_path()`, `resource_path_tuple()`, or `traverse()`, `virtual_root()`, and (usually) `has_permission()` and `principals_allowed_by_permission()`.

In general, since so much `Pyramid` infrastructure depends on location-aware resources, it's a good idea to make each resource in your tree location-aware.

13.3 Generating The URL Of A Resource

If your resources are *location* aware, you can use the `pyramid.url.resource_url()` API to generate a URL for the resource. This URL will use the resource's position in the parent tree to create a resource path, and it will prefix the path with the current application URL to form a fully-qualified URL with the scheme, host, port, and path. You can also pass extra arguments to `resource_url()` to influence the generated URL.

The simplest call to `resource_url()` looks like this:

```
1 from pyramid.url import resource_url
2 url = resource_url(resource, request)
```

The `request` passed to `resource_url` in the above example is an instance of a `Pyramid request` object.

If the resource referred to as `resource` in the above example was the root resource, and the host that was used to contact the server was `example.com`, the URL generated would be `http://example.com/`. However, if the resource was a child of the root resource named `a`, the generated URL would be `http://example.com/a/`.

A slash is appended to all resource URLs when `resource_url()` is used to generate them in this simple manner, because resources are “places” in the hierarchy, and URLs are meant to be clicked on to be visited. Relative URLs that you include on HTML pages rendered as the result of the default view of a resource are more apt to be relative to these resources than relative to their parent.

You can also pass extra elements to `resource_url()`:

```
1 from pyramid.url import resource_url
2 url = resource_url(resource, request, 'foo', 'bar')
```

If the resource referred to as `resource` in the above example was the root resource, and the host that was used to contact the server was `example.com`, the URL generated would be `http://example.com/foo/bar`. Any number of extra elements can be passed to `resource_url()` as extra positional arguments. When extra elements are passed, they are appended to the resource’s URL. A slash is not appended to the final segment when elements are passed.

You can also pass a query string:

```
1 from pyramid.url import resource_url
2 url = resource_url(resource, request, query={'a': '1'})
```

If the resource referred to as `resource` in the above example was the root resource, and the host that was used to contact the server was `example.com`, the URL generated would be `http://example.com/?a=1`.

When a *virtual root* is active, the URL generated by `resource_url()` for a resource may be “shorter” than its physical tree path. See *Virtual Root Support* for more information about virtually rooting a resource.

The shortcut method of the *request* named `pyramid.request.Request.resource_url()` can be used instead of `resource_url()` to generate a resource URL.

For more information about generating resource URLs, see the documentation for `pyramid.url.resource_url()`.

13.3.1 Overriding Resource URL Generation

If a resource object implements a `__resource_url__` method, this method will be called when `resource_url()` is called to generate a URL for the resource, overriding the default URL returned for the resource by `resource_url()`.

The `__resource_url__` hook is passed two arguments: `request` and `info`. `request` is the *request* object passed to `resource_url()`. `info` is a dictionary with two keys:

physical_path The “physical path” computed for the resource, as defined by `pyramid.traversal.resource_path(resource)`.

virtual_path The “virtual path” computed for the resource, as defined by *Virtual Root Support*. This will be identical to the physical path if virtual rooting is not enabled.

The `__resource_url__` method of a resource should return a string representing a URL. If it cannot override the default, it should return `None`. If it returns `None`, the default URL will be returned.

Here’s an example `__resource_url__` method.

```
1 class Resource(object):
2     def __resource_url__(self, request, info):
3         return request.application_url + info['virtual_path']
```

The above example actually just generates and returns the default URL, which would have been what was returned anyway, but your code can perform arbitrary logic as necessary. For example, your code may wish to override the hostname or port number of the generated URL.

Note that the URL generated by `__resource_url__` should be fully qualified, should end in a slash, and should not contain any query string or anchor elements (only path elements) to work best with `resource_url()`.

13.4 Generating the Path To a Resource

`pyramid.traversal.resource_path()` returns a string object representing the absolute physical path of the resource object based on its position in the resource tree. Each segment of the path is separated with a slash character.

```
1 from pyramid.traversal import resource_path
2 url = resource_path(resource)
```

If `resource` in the example above was accessible in the tree as `root['a']['b']`, the above example would generate the string `/a/b`.

Any positional arguments passed in to `resource_path()` will be appended as path segments to the end of the resource path.

```
1 from pyramid.traversal import resource_path
2 url = resource_path(resource, 'foo', 'bar')
```

If `resource` in the example above was accessible in the tree as `root['a']['b']`, the above example would generate the string `/a/b/foo/bar`.

The resource passed in must be *location*-aware.

The presence or absence of a *virtual root* has no impact on the behavior of `resource_path()`.

13.5 Finding a Resource by Path

If you have a string path to a resource, you can grab the resource from that place in the application's resource tree using `pyramid.traversal.find_resource()`.

You can resolve an absolute path by passing a string prefixed with a `/` as the `path` argument:

```
1 from pyramid.traversal import find_resource
2 url = find_resource(anyresource, '/path')
```

Or you can resolve a path relative to the resource you pass in by passing a string that isn't prefixed by `/`:

```
1 from pyramid.traversal import find_resource
2 url = find_resource(anyresource, 'path')
```

Often the paths you pass to `find_resource()` are generated by the `resource_path()` API. These APIs are “mirrors” of each other.

If the path cannot be resolved when calling `find_resource()` (if the respective resource in the tree does not exist), a `KeyError` will be raised.

See the `pyramid.traversal.find_resource()` documentation for more information about resolving a path to a resource.

13.6 Obtaining the Lineage of a Resource

`pyramid.location.lineage()` returns a generator representing the *lineage* of the *location* aware *resource* object.

The `lineage()` function returns the resource it is passed, then each parent of the resource, in order. For example, if the resource tree is composed like so:

```
1 class Thing(object): pass
2
3 thing1 = Thing()
4 thing2 = Thing()
5 thing2.__parent__ = thing1
```

Calling `lineage(thing2)` will return a generator. When we turn it into a list, we will get:

```
list(lineage(thing2))
[ <Thing object at thing2>, <Thing object at thing1> ]
```

The generator returned by `lineage()` first returns the resource it was passed unconditionally. Then, if the resource supplied a `__parent__` attribute, it returns the resource represented by `resource.__parent__`. If *that* resource has a `__parent__` attribute, return that resource's parent, and so on, until the resource being inspected either has no `__parent__` attribute or which has a `__parent__` attribute of `None`.

See the documentation for `pyramid.location.lineage()` for more information.

13.7 Determining if a Resource is In The Lineage of Another Resource

Use the `pyramid.location.inside()` function to determine if one resource is in the *lineage* of another resource.

For example, if the resource tree is:

```
1 class Thing(object): pass
2
3 a = Thing()
4 b = Thing()
5 b.__parent__ = a
```


Calling `inside(b, a)` will return `True`, because `b` has a lineage that includes `a`. However, calling `inside(a, b)` will return `False` because `a` does not have a lineage that includes `b`.

The argument list for `inside()` is `(resource1, resource2)`. `resource1` is ‘inside’ `resource2` if `resource2` is a *lineage* ancestor of `resource1`. It is a lineage ancestor if its parent (or one of its parent’s parents, etc.) is an ancestor.

See `pyramid.location.inside()` for more information.

13.8 Finding the Root Resource

Use the `pyramid.traversal.find_root()` API to find the *root* resource. The root resource is the root resource of the *resource tree*. The API accepts a single argument: `resource`. This is a resource that is *location* aware. It can be any resource in the tree for which you want to find the root.

For example, if the resource tree is:

```

1 class Thing(object): pass
2
3 a = Thing()
4 b = Thing()
5 b.__parent__ = a

```

Calling `find_root(b)` will return `a`.

The root resource is also available as `request.root` within *view callable* code.

The presence or absence of a *virtual root* has no impact on the behavior of `find_root()`. The root object returned is always the *physical* root object.

13.9 Resources Which Implement Interfaces

Resources can optionally be made to implement an *interface*. An interface is used to tag a resource object with a “type” that can later be referred to within *view configuration* and by `pyramid.traversal.find_interface()`.

Specifying an interface instead of a class as the `context` or `containment` predicate arguments within *view configuration* statements makes it possible to use a single view callable for more than one class of resource object. If your application is simple enough that you see no reason to want to do this, you can skip reading this section of the chapter.

For example, here’s some code which describes a blog entry which also declares that the blog entry implements an *interface*.

13. RESOURCES

```
1 import datetime
2 from zope.interface import implements
3 from zope.interface import Interface
4
5 class IBlogEntry(Interface):
6     pass
7
8 class BlogEntry(object):
9     implements(IBlogEntry)
10    def __init__(self, title, body, author):
11        self.title = title
12        self.body = body
13        self.author = author
14        self.created = datetime.datetime.now()
```

This resource consists of two things: the class which defines the resource constructor as the class `BlogEntry`, and an *interface* attached to the class via an `implements` statement at class scope using the `IBlogEntry` interface as its sole argument.

The interface object used must be an instance of a class that inherits from `zope.interface.Interface`.

A resource class may implement zero or more interfaces. You specify that a resource implements an interface by using the `zope.interface.implements()` function at class scope. The above `BlogEntry` resource implements the `IBlogEntry` interface.

You can also specify that a particular resource *instance* provides an interface, as opposed to its class. When you declare that a class implements an interface, all instances of that class will also provide that interface. However, you can also just say that a single object provides the interface. To do so, use the `zope.interface.directlyProvides()` function:

```
1 from zope.interface import directlyProvides
2 from zope.interface import Interface
3
4 class IBlogEntry(Interface):
5     pass
6
7 class BlogEntry(object):
8     def __init__(self, title, body, author):
9         self.title = title
10        self.body = body
11        self.author = author
12        self.created = datetime.datetime.now()
13
```

```

14 entry = BlogEntry('title', 'body', 'author')
15 directlyProvides(entry, IBlogEntry)

```

`zope.interface.directlyProvides()` will replace any existing interface that was previously provided by an instance. If a resource object already has instance-level interface declarations that you don't want to replace, use the `zope.interface.alsoProvides()` function:

```

1  from zope.interface import alsoProvides
2  from zope.interface import directlyProvides
3  from zope.interface import Interface
4
5  class IBlogEntry1(Interface):
6      pass
7
8  class IBlogEntry2(Interface):
9      pass
10
11 class BlogEntry(object):
12     def __init__(self, title, body, author):
13         self.title = title
14         self.body = body
15         self.author = author
16         self.created = datetime.datetime.now()
17
18 entry = BlogEntry('title', 'body', 'author')
19 directlyProvides(entry, IBlogEntry1)
20 alsoProvides(entry, IBlogEntry2)

```

`zope.interface.alsoProvides()` will augment the set of interfaces directly provided by an instance instead of overwriting them like `zope.interface.directlyProvides()` does.

For more information about how resource interfaces can be used by view configuration, see *Using Resource Interfaces In View Configuration*.

13.10 Finding a Resource With a Class or Interface in Lineage

Use the `find_interface()` API to locate a parent that is of a particular Python class, or which implements some *interface*.

For example, if your resource tree is composed as follows:

13. RESOURCES

```
1 class Thing1(object): pass
2 class Thing2(object): pass
3
4 a = Thing1()
5 b = Thing2()
6 b.__parent__ = a
```

Calling `find_interface(a, Thing1)` will return the `a` resource because `a` is of class `Thing1` (the resource passed as the first argument is considered first, and is returned if the class or interface spec matches).

Calling `find_interface(b, Thing1)` will return the `a` resource because `a` is of class `Thing1` and `a` is the first resource in `b`'s lineage of this class.

Calling `find_interface(b, Thing2)` will return the `b` resource.

The second argument to `find_interface` may also be a *interface* instead of a class. If it is an interface, each resource in the lineage is checked to see if the resource implements the specified interface (instead of seeing if the resource is of a class). See also *Resources Which Implement Interfaces*.

13.11 Pyramid API Functions That Act Against Resources

A resource object is used as the *context* provided to a view. See *Traversal* and *URL Dispatch* for more information about how a resource object becomes the context.

The APIs provided by `pyramid.traversal` are used against resource objects. These functions can be used to find the “path” of a resource, the root resource in a resource tree, or to generate a URL for a resource.

The APIs provided by `pyramid.location` are used against resources. These can be used to walk down a resource tree, or conveniently locate one resource “inside” another.

Some APIs in `pyramid.security` accept a resource object as a parameter. For example, the `has_permission()` API accepts a resource object as one of its arguments; the ACL is obtained from this resource or one of its ancestors. Other APIs in the `pyramid.security` module also accept *context* as an argument, and a context is always a resource.

STATIC ASSETS

An *asset* is any file contained within a Python *package* which is *not* a Python source code file. For example, each of the following is an asset:

- a GIF image file contained within a Python package or contained within any subdirectory of a Python package.
- a CSS file contained within a Python package or contained within any subdirectory of a Python package.
- a JavaScript source file contained within a Python package or contained within any subdirectory of a Python package.
- A directory within a package that does not have an `__init__.py` in it (if it possessed an `__init__.py` it would *be* a package).
- a *Chameleon* or *Mako* template file contained within a Python package.

The use of assets is quite common in most web development projects. For example, when you create a Pyramid application using one of the available “paster” templates, as described in *Creating the Project*, the directory representing the application contains a Python *package*. Within that Python package, there are directories full of files which are static assets. For example, there’s a `static` directory which contains `.css`, `.js`, and `.gif` files. These asset files are delivered when a user visits an application URL.

14.1 Understanding Asset Specifications

Let’s imagine you’ve created a Pyramid application that uses a *Chameleon* ZPT template via the `pyramid.renderers.render_to_response()` API. For example, the application might address the asset using the *asset specification* `myapp:templates/some_template.pt` using that API within a `views.py` file inside a `myapp` package:

14. STATIC ASSETS

```
1 from pyramid.renderers import render_to_response
2 render_to_response('myapp:templates/some_template.pt', {}, request)
```

“Under the hood”, when this API is called, Pyramid attempts to make sense out of the string `myapp:templates/some_template.pt` provided by the developer. This string is an *asset specification*. It is composed of two parts:

- The *package name* (`myapp`)
- The *asset name* (`templates/some_template.pt`), relative to the package directory.

The two parts are separated by the colon character.

Pyramid uses the Python *pkg_resources* API to resolve the package name and asset name to an absolute (operating-system-specific) file name. It eventually passes this resolved absolute filesystem path to the Chameleon templating engine, which then uses it to load, parse, and execute the template file.

There is a second form of asset specification: a *relative* asset specification. Instead of using an “absolute” asset specification which includes the package name, in certain circumstances you can omit the package name from the specification. For example, you might be able to use `templates/mytemplate.pt` instead of `myapp:templates/some_template.pt`. Such asset specifications are usually relative to a “current package.” The “current package” is usually the package which contains the code that *uses* the asset specification. Pyramid APIs which accept relative asset specifications typically describe what the asset is relative to in their individual documentation.

14.2 Serving Static Assets

Pyramid makes it possible to serve up static asset files from a directory on a filesystem to an application user’s browser. Use the `pyramid.config.Configurator.add_static_view()` to instruct Pyramid to serve static assets such as JavaScript and CSS files. This mechanism makes a directory of static files available at a name relative to the application root URL, e.g. `/static` or as an external URL.



`add_static_view()` cannot serve a single file, nor can it serve a directory of static files directly relative to the root URL of a Pyramid application. For these features, see *Advanced: Serving Static Assets Using a View Callable*.

Here’s an example of a use of `add_static_view()` that will serve files up from the `/var/www/static` directory of the computer which runs the Pyramid application as URLs beneath the `/static` URL prefix.

```
1 # config is an instance of pyramid.config.Configurator
2 config.add_static_view(name='static', path='/var/www/static')
```

The name represents a URL *prefix*. In order for files that live in the `path` directory to be served, a URL that requests one of them must begin with that prefix. In the example above, `name` is `static`, and `path` is `/var/www/static`. In English, this means that you wish to serve the files that live in `/var/www/static` as sub-URLs of the `/static` URL prefix. Therefore, the file `/var/www/static/foo.css` will be returned when the user visits your application's URL `/static/foo.css`.

A static directory named at `path` may contain subdirectories recursively, and any subdirectories may hold files; these will be resolved by the static view as you would expect. The `Content-Type` header returned by the static view for each particular type of file is dependent upon its file extension.

By default, all files made available via `add_static_view()` are accessible by completely anonymous users. Simple authorization can be required, however. To protect a set of static files using a permission, in addition to passing the required `name` and `path` arguments, also pass the `permission` keyword argument to `add_static_view()`. The value of the `permission` argument represents the *permission* that the user must have relative to the current *context* when the static view is invoked. A user will be required to possess this permission to view any of the files represented by `path` of the static view. If your static assets must be protected by a more complex authorization scheme, see *Advanced: Serving Static Assets Using a View Callable*.

Here's another example that uses an *asset specification* instead of an absolute path as the `path` argument. To convince `add_static_view()` to serve files up under the `/static` URL from the `a/b/c/static` directory of the Python package named `some_package`, we can use a fully qualified *asset specification* as the `path`:

```
1 # config is an instance of pyramid.config.Configurator
2 config.add_static_view(name='static', path='some_package:a/b/c/static')
```

The `path` provided to `add_static_view()` may be a fully qualified *asset specification* or an *absolute path*.

Instead of representing a URL prefix, the `name` argument of a call to `add_static_view()` can alternately be a *URL*. Each of examples we've seen so far have shown usage of the `name` argument as a URL prefix. However, when `name` is a *URL*, static assets can be served from an external webserver. In this mode, the `name` is used as the URL prefix when generating a URL using `pyramid.url.static_url()`.

For example, `add_static_view()` may be fed a `name` argument which is `http://example.com/images`:

14. STATIC ASSETS

```
1 # config is an instance of pyramid.config.Configurator
2 config.add_static_view(name='http://example.com/images',
3                       path='mypackage:images')
```

Because `add_static_view()` is provided with a `name` argument that is the URL `http://example.com/images`, subsequent calls to `static_url()` with paths that start with the `path` argument passed to `add_static_view()` will generate a URL something like `http://example.com/images/logo.png`. The external webserver listening on `example.com` must be itself configured to respond properly to such a request. The `static_url()` API is discussed in more detail later in this chapter.

14.2.1 Generating Static Asset URLs

When a `add_static_view()` method is used to register a static asset directory, a special helper API named `pyramid.url.static_url()` can be used to generate the appropriate URL for an asset that lives in one of the directories named by the static registration `path` attribute.

For example, let's assume you create a set of static declarations like so:

```
1 config.add_static_view(name='static1', path='mypackage:assets/1')
2 config.add_static_view(name='static2', path='mypackage:assets/2')
```

These declarations create URL-accessible directories which have URLs that begin with `/static1` and `/static2`, respectively. The assets in the `assets/1` directory of the `mypackage` package are consulted when a user visits a URL which begins with `/static1`, and the assets in the `assets/2` directory of the `mypackage` package are consulted when a user visits a URL which begins with `/static2`.

You needn't generate the URLs to static assets "by hand" in such a configuration. Instead, use the `static_url()` API to generate them for you. For example:

```
1 from pyramid.url import static_url
2 from pyramid.chameleon_zpt import render_template_to_response
3
4 def my_view(request):
5     css_url = static_url('mypackage:assets/1/foo.css', request)
6     js_url = static_url('mypackage:assets/2/foo.js', request)
7     return render_template_to_response('templates/my_template.pt',
8                                       css_url = css_url,
9                                       js_url = js_url)
```


If the request “application URL” of the running system is `http://example.com`, the `css_url` generated above would be: `http://example.com/static1/foo.css`. The `js_url` generated above would be `http://example.com/static2/foo.js`.

One benefit of using the `static_url()` function rather than constructing static URLs “by hand” is that if you need to change the name of a static URL declaration, the generated URLs will continue to resolve properly after the rename.

URLs may also be generated by `static_url()` to static assets that live *outside* the Pyramid application. This will happen when the `add_static_view()` API associated with the path fed to `static_url()` is a *URL* instead of a view name. For example, the name argument may be `http://example.com` while the the path given may be `mypackage:images`:

```
1 config.add_static_view(name='http://example.com/images',
2                       path='mypackage:images')
```

Under such a configuration, the URL generated by `static_url` for assets which begin with `mypackage:images` will be prefixed with `http://example.com/images`:

```
1 static_url('mypackage:images/logo.png', request)
2 # -> http://example.com/images/logo.png
```

Using `static_url()` in conjunction with a `add_static_view()` makes it possible to put static media on a separate webserver during production (if the name argument to `add_static_view()` is a URL), while keeping static media package-internal and served by the development webserver during development (if the name argument to `add_static_view()` is a URL prefix). To create such a circumstance, we suggest using the `pyramid.registry.Registry.settings` API in conjunction with a setting in the application `.ini` file named `media_location`. Then set the value of `media_location` to either a prefix or a URL depending on whether the application is being run in development or in production (use a different `.ini` file for production than you do for development). This is just a suggestion for a pattern; any setting name other than `media_location` could be used.

14.3 Advanced: Serving Static Assets Using a View Callable

For more flexibility, static assets can be served by a *view callable* which you register manually. For example, if you’re using *URL dispatch*, you may want static assets to only be available as a fallback if no previous route matches. Alternately, you might like to serve a particular static asset manually, because its download requires authentication.

Note that you cannot use the `static_url()` API to generate URLs against assets made accessible by registering a custom static view.

14.3.1 Root-Relative Custom Static View (URL Dispatch Only)

The `pyramid.view.static` helper class generates a Pyramid view callable. This view callable can serve static assets from a directory. An instance of this class is actually used by the `add_static_view()` configuration method, so its behavior is almost exactly the same once it's configured.



The following example *will not work* for applications that use *traversal*, it will only work if you use *URL dispatch* exclusively. The root-relative route we'll be registering will always be matched before traversal takes place, subverting any views registered via `add_view` (at least those without a `route_name`). A `static` static view cannot be made root-relative when you use *traversal*.

To serve files within a directory located on your filesystem at `/path/to/static/dir` as the result of a “catchall” route hanging from the root that exists at the end of your routing table, create an instance of the `static` class inside a `static.py` file in your application root as below.

```
1 from pyramid.view import static
2 static_view = static('/path/to/static/dir')
```



For better cross-system flexibility, use an *asset specification* as the argument to `static` instead of a physical absolute filesystem path, e.g. `mypackage:static` instead of `/path/to/mypackage/static`.

Subsequently, you may wire the files that are served by this view up to be accessible as `/<filename>` using a configuration method in your application's startup code.

```
1 # .. every other add_route declaration should come
2 # before this one, as it will, by default, catch all requests
3
4 config.add_route('catchall_static', '/*subpath', 'myapp.static.static_view')
```

The special name `*subpath` above is used by the `static` view callable to signify the path of the file relative to the directory you're serving.

14.3.2 Registering A View Callable to Serve a “Static” Asset

You can register a simple view callable to serve a single static asset. To do so, do things “by hand”. First define the view callable.

```

1 import os
2 from webob import Response
3
4 def favicon_view(request):
5     here = os.path.dirname(__file__)
6     icon = open(os.path.join(here, 'static', 'favicon.ico'))
7     return Response(content_type='image/x-icon', app_iter=icon)

```

The above bit of code within `favicon_view` computes “here”, which is a path relative to the Python file in which the function is defined. It then uses the Python `open` function to obtain a file handle to a file within “here” named `static`, and returns a response using the open the file handle as the response’s `app_iter`. It makes sure to set the right `content_type` too.

You might register such a view via configuration as a view callable that should be called as the result of a traversal:

```

1 config.add_view('myapp.views.favicon_view', name='favicon.ico')

```

Or you might register it to be the view callable for a particular route:

```

1 config.add_route('favicon', '/favicon.ico',
2                 view='myapp.views.favicon_view')

```

Because this is a simple view callable, it can be protected with a *permission* or can be configured to respond under different circumstances using *view predicate* arguments.


14.4 Overriding Assets

It can often be useful to override specific assets from “outside” a given Pyramid application. For example, you may wish to reuse an existing Pyramid application more or less unchanged. However, some specific template file owned by the application might have inappropriate HTML, or some static asset (such as a logo file or some CSS file) might not be appropriate. You *could* just fork the application entirely, but it’s often more convenient to just override the assets that are inappropriate and reuse the application “as is”. This is particularly true when you reuse some “core” application over and over again for some set of customers (such as a CMS application, or some bug tracking application), and you want to make arbitrary visual modifications to a particular application deployment without forking the underlying code.

To this end, Pyramid contains a feature that makes it possible to “override” one asset with one or more other assets. In support of this feature, a *Configurator* API exists named `pyramid.config.Configurator.override_asset()`. This API allows you to *override* the following kinds of assets defined in any Python package:

14. STATIC ASSETS

- Individual *Chameleon* templates.
- A directory containing multiple Chameleon templates.
- Individual static files served up by an instance of the `pyramid.view.static` helper class.
- A directory of static files served up by an instance of the `pyramid.view.static` helper class.
- Any other asset (or set of assets) addressed by code that uses the `setuptools pkg_resources` API.

 The *ZCML* directive named `asset` serves the same purpose as the `override_asset()` method.

14.4.1 The `override_asset` API

An individual call to `override_asset()` can override a single asset. For example:

```
1 config.override_asset(  
2     to_override='some.package:templates/mytemplate.pt',  
3     override_with='another.package:othertemplates/anothertemplate.pt')
```

The string value passed to both `to_override` and `override_with` sent to the `override_asset` API is called an *asset specification*. The colon separator in a specification separates the *package name* from the *asset name*. The colon and the following asset name are optional. If they are not specified, the override attempts to resolve every lookup into a package from the directory of another package. For example:

```
1 config.override_asset(to_override='some.package',  
2                       override_with='another.package')
```

Individual subdirectories within a package can also be overridden:

```
1 config.override_asset(to_override='some.package:templates/',  
2                       override_with='another.package:othertemplates/')
```

If you wish to override a directory with another directory, you *must* make sure to attach the slash to the end of both the `to_override` specification and the `override_with` specification. If you fail to attach a slash to the end of a specification that points to a directory, you will get unexpected results.

You cannot override a directory specification with a file specification, and vice versa: a startup error will occur if you try. You cannot override an asset with itself: a startup error will occur if you try.

Only individual *package* assets may be overridden. Overrides will not traverse through subpackages within an overridden package. This means that if you want to override assets for both `some.package:templates`, and `some.package.views:templates`, you will need to register two overrides.

The package name in a specification may start with a dot, meaning that the package is relative to the package in which the configuration construction file resides (or the `package` argument to the `Configurator` class construction). For example:

```
1 config.override_asset(to_override='.subpackage:templates/',  
2                       override_with='another.package:templates/')
```

Multiple calls to `override_asset` which name a shared `to_override` but a different `override_with` specification can be “stacked” to form a search path. The first asset that exists in the search path will be used; if no asset exists in the override path, the original asset is used.

Asset overrides can actually override assets other than templates and static files. Any software which uses the `pkg_resources.get_resource_filename()`, `pkg_resources.get_resource_stream()` or `pkg_resources.get_resource_string()` APIs will obtain an overridden file when an override is used.

REQUEST AND RESPONSE OBJECTS



This chapter is adapted from a portion of the *WebOb* documentation, originally written by Ian Bicking.

Pyramid uses the *WebOb* package to supply *request* and *response* object implementations. The *request* object that is passed to a Pyramid *view* is an instance of the `pyramid.request.Request` class, which is a subclass of `webob.Request`. The *response* returned from a Pyramid *view renderer* is an instance of the `webob.Response` class. Users can also return an instance of `webob.Response` directly from a *view* as necessary.

WebOb is a project separate from Pyramid with a separate set of authors and a fully separate set of documentation. Pyramid adds some functionality to the standard WebOb request, which is documented in the *pyramid.request* API documentation.

WebOb provides objects for HTTP requests and responses. Specifically it does this by wrapping the WSGI request environment and response status/headers/app_iter (body).

WebOb request and response objects provide many conveniences for parsing WSGI requests and forming WSGI responses. WebOb is a nice way to represent “raw” WSGI requests and responses; however, we won’t cover that use case in this document, as users of Pyramid don’t typically need to use the WSGI-related features of WebOb directly. The reference documentation shows many examples of creating requests and using response objects in this manner, however.

15.1 Request

The request object is a wrapper around the WSGI environ dictionary. This dictionary contains keys for each header, keys that describe the request (including the path and query string), a file-like object for the request body, and a variety of custom keys. You can always access the environ with `req.environ`.

Some of the most important/interesting attributes of a request object:

req.method: The request method, e.g., 'GET', 'POST'

req.GET: A *multidict* with all the variables in the query string.

req.POST: A *multidict* with all the variables in the request body. This only has variables if the request was a POST and it is a form submission.

req.params: A *multidict* with a combination of everything in `req.GET` and `req.POST`.

req.body: The contents of the body of the request. This contains the entire request body as a string. This is useful when the request is a POST that is *not* a form submission, or a request like a PUT. You can also get `req.body_file` for a file-like object.

req.cookies: A simple dictionary of all the cookies.

req.headers: A dictionary of all the headers. This dictionary is case-insensitive.

req.urlvars and req.urlargs: `req.urlvars` are the keyword parameters associated with the request URL. `req.urlargs` are the positional parameters. These are set by products like Routes and Selector.

Also, for standard HTTP request headers there are usually attributes, for instance: `req.accept_language`, `req.content_length`, `req.user_agent`, as an example. These properties expose the *parsed* form of each header, for whatever parsing makes sense. For instance, `req.if_modified_since` returns a datetime object (or None if the header is was not provided).



Full API documentation for the Pyramid request object is available in *pyramid.request*.

15.1.1 Special Attributes Added to the Request by Pyramid

In addition to the standard *WebOb* attributes, Pyramid adds special attributes to every request: `context`, `registry`, `root`, `subpath`, `traversed`, `view_name`, `virtual_root`, `virtual_root_path`, `session`, and `tmpl_context`, `matchdict`, and `matched_route`. These attributes are documented further within the `pyramid.request.Request` API documentation.

15.1.2 URLs

In addition to these attributes, there are several ways to get the URL of the request. I'll show various values for an example URL `http://localhost/app/blog?id=10`, where the application is mounted at `http://localhost/app`.

req.url: The full request URL, with query string, e.g., `http://localhost/app/blog?id=10`

req.host: The host information in the URL, e.g., `localhost`

req.host_url: The URL with the host, e.g., `http://localhost`

req.application_url: The URL of the application (just the `SCRIPT_NAME` portion of the path, not `PATH_INFO`). E.g., `http://localhost/app`

req.path_url: The URL of the application including the `PATH_INFO`. e.g., `http://localhost/app/blog`

req.path: The URL including `PATH_INFO` without the host or scheme. e.g., `/app/blog`

req.path_qs: The URL including `PATH_INFO` and the query string. e.g., `/app/blog?id=10`

req.query_string: The query string in the URL, e.g., `id=10`

req.relative_url(url, to_application=False): Gives a URL, relative to the current URL. If `to_application` is `True`, then resolves it relative to `req.application_url`.

15.1.3 Methods

There are several methods but only a few you'll use often:

Request.blank(base_url): Creates a new request with blank information, based at the given URL. This can be useful for subrequests and artificial requests. You can also use `req.copy()` to copy an existing request, or for subrequests `req.copy_get()` which copies the request but always turns it into a GET (which is safer to share for subrequests).

req.get_response(wsgi_application): This method calls the given WSGI application with this request, and returns a Response object. You can also use this for subrequests, or testing.

15.1.4 Unicode

Many of the properties in the request object will return unicode values if the request encoding/charset is provided. The client *can* indicate the charset with something like `Content-Type: application/x-www-form-urlencoded; charset=utf8`, but browsers seldom set this. You can set the charset with `req.charset = 'utf8'`, or during instantiation with `Request(environ, charset='utf8')`. If you subclass `Request` you can also set `charset` as a class-level attribute.

If it is set, then `req.POST`, `req.GET`, `req.params`, and `req.cookies` will contain unicode strings. Each has a corresponding `req.str_*` (e.g., `req.str_POST`) that is always a `str`, and never unicode.

15.1.5 More Details

More detail about the request object API is available in:

- The `pyramid.request.Request` API documentation.
- The WebOb documentation. All methods and attributes of a `webob.Request` documented within the WebOb documentation will work with request objects created by Pyramid.

15.2 Response

The Pyramid response object can be imported as `pyramid.response.Response`. This import location is merely a facade for its original location: `webob.Response`.

A response object has three fundamental parts:

response.status: The response code plus reason message, like `'200 OK'`. To set the code without a message, use `status_int`, i.e.: `response.status_int = 200`.

response.headerlist: A list of all the headers, like `[('Content-Type', 'text/html')]`. There's a case-insensitive *multidict* in `response.headers` that also allows you to access these same headers.

response.app_iter: An iterable (such as a list or generator) that will produce the content of the response. This is also accessible as `response.body` (a string), `response.unicode_body` (a unicode object, informed by `response.charset`), and `response.body_file` (a file-like object; writing to it appends to `app_iter`).

Everything else in the object derives from this underlying state. Here's the highlights:

response.content_type The content type *not* including the charset parameter. Typical use:
`response.content_type = 'text/html'`.

response.charset: The charset parameter of the content-type, it also informs encoding in `response.unicode_body`. `response.content_type_params` is a dictionary of all the parameters.

response.set_cookie(key, value, max_age=None, path='/', ...): Set a cookie. The keyword arguments control the various cookie parameters. The `max_age` argument is the length for the cookie to live in seconds (you may also use a `timedelta` object). The `Expires` key will also be set based on the value of `max_age`.

response.delete_cookie(key, path='/', domain=None): Delete a cookie from the client. This sets `max_age` to 0 and the cookie value to "".

response.cache_expires(seconds=0): This makes this response cacheable for the given number of seconds, or if `seconds` is 0 then the response is uncacheable (this also sets the `Expires` header).

response(environ, start_response): The response object is a WSGI application. As an application, it acts according to how you create it. It *can* do conditional responses if you pass `conditional_response=True` when instantiating (or set that attribute later). It can also do HEAD and Range requests.

15.2.1 Headers

Like the request, most HTTP response headers are available as properties. These are parsed, so you can do things like `response.last_modified = os.path.getmtime(filename)`.

The details are available in the extracted Response documentation.

15.2.2 Instantiating the Response

Of course most of the time you just want to *make* a response. Generally any attribute of the response can be passed in as a keyword argument to the class; e.g.:

15. REQUEST AND RESPONSE OBJECTS

```
1 from pyramid.response import Response
2 response = Response(body='hello world!', content_type='text/plain')
```

The status defaults to '200 OK'. The `content_type` does not default to anything, though if you subclass `pyramid.response.Response` and set `default_content_type` you can override this behavior.

15.2.3 Exception Responses

To facilitate error responses like 404 Not Found, the module `webob.exc` contains classes for each kind of error response. These include boring, but appropriate error bodies. The exceptions exposed by this module, when used under Pyramid, should be imported from the `pyramid.httpexceptions` “facade” module. This import location is merely a facade for the original location of these exceptions: `webob.exc`.

Each class is named `pyramid.httpexceptions.HTTP*`, where `*` is the reason for the error. For instance, `pyramid.httpexceptions.HTTPNotFound`. It subclasses `pyramid.Response`, so you can manipulate the instances in the same way. A typical example is:

```
1 from pyramid.httpexceptions import HTTPNotFound
2 from pyramid.httpexceptions import HTTPMovedPermanently
3
4 response = HTTPNotFound('There is no such resource')
5 # or:
6 response = HTTPMovedPermanently(location=new_url)
```

These are not exceptions unless you are using Python 2.5+, because they are new-style classes which are not allowed as exceptions until Python 2.5. To get an exception object use `response.exception`. You can use this like:

```
1 from pyramid.httpexceptions import HTTPException
2 from pyramid.httpexceptions import HTTPNotFound
3
4 def aview(request):
5     try:
6         # ... stuff ...
7         raise HTTPNotFound('No such resource').exception
8     except HTTPException, e:
9         return request.get_response(e)
```

The exceptions are still WSGI applications, but you cannot set attributes like `content_type`, `charset`, etc. on these exception objects.

15.2.4 More Details

More details about the response object API are available in the `pyramid.response` documentation. More details about exception responses are in the `pyramid.httpexceptions` API documentation. The `WebOb` documentation is also useful.

15.3 Multidict

Several parts of `WebOb` use a “multidict”; this is a dictionary where a key can have multiple values. The quintessential example is a query string like `?pref=red&pref=blue`; the `pref` variable has two values: `red` and `blue`.

In a multidict, when you do `request.GET['pref']` you'll get back only `'blue'` (the last value of `pref`). Sometimes returning a string, and sometimes returning a list, is the cause of frequent exceptions. If you want *all* the values back, use `request.GET.getall('pref')`. If you want to be sure there is *one and only one* value, use `request.GET.getone('pref')`, which will raise an exception if there is zero or more than one value for `pref`.

When you use operations like `request.GET.items()` you'll get back something like `[('pref', 'red'), ('pref', 'blue')]`. All the key/value pairs will show up. Similarly `request.GET.keys()` returns `['pref', 'pref']`. Multidict is a view on a list of tuples; all the keys are ordered, and all the values are ordered.

SESSIONS

A *session* is a namespace which is valid for some period of continual activity that can be used to represent a user's interaction with a web application.

This chapter describes how to configure sessions, what session implementations Pyramid provides out of the box, how to store and retrieve data from sessions, and two session-specific features: flash messages, and cross-site request forgery attack prevention.

16.1 Using The Default Session Factory

In order to use sessions, you must set up a *session factory* during your Pyramid configuration.

A very basic, insecure sample session factory implementation is provided in the Pyramid core. It uses a cookie to store session information. This implementation has the following limitation:

- The session information in the cookies used by this implementation is *not* encrypted, so it can be viewed by anyone with access to the cookie storage of the user's browser or anyone with access to the network along which the cookie travels.
- The maximum number of bytes that are storable in a serialized representation of the session is fewer than 4000. This is suitable only for very small data sets.

It is digitally signed, however, and thus its data cannot easily be tampered with.

You can configure this session factory in your Pyramid application by using the `session_factory` argument to the `Configurator` class:

```
1 from pyramid.session import UnencryptedCookieSessionFactoryConfig
2 my_session_factory = UnencryptedCookieSessionFactoryConfig('itsaseekreet')
3
4 from pyramid.config import Configurator
5 config = Configurator(session_factory = my_session_factory)
```



Note the very long, very explicit name for `UnencryptedCookieSessionFactoryConfig`. It's trying to tell you that this implementation is, by default, *unencrypted*. You should not use it when you keep sensitive information in the session object, as the information can be easily read by both users of your application and third parties who have access to your users' network traffic. Use a different session factory implementation (preferably one which keeps session data on the server) for anything but the most basic of applications where "session security doesn't matter".

16.2 Using a Session Object

Once a session factory has been configured for your application, you can access session objects provided by the session factory via the `session` attribute of any `request` object. For example:

```
1 from pyramid.response import Response
2
3 def myview(request):
4     session = request.session
5     if 'abc' in session:
6         session['fred'] = 'yes'
7     session['abc'] = '123'
8     if 'fred' in session:
9         return Response('Fred was in the session')
10    else:
11        return Response('Fred was not in the session')
```

You can use a session much like a Python dictionary. It supports all dictionary methods, along with some extra attributes, and methods.

Extra attributes:

created An integer timestamp indicating the time that this session was created.

new A boolean. If `new` is `True`, this session is new. Otherwise, it has been constituted from data that was already serialized.

Extra methods:

changed() Call this when you mutate a mutable value in the session namespace. See the gotchas below for details on when, and why you should call this.

invalidate() Call this when you want to invalidate the session (dump all data, and – perhaps – set a clearing cookie).

The formal definition of the methods and attributes supported by the session object are in the `pyramid.interfaces.ISession` documentation.

Some gotchas:

- Keys and values of session data must be *pickleable*. This means, typically, that they are instances of basic types of objects, such as strings, lists, dictionaries, tuples, integers, etc. If you place an object in a session data key or value that is not pickleable, an error will be raised when the session is serialized.
- If you place a mutable value (for example, a list or a dictionary) in a session object, and you subsequently mutate that value, you must call the `changed()` method of the session object. In this case, the session has no way to know that it was modified. However, when you modify a session object directly, such as setting a value (i.e., `__setitem__`), or removing a key (e.g., `del` or `pop`), the session will automatically know that it needs to re-serialize its data, thus calling `changed()` is unnecessary. There is no harm in calling `changed()` in either case, so when in doubt, call it after you've changed sessioning data.

16.3 Using Alternate Session Factories

At the time of this writing, exactly one alternate session factory implementation exists, named `pyramid_beaker`. This is a session factory that uses the Beaker library as a backend. Beaker has support for file-based sessions, database based sessions, and encrypted cookie-based sessions. See http://github.com/Pylons/pyramid_beaker for more information about `pyramid_beaker`.

16.4 Creating Your Own Session Factory

If none of the default or otherwise available sessioning implementations for Pyramid suit you, you may create your own session object by implementing a *session factory*. Your session factory should return a *session*. The interfaces for both types are available in `pyramid.interfaces.ISessionFactory` and `pyramid.interfaces.ISession`. You might use the cookie implementation in the `pyramid.session` module as inspiration.

16.5 Flash Messages

“Flash messages” are simply a queue of message strings stored in the *session*. To use flash messaging, you must enable a *session factory* as described in *Using The Default Session Factory* or *Using Alternate Session Factories*.

Flash messaging has two main uses: to display a status message only once to the user after performing an internal redirect, and to allow generic code to log messages for single-time display without having direct access to an HTML template. The user interface consists of a number of methods of the *session* object.

16.5.1 Using the `session.flash` Method

To add a message to a flash message queue, use a session object’s `flash()` method:

```
request.session.flash('mymessage')
```

The `flash()` method appends a message to a flash queue, creating the queue if necessary.

`flash()` accepts three arguments:

flash (*message*, *queue*='', *allow_duplicate*=True)

The `message` argument is required. It represents a message you wish to later display to a user. It is usually a string but the `message` you provide is not modified in any way.

The `queue` argument allows you to choose a queue to which to append the message you provide. This can be used to push different kinds of messages into flash storage for later display in different places on a page. You can pass any name for your queue, but it must be a string. Each queue is independent, and can be popped by `pop_flash()` or examined via `peek_flash()` separately. `queue` defaults to the empty string. The empty string represents the default flash message queue.

```
request.session.flash(msg, 'myappsqueue')
```

The `allow_duplicate` argument defaults to `True`. If this is `False`, and you attempt to add a message value which is already present in the queue, it will not be added.

16.5.2 Using the `session.pop_flash` Method

Once one or more messages have been added to a flash queue by the `session.flash()` API, the `session.pop_flash()` API can be used to pop an entire queue and return it for use.

To pop a particular queue of messages from the flash object, use the session object's `pop_flash()` method. This returns a list of the messages that were added to the flash queue, and empties the queue.

`pop_flash` (*queue*='')

```
1 >>> request.session.flash('info message')
2 >>> request.session.pop_flash()
3 ['info message']
```

Calling `session.pop_flash()` again like above without a corresponding call to `session.flash()` will return an empty list, because the queue has already been popped.

```
1 >>> request.session.flash('info message')
2 >>> request.session.pop_flash()
3 ['info message']
4 >>> request.session.pop_flash()
5 []
```

16.5.3 Using the `session.peek_flash` Method

Once one or more messages has been added to a flash queue by the `session.flash()` API, the `session.peek_flash()` API can be used to “peek” at that queue. Unlike `session.pop_flash()`, the queue is not popped from flash storage.

`peek_flash` (*queue*='')

```
1 >>> request.session.flash('info message')
2 >>> request.session.peek_flash()
3 ['info message']
4 >>> request.session.peek_flash()
5 ['info message']
6 >>> request.session.pop_flash()
7 ['info message']
8 >>> request.session.peek_flash()
9 []
```

16.6 Preventing Cross-Site Request Forgery Attacks

Cross-site request forgery attacks are a phenomenon whereby a user with an identity on your website might click on a URL or button on another website which unwittingly redirects the user to your application to perform some command that requires elevated privileges.

You can avoid most of these attacks by making sure that the correct *CSRF token* has been set in an Pyramid session object before performing any actions in code which requires elevated privileges that is invoked via a form post. To use CSRF token support, you must enable a *session factory* as described in *Using The Default Session Factory* or *Using Alternate Session Factories*.

16.6.1 Using the `session.get_csrf_token` Method

To get the current CSRF token from the session, use the `session.get_csrf_token()` method.

```
token = request.session.get_csrf_token()
```

The `session.get_csrf_token()` method accepts no arguments. It returns a CSRF *token* string. If `session.get_csrf_token()` or `session.new_csrf_token()` was invoked previously for this session, the existing token will be returned. If no CSRF token previously existed for this session, a new token will be set into the session and returned. The newly created token will be opaque and randomized.

You can use the returned token as the value of a hidden field in a form that posts to a method that requires elevated privileges. The handler for the form post should use `session.get_csrf_token()` *again* to obtain the current CSRF token related to the user from the session, and compare it to the value of the hidden form field. For example, if your form rendering included the CSRF token obtained via `session.get_csrf_token()` as a hidden input field named `csrf_token`:

```
1 token = request.session.get_csrf_token()
2 if token != request.POST['csrf_token']:
3     raise ValueError('CSRF token did not match')
```

16.6.2 Using the `session.new_csrf_token` Method

To explicitly add a new CSRF token to the session, use the `session.new_csrf_token()` method. This differs only from `session.get_csrf_token()` inasmuch as it clears any existing CSRF token, creates a new CSRF token, sets the token into the session, and returns the token.

```
token = request.session.new_csrf_token()
```


SECURITY

Pyramid provides an optional declarative authorization system that can prevent a *view* from being invoked based on an *authorization policy*. Before a view is invoked, the authorization system can use the credentials in the *request* along with the *context* resource to determine if access will be allowed. Here's how it works at a high level:

- A *request* is generated when a user visits the application.
- Based on the request, a *context* resource is located through *resource location*. A context is located differently depending on whether the application uses *traversal* or *URL dispatch*, but a context is ultimately found in either case. See the *URL Dispatch* chapter for more information.
- A *view callable* is located by *view lookup* using the context as well as other attributes of the request.
- If an *authentication policy* is in effect, it is passed the request; it returns some number of *principal* identifiers.
- If an *authorization policy* is in effect and the *view configuration* associated with the view callable that was found has a *permission* associated with it, the authorization policy is passed the *context*, some number of *principal* identifiers returned by the authentication policy, and the *permission* associated with the view; it will allow or deny access.
- If the authorization policy allows access, the view callable is invoked.
- If the authorization policy denies access, the view callable is not invoked; instead the *forbidden view* is invoked.

Security in Pyramid, unlike many systems, cleanly and explicitly separates authentication and authorization. Authentication is merely the mechanism by which credentials provided in the *request* are resolved to one or more *principal* identifiers. These identifiers represent the users and groups in effect during the request. Authorization then determines access based on the *principal* identifiers, the *view callable* being invoked, and the *context* resource.

Authorization is enabled by modifying your application to include an *authentication policy* and *authorization policy*. Pyramid comes with a variety of implementations of these policies. To provide maximal flexibility, Pyramid also allows you to create custom authentication policies and authorization policies.

17.1 Enabling an Authorization Policy

By default, Pyramid enables no authorization policy. All views are accessible by completely anonymous users. In order to begin protecting views from execution based on security settings, you need to enable an authorization policy.

17.1.1 Enabling an Authorization Policy Imperatively

Passing an `authorization_policy` argument to the constructor of the `Configurator` class enables an authorization policy.

You must also enable an *authentication policy* in order to enable the authorization policy. This is because authorization, in general, depends upon authentication. Use the `authentication_policy` argument to the `Configurator` class during application setup to specify an authentication policy.

For example:

```
1 from pyramid.config import Configurator
2 from pyramid.authentication import AuthTktAuthenticationPolicy
3 from pyramid.authorization import ACLAuthorizationPolicy
4 authentication_policy = AuthTktAuthenticationPolicy('seekrit')
5 authorization_policy = ACLAuthorizationPolicy()
6 config = Configurator(authentication_policy=authentication_policy,
7                       authorization_policy=authorization_policy)
```



the `authentication_policy` and `authorization_policy` arguments may also be passed to the `Configurator` as *dotted Python name* values, each representing the dotted name path to a suitable implementation global defined at Python module scope.

The above configuration enables a policy which compares the value of an “auth ticket” cookie passed in the request’s environment which contains a reference to a single *principal* against the principals present in any *ACL* found in the resource tree when attempting to call some *view*.

While it is possible to mix and match different authentication and authorization policies, it is an error to pass an authentication policy without the authorization policy or vice versa to a *Configurator* constructor.

See also the `pyramid.authorization` and `pyramid.authentication` modules for alternate implementations of authorization and authentication policies.

17.2 Protecting Views with Permissions

To protect a *view callable* from invocation based on a user's security settings when a particular type of resource becomes the *context*, you must pass a *permission* to *view configuration*. Permissions are usually just strings, and they have no required composition: you can name permissions whatever you like.

For example, the following view declaration protects the view named `add_entry.html` when the context resource is of type `Blog` with the `add` permission using the `pyramid.config.Configurator.add_view()` API:

```

1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('mypackage.views.blog_entry_add_view',
4                 name='add_entry.html',
5                 context='mypackage.resources.Blog',
6                 permission='add')
```

The equivalent view registration including the `add` permission name may be performed via the `@view_config` decorator:

```

1 from pyramid.view import view_config
2 from resources import Blog
3
4 @view_config(context=Blog, name='add_entry.html', permission='add')
5 def blog_entry_add_view(request):
6     """ Add blog entry code goes here """
7     pass
```

As a result of any of these various view configuration statements, if an authorization policy is in place when the view callable is found during normal application operations, the requesting user will need to possess the `add` permission against the *context* resource in order to be able to invoke the `blog_entry_add_view` view. If he does not, the *Forbidden view* will be invoked.

17.2.1 Setting a Default Permission

If a permission is not supplied to a view configuration, the registered view will always be executable by entirely anonymous users: any authorization policy in effect is ignored.

In support of making it easier to configure applications which are “secure by default”, Pyramid allows you to configure a *default* permission. If supplied, the default permission is used as the permission string to all view registrations which don't otherwise name a `permission` argument.

These APIs are in support of configuring a default permission for an application:

17. SECURITY

- The `default_permission` constructor argument to the `Configurator` constructor.
- The `pyramid.config.Configurator.set_default_permission()` method.

When a default permission is registered:

- if a view configuration names an explicit `permission`, the default permission is ignored for that view registration, and the view-configuration-named permission is used.
- if a view configuration names an explicit permission as the string `__no_permission_required__`, the default permission is ignored, and the view is registered *without* a permission (making it available to all callers regardless of their credentials).



When you register a default permission, *all* views (even *exception view* views) are protected by a permission. For all views which are truly meant to be anonymously accessible, you will need to associate the view's configuration with the `__no_permission_required__` permission.

17.3 Assigning ACLs to your Resource Objects

When the default Pyramid *authorization policy* determines whether a user possesses a particular permission with respect to a resource, it examines the *ACL* associated with the resource. An ACL is associated with a resource by adding an `__acl__` attribute to the resource object. This attribute can be defined on the resource *instance* if you need instance-level security, or it can be defined on the resource *class* if you just need type-level security.

For example, an ACL might be attached to the resource for a blog via its class:

```
1 from pyramid.security import Everyone
2 from pyramid.security import Allow
3
4 class Blog(object):
5     __acl__ = [
6         (Allow, Everyone, 'view'),
7         (Allow, 'group:editors', 'add'),
8         (Allow, 'group:editors', 'edit'),
9     ]
```

Or, if your resources are persistent, an ACL might be specified via the `__acl__` attribute of an *instance* of a resource:

```

1 from pyramid.security import Everyone
2 from pyramid.security import Allow
3
4 class Blog(object):
5     pass
6
7 blog = Blog()
8
9 blog.__acl__ = [
10     (Allow, Everyone, 'view'),
11     (Allow, 'group:editors', 'add'),
12     (Allow, 'group:editors', 'edit'),
13 ]

```

Whether an ACL is attached to a resource’s class or an instance of the resource itself, the effect is the same. It is useful to decorate individual resource instances with an ACL (as opposed to just decorating their class) in applications such as “CMS” systems where fine-grained access is required on an object-by-object basis.

17.4 Elements of an ACL

Here’s an example ACL:

```

1 from pyramid.security import Everyone
2 from pyramid.security import Allow
3
4 __acl__ = [
5     (Allow, Everyone, 'view'),
6     (Allow, 'group:editors', 'add'),
7     (Allow, 'group:editors', 'edit'),
8 ]

```

The example ACL indicates that the `pyramid.security.Everyone` principal – a special system-defined principal indicating, literally, everyone – is allowed to view the blog, the `group:editors` principal is allowed to add to and edit the blog.

Each element of an ACL is an *ACE* or access control entry. For example, in the above code block, there are three ACEs: `(Allow, Everyone, 'view')`, `(Allow, 'group:editors', 'add')`, and `(Allow, 'group:editors', 'edit')`.

17. SECURITY

The first element of any ACE is either `pyramid.security.Allow`, or `pyramid.security.Deny`, representing the action to take when the ACE matches. The second element is a *principal*. The third argument is a permission or sequence of permission names.

A principal is usually a user id, however it also may be a group id if your authentication system provides group information and the effective *authentication policy* policy is written to respect group information. For example, the `pyramid.authentication.RepozeWho1AuthenticationPolicy` respects group information if you configure it with a `callback`.

Each ACE in an ACL is processed by an authorization policy *in the order dictated by the ACL*. So if you have an ACL like this:

```
1 from pyramid.security import Everyone
2 from pyramid.security import Allow
3 from pyramid.security import Deny
4
5 __acl__ = [
6     (Allow, Everyone, 'view'),
7     (Deny, Everyone, 'view'),
8 ]
```

The default authorization policy will *allow* everyone the view permission, even though later in the ACL you have an ACE that denies everyone the view permission. On the other hand, if you have an ACL like this:

```
1 from pyramid.security import Everyone
2 from pyramid.security import Allow
3 from pyramid.security import Deny
4
5 __acl__ = [
6     (Deny, Everyone, 'view'),
7     (Allow, Everyone, 'view'),
8 ]
```

The authorization policy will deny everyone the view permission, even though later in the ACL is an ACE that allows everyone.

The third argument in an ACE can also be a sequence of permission names instead of a single permission name. So instead of creating multiple ACEs representing a number of different permission grants to a single `group:editors` group, we can collapse this into a single ACE, as below.

```
1 from pyramid.security import Everyone
2 from pyramid.security import Allow
3
4 __acl__ = [
5     (Allow, Everyone, 'view'),
6     (Allow, 'group:editors', ('add', 'edit')),
7     ]
```

17.5 Special Principal Names

Special principal names exist in the `pyramid.security` module. They can be imported for use in your own code to populate ACLs, e.g. `pyramid.security.Everyone`.

`pyramid.security.Everyone`

Literally, everyone, no matter what. This object is actually a string “under the hood” (`system.Everyone`). Every user “is” the principal named `Everyone` during every request, even if a security policy is not in use.

`pyramid.security.Authenticated`

Any user with credentials as determined by the current security policy. You might think of it as any user that is “logged in”. This object is actually a string “under the hood” (`system.Authenticated`).

17.6 Special Permissions

Special permission names exist in the `pyramid.security` module. These can be imported for use in ACLs. `pyramid.security.ALL_PERMISSIONS`

An object representing, literally, *all* permissions. Useful in an ACL like so: `(Allow, 'fred', ALL_PERMISSIONS)`. The `ALL_PERMISSIONS` object is actually a stand-in object that has a `__contains__` method that always returns `True`, which, for all known authorization policies, has the effect of indicating that a given principal “has” any permission asked for by the system.

17.7 Special ACEs

A convenience *ACE* is defined representing a deny to everyone of all permissions in `pyramid.security.DENY_ALL`. This ACE is often used as the *last* ACE of an ACL to explicitly cause inheriting authorization policies to “stop looking up the traversal tree” (effectively breaking any inheritance). For example, an ACL which allows *only fred* the view permission for a particular resource despite what inherited ACLs may say when the default authorization policy is in effect might look like so:

```
1 from pyramid.security import Allow
2 from pyramid.security import DENY_ALL
3
4 __acl__ = [ (Allow, 'fred', 'view'), DENY_ALL ]
```

“Under the hood”, the `pyramid.security.DENY_ALL` ACE equals the following:

```
1 from pyramid.security import ALL_PERMISSIONS
2 __acl__ = [ (Deny, Everyone, ALL_PERMISSIONS) ]
```

17.8 ACL Inheritance and Location-Awareness

While the default *authorization policy* is in place, if a resource object does not have an ACL when it is the context, its *parent* is consulted for an ACL. If that object does not have an ACL, *its* parent is consulted for an ACL, ad infinitum, until we’ve reached the root and there are no more parents left.

In order to allow the security machinery to perform ACL inheritance, resource objects must provide *location-awareness*. Providing *location-awareness* means two things: the root object in the resource tree must have a `__name__` attribute and a `__parent__` attribute.

```
1 class Blog(object):
2     __name__ = ''
3     __parent__ = None
```

An object with a `__parent__` attribute and a `__name__` attribute is said to be *location-aware*. Location-aware objects define an `__parent__` attribute which points at their parent object. The root object’s `__parent__` is `None`.

See `pyramid.location` for documentations of functions which use location-awareness. See also *Location-Aware Resources*.

17.9 Changing the Forbidden View

When Pyramid denies a view invocation due to an authorization denial, the special `forbidden` view is invoked. “Out of the box”, this forbidden view is very plain. See *Changing the Forbidden View* within *Using Hooks* for instructions on how to create a custom forbidden view and arrange for it to be called when view authorization is denied.

17.10 Debugging View Authorization Failures

If your application in your judgment is allowing or denying view access inappropriately, start your application under a shell using the `PYRAMID_DEBUG_AUTHORIZATION` environment variable set to 1. For example:

```
$ PYRAMID_DEBUG_AUTHORIZATION=1 bin/paster serve myproject.ini
```

When any authorization takes place during a top-level view rendering, a message will be logged to the console (to `stderr`) about what ACE in which ACL permitted or denied the authorization based on authentication information.

This behavior can also be turned on in the application `.ini` file by setting the `debug_authorization` key to `true` within the application’s configuration section, e.g.:

```
1 [app:main]
2 use = egg:MyProject#app
3 debug_authorization = true
```

With this debug flag turned on, the response sent to the browser will also contain security debugging information in its body.

17.11 Debugging Imperative Authorization Failures

The `pyramid.security.has_permission()` API is used to check security within view functions imperatively. It returns instances of objects that are effectively booleans. But these objects are not raw `True` or `False` objects, and have information attached to them about why the permission was allowed or denied. The object will be one of `pyramid.security.ACLAllowed`, `pyramid.security.ACLDenied`, `pyramid.security.Allowed`, or `pyramid.security.Denied`, as documented in *pyramid.security*. At the very minimum these objects will have a `msg` attribute, which is a string indicating why the permission was denied or allowed. Introspecting this information in the debugger or via print statements when a call to `has_permission()` fails is often useful.

17.12 Creating Your Own Authentication Policy

Pyramid ships with a number of useful out-of-the-box security policies (see `pyramid.authentication`). However, creating your own authentication policy is often necessary when you want to control the “horizontal and vertical” of how your users authenticate. Doing so is a matter of creating an instance of something that implements the following interface:

```
1 class AuthenticationPolicy(object):
2     """ An object representing a Pyramid authentication policy. """
3     def authenticated_userid(self, request):
4         """ Return the authenticated userid or ``None`` if no
5             authenticated userid can be found. """
6
7     def effective_principals(self, request):
8
9         """ Return a sequence representing the effective principals
10            including the userid and any groups belonged to by the current
11            user, including 'system' groups such as
12            ``pyramid.security.Everyone`` and
13            ``pyramid.security.Authenticated``. """
14
15     def remember(self, request, principal, **kw):
16         """ Return a set of headers suitable for 'remembering' the
17            principal named 'principal' when set in a response. An
18            individual authentication policy and its consumers can decide
19            on the composition and meaning of **kw. """
20
21     def forget(self, request):
22         """ Return a set of headers suitable for 'forgetting' the
23            current user on subsequent requests. """
```

After you do so, you can pass an instance of such a class into the `Configurator` class at configuration time as `authentication_policy` to use it.

17.13 Creating Your Own Authorization Policy

An authorization policy is a policy that allows or denies access after a user has been authenticated. By default, Pyramid will use the `pyramid.authorization.ACLAuthorizationPolicy` if an authentication policy is activated and an authorization policy isn't otherwise specified.

In some cases, it's useful to be able to use a different authorization policy than the default `ACLAuthorizationPolicy`. For example, it might be desirable to construct an alternate authorization policy which allows the application to use an authorization mechanism that does not involve *ACL* objects.

Pyramid ships with only a single default authorization policy, so you'll need to create your own if you'd like to use a different one. Creating and using your own authorization policy is a matter of creating an instance of an object that implements the following interface:

```
1 class IAuthorizationPolicy(object):
2     """ An object representing a Pyramid authorization policy. """
3     def permits(self, context, principals, permission):
4         """ Return 'True' if any of the 'principals' is allowed the
5             'permission' in the current 'context', else return 'False'
6         """
7
8     def principals_allowed_by_permission(self, context, permission):
9         """ Return a set of principal identifiers allowed by the
10            'permission' in 'context'. This behavior is optional; if you
11            choose to not implement it you should define this method as
12            something which raises a 'NotImplementedError'. This method
13            will only be called when the
14            'pyramid.security.principals_allowed_by_permission' API is
15            used. """
```

After you do so, you can pass an instance of such a class into the `Configurator` class at configuration time as `authorization_policy` to use it.

COMBINING TRAVERSAL AND URL DISPATCH

When you write most Pyramid applications, you'll be using one or the other of two available *resource location* subsystems: traversal or URL dispatch. However, to solve a limited set of problems, it's useful to use *both* traversal and URL dispatch together within the same application. Pyramid makes this possible via *hybrid* applications.



Reasoning about the behavior of a “hybrid” URL dispatch + traversal application can be challenging. To successfully reason about using URL dispatch and traversal together, you need to understand URL pattern matching, root factories, and the *traversal* algorithm, and the potential interactions between them. Therefore, we don't recommend creating an application that relies on hybrid behavior unless you must.

18.1 A Review of Non-Hybrid Applications

When used according to the tutorials in its documentation Pyramid is a “dual-mode” framework: the tutorials explain how to create an application in terms of using either *url dispatch* or *traversal*. This chapter details how you might combine these two dispatch mechanisms, but we'll review how they work in isolation before trying to combine them.

18.1.1 URL Dispatch Only

An application that uses *url dispatch* exclusively to map URLs to code will often have statements like this within your application startup configuration:

18. COMBINING TRAVERSAL AND URL DISPATCH

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_route('foobar', '{foo}/{bar}', view='myproject.views.foobar')
4 config.add_route('bazbuz', '{baz}/{buz}', view='myproject.views.bazbuz')
```

Each *route* typically corresponds to a single view callable, and when that route is matched during a request, the view callable named by the `view` attribute is invoked.

Typically, an application that uses only URL dispatch won't perform any calls to `pyramid.config.Configurator.add_view()` in its startup code.

18.1.2 Traversal Only

An application that uses only traversal will have view configuration declarations that look like this:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_view('mypackage.views.foobar', name='foobar')
4 config.add_view('mypackage.views.bazbuz', name='bazbuz')
```

When the above configuration is applied to an application, the `mypackage.views.foobar` view callable above will be called when the URL `/foobar` is visited. Likewise, the view `mypackage.views.bazbuz` will be called when the URL `/bazbuz` is visited.

Typically, an application that uses traversal exclusively won't perform any calls to `pyramid.config.Configurator.add_route()` in its startup code.

18.2 Hybrid Applications

Either traversal or url dispatch alone can be used to create a Pyramid application. However, it is also possible to combine the concepts of traversal and url dispatch when building an application: the result is a hybrid application. In a hybrid application, traversal is performed *after* a particular route has matched.

A hybrid application is a lot more like a “pure” traversal-based application than it is like a “pure” URL-dispatch based application. But unlike in a “pure” traversal-based application, in a hybrid application, *traversal* is performed during a request after a route has already matched. This means that the URL pattern that represents the `pattern` argument of a route must match the `PATH_INFO` of a request, and after the route pattern has matched, most of the “normal” rules of traversal with respect to *resource location* and *view lookup* apply.

There are only four real differences between a purely traversal-based application and a hybrid application:

- In a purely traversal based application, no routes are defined; in a hybrid application, at least one route will be defined.
- In a purely traversal based application, the root object used is global, implied by the *root factory* provided at startup time; in a hybrid application, the *root* object at which traversal begins may be varied on a per-route basis.
- In a purely traversal-based application, the `PATH_INFO` of the underlying *WSGI* environment is used wholesale as a traversal path; in a hybrid application, the traversal path is not the entire `PATH_INFO` string, but a portion of the URL determined by a matching pattern in the matched route configuration's pattern.
- In a purely traversal based application, view configurations which do not mention a `route_name` argument are considered during *view lookup*; in a hybrid application, when a route is matched, only view configurations which mention that route's name as a `route_name` are considered during *view lookup*.

More generally, a hybrid application *is* a traversal-based application except:

- the traversal *root* is chosen based on the route configuration of the route that matched instead of from the `root_factory` supplied during application startup configuration.
- the traversal *path* is chosen based on the route configuration of the route that matched rather than from the `PATH_INFO` of a request.
- the set of views that may be chosen during *view lookup* when a route matches are limited to those which specifically name a `route_name` in their configuration that is the same as the matched route's name.

To create a hybrid mode application, use a *route configuration* that implies a particular *root factory* and which also includes a `pattern` argument that contains a special dynamic part: either `*traverse` or `*subpath`.

18.2.1 The Root Object for a Route Match

A hybrid application implies that traversal is performed during a request after a route has matched. Traversal, by definition, must always begin at a root object. Therefore it's important to know *which* root object will be traversed after a route has matched.

Figuring out which *root* object results from a particular route match is straightforward. When a route is matched:

- If the route’s configuration has a `factory` argument which points to a *root factory* callable, that callable will be called to generate a *root* object.
- If the route’s configuration does not have a `factory` argument, the *global root factory* will be called to generate a *root* object. The global root factory is the callable implied by the `root_factory` argument passed to the `Configurator` at application startup time.
- If a `root_factory` argument is not provided to the `Configurator` at startup time, a *default* root factory is used. The default root factory is used to generate a root object.



Root factories related to a route were explained previously within *Route Factories*. Both the global root factory and default root factory were explained previously within *The Resource Tree*.

18.2.2 Using `*traverse` In a Route Pattern

A hybrid application most often implies the inclusion of a route configuration that contains the special token `*traverse` at the end of a route’s pattern:

```
1 config.add_route('home', '{foo}/{bar}/*traverse')
```

A `*traverse` token at the end of the pattern in a route’s configuration implies a “remainder” *capture* value. When it is used, it will match the remainder of the path segments of the URL. This remainder becomes the path used to perform traversal.



The `*remainder` route pattern syntax is explained in more detail within *Route Pattern Syntax*.

Note that unlike the examples provided within *URL Dispatch*, the `add_route` configuration statement named previously does not pass a `view` argument. This is because a hybrid mode application relies on *traversal* to do *resource location* and *view lookup* instead of invariably invoking a specific view callable named directly within the matched route’s configuration.

Because the pattern of the above route ends with `*traverse`, when this route configuration is matched during a request, Pyramid will attempt to use *traversal* against the *root* object implied by the *root factory* that is implied by the route’s configuration. Since no `root_factory` argument is explicitly specified for this route, this will either be the *global* root factory for the application, or the *default* root factory. Once *traversal* has found a *context* resource, *view lookup* will be invoked in almost exactly the same way it would have been invoked in a “pure” traversal-based application.

Let’s assume there is no *global root factory* configured in this application. The *default root factory* cannot be traversed: it has no useful `__getitem__` method. So we’ll need to associate this route configuration with a custom root factory in order to create a useful hybrid application. To that end, let’s imagine that we’ve created a root factory that looks like so in a module named `routes.py`:

```

1 class Resource(object):
2     def __init__(self, subobjects):
3         self.subobjects = subobjects
4
5     def __getitem__(self, name):
6         return self.subobjects[name]
7
8 root = Traversable(
9     {'a':Resource({'b':Resource({'c':Resource({})})})})
10 )
11
12 def root_factory(request):
13     return root

```

Above, we’ve defined a (bogus) resource tree that can be traversed, and a `root_factory` function that can be used as part of a particular route configuration statement:

```

1 config.add_route('home', '{foo}/{bar}/*traverse',
2                   factory='mypackage.routes.root_factory')

```

The factory above points at the function we’ve defined. It will return an instance of the `Traversable` class as a root object whenever this route is matched. Instances of the “Resource” class can be used for tree traversal because they have a `__getitem__` method that does something nominally useful. Since traversal uses `__getitem__` to walk the resources of a resource tree, using traversal against the root resource implied by our route statement is a reasonable thing to do.



We could have also used our `root_factory` callable as the `root_factory` argument of the `Configurator` constructor, instead of associating it with a particular route inside the route’s configuration. Every hybrid route configuration that is matched but which does *not* name a factory attribute will use the use global `root_factory` function to generate a root object.

When the route configuration named `home` above is matched during a request, the matchdict generated will be based on its pattern: `{foo}/{bar}/*traverse`. The “capture value” implied by the `*traverse` element in the pattern will be used to traverse the resource tree in order to find a context resource, starting from the root object returned from the root factory. In the above example, the `root` object found will be the instance named `root` in `routes.py`.

If the URL that matched a route with the pattern `{foo}/{bar}/*traverse`, is `http://example.com/one/two/a/b/c`, the traversal path used against the root object will be `a/b/c`. As a result, Pyramid will attempt to traverse through the edges `a`, `b`, and `c`, beginning at the root object.

18. COMBINING TRAVERSAL AND URL DISPATCH

In our above example, this particular set of traversal steps will mean that the *context* resource of the view would be the Traversable object we've named *c* in our bogus resource tree and the *view name* resulting from traversal will be the empty string; if you need a refresher about why this outcome is presumed, see *The Traversal Algorithm*.

At this point, a suitable view callable will be found and invoked using *view lookup* as described in *View Configuration*, but with a caveat: in order for view lookup to work, we need to define a view configuration that will match when *view lookup* is invoked after a route matches:

```
1 config.add_route('home', '{foo}/{bar}/*traverse',
2                 factory='mypackage.routes.root_factory')
3 config.add_view('mypackage.views.myview', route_name='home')
```

Note that the above call to `add_view()` includes a `route_name` argument. View configurations that include a `route_name` argument are meant to associate a particular view declaration with a route, using the route's name, in order to indicate that the view should *only be invoked when the route matches*.

Calls to `add_view()` may pass a `route_name` attribute, which refers to the value of an existing route's name argument. In the above example, the route name is `home`, referring to the name of the route defined above it.

The above `mypackage.views.myview` view callable will be invoked when:

- the route named “home” is matched
- the *view name* resulting from traversal is the empty string.
- the *context* resource is any object.

It is also possible to declare alternate views that may be invoked when a hybrid route is matched:

```
1 config.add_route('home', '{foo}/{bar}/*traverse',
2                 factory='mypackage.routes.root_factory')
3 config.add_view('mypackage.views.myview', name='home')
4 config.add_view('mypackage.views.another_view', name='another',
5                 route_name='home')
```

The `add_view` call for `mypackage.views.another_view` above names a different view and, more importantly, a different *view name*. The above `mypackage.views.another_view` view will be invoked when:

- the route named “home” is matched

- the *view name* resulting from traversal is `another`.
- the *context* resource is any object.

For instance, if the URL `http://example.com/one/two/a/another` is provided to an application that uses the previously mentioned resource tree, the `mypackage.views.another` view callable will be called instead of the `mypackage.views.myview` view callable because the *view name* will be `another` instead of the empty string.

More complicated matching can be composed. All arguments to *route* configuration statements and *view* configuration statements are supported in hybrid applications (such as *predicate* arguments).

18.2.3 Using the `traverse` Argument In a Route Definition

Rather than using the `*traverse` remainder marker in a pattern, you can use the `traverse` argument to the `add_route()` method.

When you use the `*traverse` remainder marker, the traversal path is limited to being the remainder segments of a request URL when a route matches. However, when you use the `traverse` argument or attribute, you have more control over how to compose a traversal path.

Here's a use of the `traverse` pattern in a call to `add_route()`:

```
1 config.add_route('abc', '/articles/{article}/edit',
2                 traverse='/articles/{article}')
```

The syntax of the `traverse` argument is the same as it is for `pattern`.

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

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

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

Traversal will begin at the root object implied by this route (either the global root, or the object returned by the `factory` associated with this route).

Making Global Views Match

By default, only view configurations that mention a `route_name` will be found during view lookup when a route that has a `*traverse` in its pattern matches. You can allow views without a `route_name` attribute to match a route by adding the `use_global_views` flag to the route definition. For example, the `myproject.views.bazbuzz` view below will be found if the route named `abc` below is matched and the `PATH_INFO` is `/abc/bazbuzz`, even though the view configuration statement does not have the `route_name="abc"` attribute.

```
1 config.add_route('abc', '/abc/*traverse', use_global_views=True)
2 config.add_view('myproject.views.bazbuzz', name='bazbuzz')
```

18.2.4 Using `*subpath` in a Route Pattern

There are certain extremely rare cases when you'd like to influence the traversal *subpath* when a route matches without actually performing traversal. For instance, the `pyramid.wsgi.wsgiapp2()` decorator and the `pyramid.view.static` helper attempt to compute `PATH_INFO` from the request's subpath, so it's useful to be able to influence this value.

When `*subpath` exists in a pattern, no path is actually traversed, but the traversal algorithm will return a *subpath* list implied by the capture value of `*subpath`. You'll see this pattern most commonly in route declarations that look like this:

```
1 config.add_route('static', '/static/*subpath',
2                   view='mypackage.views.static_view')
```

Where `mypackage.views.static_view` is an instance of `pyramid.view.static`. This effectively tells the static helper to traverse everything in the subpath as a filename.

18.3 Corner Cases

A number of corner case “gotchas” exist when using a hybrid application. We'll detail them here.

18.3.1 Registering a Default View for a Route That Has a `view` Attribute

It is an error to provide *both* a `view` argument to a *route configuration* and a *view configuration* which names a `route_name` that has no name value or the empty name value. For example, this pair of declarations will generate a “conflict” error at startup time.

```

1 config.add_route('home', '{foo}/{bar}/*traverse',
2                 view='myproject.views.home')
3 config.add_view('myproject.views.another', route_name='home')

```

This is because the `view` argument to the `add_route()` above is an *implicit* default view when that route matches. `add_route` calls don't *need* to supply a view attribute. For example, this `add_route` call:

```

1 config.add_route('home', '{foo}/{bar}/*traverse',
2                 view='myproject.views.home')

```

Can also be spelled like so:

```

1 config.add_route('home', '{foo}/{bar}/*traverse')
2 config.add_view('myproject.views.home', route_name='home')

```

The two spellings are logically equivalent. In fact, the former is just a syntactical shortcut for the latter.

18.3.2 Binding Extra Views Against a Route Configuration that Doesn't Have a `*traverse` Element In Its Pattern

Here's another corner case that just makes no sense:

```

1 config.add_route('abc', '/abc', view='myproject.views.abc')
2 config.add_view('myproject.views.bazbuzz', name='bazbuzz',
3               route_name='abc')

```

The above view declaration is useless, because it will never be matched when the route it references has matched. Only the view associated with the route itself (`myproject.views.abc`) will ever be invoked when the route matches, because the default view is always invoked when a route matches and when no post-match traversal is performed.

To make the above view declaration useful, the special `*traverse` token must end the route's pattern. For example:

```

1 config.add_route('abc', '/abc/*traverse', view='myproject.views.abc')
2 config.add_view('myproject.views.bazbuzz', name='bazbuzz',
3               route_name='abc')

```

With the above configuration, the `myproject.views.bazbuzz` view will be invoked when the request URI is `/abc/bazbuzz`, assuming there is no object contained by the root object with the key `bazbuzz`. A different request URI, such as `/abc/foo/bar`, would invoke the default `myproject.views.abc` view.

INTERNATIONALIZATION AND LOCALIZATION

Internationalization (i18n) is the act of creating software with a user interface that can potentially be displayed in more than one language or cultural context. *Localization* (l10n) is the process of displaying the user interface of an internationalized application in a *particular* language or cultural context.

Pyramid offers internationalization and localization subsystems that can be used to translate the text of buttons, error messages and other software- and template-defined values into the native language of a user of your application.

19.1 Creating a Translation String

While you write your software, you can insert specialized markup into your Python code that makes it possible for the system to translate text values into the languages used by your application's users. This markup creates a *translation string*. A translation string is an object that behaves mostly like a normal Unicode object, except that it also carries around extra information related to its job as part of the Pyramid translation machinery.

19.1.1 Using The `TranslationString` Class

The most primitive way to create a translation string is to use the `pyramid.i18n.TranslationString` callable:

19. INTERNATIONALIZATION AND LOCALIZATION

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add')
```

This creates a Unicode-like object that is a `TranslationString`.



For people more familiar with *Zope i18n*, a `TranslationString` is a lot like a `zope.i18nmessageid.Message` object. It is not a subclass, however. For people more familiar with *Pylons* or *Django i18n*, using a `TranslationString` is a lot like using “lazy” versions of related `gettext` APIs.

The first argument to `TranslationString` is the `msgid`; it is required. It represents the key into the translation mappings provided by a particular localization. The `msgid` argument must be a Unicode object or an ASCII string. The `msgid` may optionally contain *replacement markers*. For instance:

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add ${number}')
```

Within the string above, `${number}` is a replacement marker. It will be replaced by whatever is in the *mapping* for a translation string. The mapping may be supplied at the same time as the replacement marker itself:

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add ${number}', mapping={'number':1})
```

Any number of replacement markers can be present in the `msgid` value, any number of times. Only markers which can be replaced by the values in the *mapping* will be replaced at translation time. The others will not be interpolated and will be output literally.

A translation string should also usually carry a *domain*. The domain represents a translation category to disambiguate it from other translations of the same `msgid`, in case they conflict.

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('Add ${number}', mapping={'number':1},
3                       domain='form')
```

The above translation string named a domain of `form`. A *translator* function will often use the domain to locate the right translator file on the filesystem which contains translations for a given domain. In this case, if it were trying to translate our `msgid` to German, it might try to find a translation from a *gettext* file within a *translation directory* like this one:

```
locale/de/LC_MESSAGES/form.mo
```

In other words, it would want to take translations from the `form.mo` translation file in the German language.

Finally, the `TranslationString` constructor accepts a `default` argument. If a `default` argument is supplied, it replaces usages of the `msgid` as the *default value* for the translation string. When `default` is `None`, the `msgid` value passed to a `TranslationString` is used as an implicit message identifier. Message identifiers are matched with translations in translation files, so it is often useful to create translation strings with “opaque” message identifiers unrelated to their default text:

```
1 from pyramid.i18n import TranslationString
2 ts = TranslationString('add-number', default='Add ${number}',
3                       domain='form', mapping={'number':1})
```

When default text is used, Default text objects may contain replacement values.

19.1.2 Using the `TranslationStringFactory` Class

Another way to generate a translation string is to use the `TranslationStringFactory` object. This object is a *translation string factory*. Basically a translation string factory presets the domain value of any *translation string* generated by using it. For example:

```
1 from pyramid.i18n import TranslationStringFactory
2 _ = TranslationStringFactory('pyramid')
3 ts = _('Add ${number}', msgid='add-number', mapping={'number':1})
```



We assigned the translation string factory to the name `_`. This is a convention which will be supported by translation file generation tools.

After assigning `_` to the result of a `TranslationStringFactory()`, the subsequent result of calling `_` will be a `TranslationString` instance. Even though a domain value was not passed to `_` (as would have been necessary if the `TranslationString` constructor were used instead of a translation string factory), the `domain` attribute of the resulting translation string will be `pyramid`. As a result, the previous code example is completely equivalent (except for spelling) to:

```
1 from pyramid.i18n import TranslationString as _
2 ts = _('Add ${number}', msgid='add-number', mapping={'number':1},
3       domain='pyramid')
```

You can set up your own translation string factory much like the one provided above by using the `TranslationStringFactory` class. For example, if you'd like to create a translation string factory which presets the domain value of generated translation strings to `form`, you'd do something like this:

```
1 from pyramid.i18n import TranslationStringFactory
2 _ = TranslationStringFactory('form')
3 ts = _('Add ${number}', msgid='add-number', mapping={'number':1})
```

Creating a unique domain for your application via a translation string factory is best practice. Using your own unique translation domain allows another person to reuse your application without needing to merge your translation files with his own. Instead, he can just include your package's *translation directory* via the `pyramid.config.Configurator.add_translation_dirs()` method.



For people familiar with Zope internationalization, a `TranslationStringFactory` is a lot like a `zope.i18nmessageid.MessageFactory` object. It is not a subclass, however.

19.2 Working With `gettext` Translation Files

The basis of Pyramid translation services is GNU *gettext*. Once your application source code files and templates are marked up with translation markers, you can work on translations by creating various kinds of `gettext` files.



The steps a developer must take to work with *gettext message catalog* files within a Pyramid application are very similar to the steps a *Pylons* developer must take to do the same. See the *Pylons* internationalization documentation for more information.

GNU `gettext` uses three types of files in the translation framework, `.pot` files, `.po` files and `.mo` files.

`.pot` (Portable Object Template) files

A `.pot` file is created by a program which searches through your project's source code and which picks out every *message identifier* passed to one of the `'_()` functions (eg. *translation string* constructions). The list of all message identifiers is placed into a `.pot` file, which serves as a template for creating `.po` files.

`.po` (Portable Object) files

The list of messages in a `.pot` file are translated by a human to a particular language; the result is saved as a `.po` file.

`.mo` (Machine Object) files

A `.po` file is turned into a machine-readable binary file, which is the `.mo` file. Compiling the translations to machine code makes the localized program run faster.

The tool for working with *gettext* translation files related to a Pyramid application is *Babel*.

19.2.1 Installing Babel

In order for the commands related to working with *gettext* translation files to work properly, you will need to have *Babel* installed into the same environment in which Pyramid is installed.

Installation on UNIX

If the *virtualenv* into which you've installed your Pyramid application lives in `/my/virtualenv`, you can install Babel like so:

```
$ cd /my/virtualenv
$ bin/easy_install Babel
```


Installation on Windows

If the *virtualenv* into which you've installed your Pyramid application lives in `C:\my\virtualenv`, you can install Babel like so:

```
C> cd \my\virtualenv
C> bin\easy_install Babel
```

Changing the `setup.py`

You need to add a few boilerplate lines to your application's `setup.py` file in order to properly generate *gettext* files from your application.

 See *Creating a Pyramid Project* to learn about about the composition of an application's `setup.py` file.

In particular, add the *Babel* distribution to the `install_requires` list and insert a set of references to *Babel message extractors* within the call to `setuptools.setup()` inside your application's `setup.py` file:

```
1  setup(name="mypackage",
2      # ...
3      install_requires = [
4          # ...
5          'Babel',
6      ],
7      message_extractors = { '.': [
8          ('**.py', 'chameleon_python', None),
9          ('**.pt', 'chameleon_xml', None),
10         ]},
11     )
```

The `message_extractors` stanza placed into the `setup.py` file causes the *Babel* message catalog extraction machinery to also consider `** .pt` files when doing message id extraction.

19.2.2 Extracting Messages from Code and Templates

Once *Babel* is installed and your application's `setup.py` file has the correct message extractor references, you may extract a message catalog template from the code and *Chameleon* templates which reside in your Pyramid application. You run a `setup.py` command to extract the messages:

```
$ cd /place/where/myapplication/setup.py/lives
$ mkdir -p myapplication/locale
$ python setup.py extract_messages
```

The message catalog `.pot` template will end up in:

`myapplication/locale/myapplication.pot`.

Translation Domains

The name `myapplication` above in the filename `myapplication.pot` denotes the *translation domain* of the translations that must be performed to localize your application. By default, the translation domain is the *project* name of your Pyramid application.

To change the translation domain of the extracted messages in your project, edit the `setup.cfg` file of your application. The default `setup.cfg` file of a Paster-generated Pyramid application has stanzas in it that look something like the following:

```
1  [compile_catalog]
2  directory = myproject/locale
3  domain = MyProject
4  statistics = true
5
6  [extract_messages]
7  add_comments = TRANSLATORS:
8  output_file = myproject/locale/MyProject.pot
9  width = 80
10
11 [init_catalog]
12 domain = MyProject
13 input_file = myproject/locale/MyProject.pot
14 output_dir = myproject/locale
15
16 [update_catalog]
17 domain = MyProject
18 input_file = myproject/locale/MyProject.pot
19 output_dir = myproject/locale
20 previous = true
```

In the above example, the project name is `MyProject`. To indicate that you'd like the domain of your translations to be `mydomain` instead, change the `setup.cfg` file stanzas to look like so:

```
1  [compile_catalog]
2  directory = myproject/locale
3  domain = mydomain
4  statistics = true
5
6  [extract_messages]
7  add_comments = TRANSLATORS:
8  output_file = myproject/locale/mydomain.pot
9  width = 80
10
```

```
11 [init_catalog]
12 domain = mydomain
13 input_file = myproject/locale/mydomain.pot
14 output_dir = myproject/locale
15
16 [update_catalog]
17 domain = mydomain
18 input_file = myproject/locale/mydomain.pot
19 output_dir = myproject/locale
20 previous = true
```

19.2.3 Initializing a Message Catalog File

Once you've extracted messages into a `.pot` file (see *Extracting Messages from Code and Templates*), to begin localizing the messages present in the `.pot` file, you need to generate at least one `.po` file. A `.po` file represents translations of a particular set of messages to a particular locale. Initialize a `.po` file for a specific locale from a pre-generated `.pot` template by using the `setup.py init_catalog` command:

```
$ cd /place/where/myapplication/setup.py/lives
$ python setup.py init_catalog -l es
```

By default, the message catalog `.po` file will end up in:

```
myapplication/locale/es/LC_MESSAGES/myapplication.po.
```

Once the file is there, it can be worked on by a human translator. One tool which may help with this is Poedit.

Note that Pyramid itself ignores the existence of all `.po` files. For a running application to have translations available, a `.mo` file must exist. See *Compiling a Message Catalog File*.

19.2.4 Updating a Catalog File

If more translation strings are added to your application, or translation strings change, you will need to update existing `.po` files based on changes to the `.pot` file, so that the new and changed messages can also be translated or re-translated.

First, regenerate the `.pot` file as per *Extracting Messages from Code and Templates*. Then use the `setup.py update_catalog` command.

```
$ cd /place/where/myapplication/setup.py/lives
$ python setup.py update_catalog
```

19.2.5 Compiling a Message Catalog File

Finally, to prepare an application for performing actual runtime translations, compile `.po` files to `.mo` files:

```
$ cd /place/where/myapplication/setup.py/lives
$ python setup.py compile_catalog
```

This will create a `.mo` file for each `.po` file in your application. As long as the *translation directory* in which the `.mo` file ends up in is configured into your application, these translations will be available to Pyramid.

19.3 Using a Localizer

A *localizer* is an object that allows you to perform translation or pluralization “by hand” in an application. You may use the `pyramid.i18n.get_localizer()` function to obtain a *localizer*. This function will return either the localizer object implied by the active *locale negotiator* or a default localizer object if no explicit locale negotiator is registered.


```
1 from pyramid.i18n import get_localizer
2
3 def aview(request):
4     locale = get_localizer(request)
```

19.3.1 Performing a Translation

A *localizer* has a `translate` method which accepts either a *translation string* or a Unicode string and which returns a Unicode object representing the translation. So, generating a translation in a view component of an application might look like so:

```
1 from pyramid.i18n import get_localizer
2 from pyramid.i18n import TranslationString
3
4 ts = TranslationString('Add ${number}', mapping={'number':1},
5                       domain='pyramid')
6
7 def aview(request):
8     localizer = get_localizer(request)
9     translated = localizer.translate(ts) # translation string
10    # ... use translated ...
```

The `get_localizer()` function will return a `pyramid.i18n.Localizer` object bound to the locale name represented by the request. The translation returned from its `pyramid.i18n.Localizer.translate()` method will depend on the domain attribute of the provided translation string as well as the locale of the localizer.

 If you're using *Chameleon* templates, you don't need to pre-translate translation strings this way. See *Chameleon Template Support for Translation Strings*.

19.3.2 Performing a Pluralization

A *localizer* has a `pluralize` method with the following signature:

```
1 def pluralize(singular, plural, n, domain=None, mapping=None):
2     ...
```

The `singular` and `plural` arguments should each be a Unicode value representing a *message identifier*. `n` should be an integer. `domain` should be a *translation domain*, and `mapping` should be a dictionary that is used for *replacement value* interpolation of the translated string. If `n` is plural for the current locale, `pluralize` will return a Unicode translation for the message id `plural`, otherwise it will return a Unicode translation for the message id `singular`.

The arguments provided as `singular` and/or `plural` may also be *translation string* objects, but the domain and mapping information attached to those objects is ignored.

```
1 from pyramid.i18n import get_localizer
2
3 def aview(request):
4     localizer = get_localizer(request)
5     translated = localizer.pluralize('Item', 'Items', 1, 'mydomain')
6     # ... use translated ...
```

19.4 Obtaining the Locale Name for a Request

You can obtain the locale name related to a request by using the `pyramid.i18n.get_locale_name()` function.

```
1 from pyramid.i18n import get_locale_name
2
3 def aview(request):
4     locale_name = get_locale_name(request)
```

This returns the locale name negotiated by the currently active *locale negotiator* or the *default locale name* if the locale negotiator returns `None`. You can change the default locale name by changing the `default_locale_name` setting; see *Default Locale Name*.

Once `get_locale_name()` is first run, the locale name is stored on the request object. Subsequent calls to `get_locale_name()` will return the stored locale name without invoking the *locale negotiator*. To avoid this caching, you can use the `pyramid.i18n.negotiate_locale_name()` function:

```
1 from pyramid.i18n import negotiate_locale_name
2
3 def aview(request):
4     locale_name = negotiate_locale_name(request)
```

You can also obtain the locale name related to a request using the `locale_name` attribute of a *localizer*.

```
1 from pyramid.i18n import get_localizer
2
3 def aview(request):
4     localizer = get_localizer(request)
5     locale_name = localizer.locale_name
```

Obtaining the locale name as an attribute of a localizer is equivalent to obtaining a locale name by calling the `get_locale_name()` function.

19.5 Performing Date Formatting and Currency Formatting

Pyramid does not itself perform date and currency formatting for different locales. However, *Babel* can help you do this via the `babel.core.Locale` class. The Babel documentation for this class provides

19. INTERNATIONALIZATION AND LOCALIZATION

minimal information about how to perform date and currency related locale operations. See *Installing Babel* for information about how to install Babel.

The `babel.core.Locale` class requires a *locale name* as an argument to its constructor. You can use Pyramid APIs to obtain the locale name for a request to pass to the `babel.core.Locale` constructor; see *Obtaining the Locale Name for a Request*. For example:

```
1 from babel.core import Locale
2 from pyramid.i18n import get_locale_name
3
4 def aview(request):
5     locale_name = get_locale_name(request)
6     locale = Locale(locale_name)
```

19.6 Chameleon Template Support for Translation Strings

When a *translation string* is used as the subject of textual rendering by a *Chameleon* template renderer, it will automatically be translated to the requesting user's language if a suitable translation exists. This is true of both the ZPT and text variants of the Chameleon template renderers.

For example, in a Chameleon ZPT template, the translation string represented by "some_translation_string" in each example below will go through translation before being rendered:

```
1 <span tal:content="some_translation_string"/>
```

```
1 <span tal:replace="some_translation_string"/>
```

```
1 <span>${some_translation_string}</span>
```

```
1 <a tal:attributes="href some_translation_string">Click here</a>
```

The features represented by attributes of the `i18n` namespace of Chameleon will also consult the Pyramid translations. See <http://chameleon.repoze.org/docs/latest/i18n.html#the-i18n-namespace>.



Unlike when Chameleon is used outside of Pyramid, when it is used *within* Pyramid, it does not support use of the `zope.i18n` translation framework. Applications which use Pyramid should use the features documented in this chapter rather than `zope.i18n`.

Third party Pyramid template renderers might not provide this support out of the box and may need special code to do an equivalent. For those, you can always use the more manual translation facility described in *Performing a Translation*.

19.7 Mako Pyramid I18N Support

There exists a recipe within the *Pyramid Cookbook* named “Mako Internationalization” which explains how to add idiomatic I18N support to *Mako* templates.

19.8 Localization-Related Deployment Settings

A Pyramid application will have a `default_locale_name` setting. This value represents the *default locale name* used when the *locale negotiator* returns `None`. Pass it to the `Configurator` constructor at startup time:

```
1 from pyramid.config import Configurator
2 config = Configurator(settings={'default_locale_name': 'de'})
```

You may alternately supply a `default_locale_name` via an application’s Paste `.ini` file:

```
1 [app:main]
2 use = egg:MyProject#app
3 reload_templates = true
4 debug_authorization = false
5 debug_notfound = false
6 default_locale_name = de
```

If this value is not supplied via the `Configurator` constructor or via a Paste config file, it will default to `en`.

If this setting is supplied within the Pyramid application `.ini` file, it will be available as a settings key:

```
1 from pyramid.threadlocal import get_current_registry
2 settings = get_current_registry().settings
3 default_locale_name = settings['default_locale_name']
```

19.9 “Detecting” Available Languages

Other systems provide an API that returns the set of “available languages” as indicated by the union of all languages in all translation directories on disk at the time of the call to the API.

It is by design that Pyramid doesn’t supply such an API. Instead, the application itself is responsible for knowing the “available languages”. The rationale is this: any particular application deployment must always know which languages it should be translatable to anyway, regardless of which translation files are on disk.

Here’s why: it’s not a given that because translations exist in a particular language within the registered set of translation directories that this particular deployment wants to allow translation to that language. For example, some translations may exist but they may be incomplete or incorrect. Or there may be translations to a language but not for all translation domains.

Any nontrivial application deployment will always need to be able to selectively choose to allow only some languages even if that set of languages is smaller than all those detected within registered translation directories. The easiest way to allow for this is to make the application entirely responsible for knowing which languages are allowed to be translated to instead of relying on the framework to divine this information from translation directory file info.

You can set up a system to allow a deployer to select available languages based on convention by using the `pyramid.settings` mechanism:

Allow a deployer to modify your application’s PasteDeploy `.ini` file:

```
1 [app:main]
2 use = egg:MyProject#app
3 # ...
4 available_languages = fr de en ru
```

Then as a part of the code of a custom *locale negotiator*:

```
1 from pyramid.threadlocal import get_current_registry
2 settings = get_current_registry().settings
3 languages = settings['available_languages'].split()
```

This is only a suggestion. You can create your own “available languages” configuration scheme as necessary.

19.10 Activating Translation

By default, a Pyramid application performs no translation. To turn translation on, you must:

- add at least one *translation directory* to your application.
- ensure that your application sets the *locale name* correctly.

19.10.1 Adding a Translation Directory

gettext is the underlying machinery behind the Pyramid translation machinery. A translation directory is a directory organized to be useful to *gettext*. A translation directory usually includes a listing of language directories, each of which itself includes an `LC_MESSAGES` directory. Each `LC_MESSAGES` directory should contain one or more `.mo` files. Each `.mo` file represents a *message catalog*, which is used to provide translations to your application.

Adding a *translation directory* registers all of its constituent *message catalog* files within your Pyramid application to be available to use for translation services. This includes all of the `.mo` files found within all `LC_MESSAGES` directories within each locale directory in the translation directory.

You can add a translation directory imperatively by using the `pyramid.config.Configurator.add_translation_dirs()` during application startup. For example:

```
1 from pyramid.config import Configurator
2 config.add_translation_dirs('my.application:locale/',
3                             'another.application:locale/')
```

A message catalog in a translation directory added via `add_translation_dirs()` will be merged into translations from a message catalog added earlier if both translation directories contain translations for the same locale and *translation domain*.

19.10.2 Setting the Locale

When the *default locale negotiator* (see *The Default Locale Negotiator*) is in use, you can inform Pyramid of the current locale name by doing any of these things before any translations need to be performed:

- Set the `__LOCALE__` attribute of the request to a valid locale name (usually directly within view code). E.g. `request.__LOCALE__ = 'de'`.
- Ensure that a valid locale name value is in the `request.params` dictionary under the key named `__LOCALE__`. This is usually the result of passing a `__LOCALE__` value in the query string or in the body of a form post associated with a request. For example, visiting `http://my.application?__LOCALE__=de`.
- Ensure that a valid locale name value is in the `request.cookies` dictionary under the key named `__LOCALE__`. This is usually the result of setting a `__LOCALE__` cookie in a prior response, e.g. `response.set_cookie('__LOCALE__', 'de')`.



If this locale negotiation scheme is inappropriate for a particular application, you can configure a custom *locale negotiator* function into that application as required. See *Using a Custom Locale Negotiator*.

19.11 Locale Negotiators

A *locale negotiator* informs the operation of a *localizer* by telling it what *locale name* is related to a particular request. A locale negotiator is a bit of code which accepts a request and which returns a *locale name*. It is consulted when `pyramid.i18n.Localizer.translate()` or `pyramid.i18n.Localizer.pluralize()` is invoked. It is also consulted when `get_locale_name()` or `negotiate_locale_name()` is invoked.

19.11.1 The Default Locale Negotiator

Most applications can make use of the default locale negotiator, which requires no additional coding or configuration.

The default locale negotiator implementation named `default_locale_negotiator` uses the following set of steps to determine the locale name.

- First, the negotiator looks for the `__LOCALE__` attribute of the request object (possibly set directly by view code or by a listener for an *event*).
- Then it looks for the `request.params['_LOCALE_']` value.
- Then it looks for the `request.cookies['_LOCALE_']` value.
- If no locale can be found via the request, it falls back to using the *default locale name* (see *Localization-Related Deployment Settings*).
- Finally, if the default locale name is not explicitly set, it uses the locale name `en`.

19.11.2 Using a Custom Locale Negotiator

Locale negotiation is sometimes policy-laden and complex. If the (simple) default locale negotiation scheme described in *Activating Translation* is inappropriate for your application, you may create and a special *locale negotiator*. Subsequently you may override the default locale negotiator by adding your newly created locale negotiator to your application's configuration.

A locale negotiator is simply a callable which accepts a request and returns a single *locale name* or `None` if no locale can be determined.

Here's an implementation of a simple locale negotiator:

```
1 def my_locale_negotiator(request):
2     locale_name = request.params.get('my_locale')
3     return locale_name
```

If a locale negotiator returns `None`, it signifies to Pyramid that the default application locale name should be used.

You may add your newly created locale negotiator to your application's configuration by passing an object which can act as the negotiator (or a *dotted Python name* referring to the object) as the `locale_negotiator` argument of the `Configurator` instance during application startup. For example:

```
1 from pyramid.config import Configurator
2 config = Configurator(locale_negotiator=my_locale_negotiator)
```

Alternately, use the `pyramid.config.Configurator.set_locale_negotiator()` method.

For example:

19. INTERNATIONALIZATION AND LOCALIZATION

```
1 from pyramid.config import Configurator
2 config = Configurator()
3 config.set_locale_negotiator(my_locale_negotiator)
```

VIRTUAL HOSTING

“Virtual hosting” is, loosely, the act of serving a Pyramid application or a portion of a Pyramid application under a URL space that it does not “naturally” inhabit.

Pyramid provides facilities for serving an application under a URL “prefix”, as well as serving a *portion* of a *traversal* based application under a root URL.

20.1 Hosting an Application Under a URL Prefix

Pyramid supports a common form of virtual hosting whereby you can host a Pyramid application as a “subset” of some other site (e.g. under `http://example.com/mypyramidapplication/` as opposed to under `http://example.com/`).

If you use a “pure Python” environment, this functionality is provided by Paste’s `urlmap` “composite” WSGI application. Alternately, you can use `mod_wsgi` to serve your application, which handles this virtual hosting translation for you “under the hood”.

If you use the `urlmap` composite application “in front” of a Pyramid application or if you use `mod_wsgi` to serve up a Pyramid application, nothing special needs to be done within the application for URLs to be generated that contain a prefix. `paste.urlmap` and `mod_wsgi` manipulate the `WSGI` environment in such a way that the `PATH_INFO` and `SCRIPT_NAME` variables are correct for some given prefix.

Here’s an example of a PasteDeploy configuration snippet that includes a `urlmap` composite.

```
1 [app:mypyramidapp]
2 use = egg:mypyramidapp#app
3
4 [composite:main]
5 use = egg:Paste#urlmap
6 /pyramidapp = mypyramidapp
```

This “roots” the Pyramid application at the prefix `/pyramidapp` and serves up the composite as the “main” application in the file.



If you’re using an Apache server to proxy to a Paste `urlmap` composite, you may have to use the `ProxyPreserveHost` directive to pass the original `HTTP_HOST` header along to the application, so URLs get generated properly. As of this writing the `urlmap` composite does not seem to respect the `HTTP_X_FORWARDED_HOST` parameter, which will contain the original host header even if `HTTP_HOST` is incorrect.

If you use `mod_wsgi`, you do not need to use a `composite` application in your `.ini` file. The `WSGIScriptAlias` configuration setting in a `mod_wsgi` configuration does the work for you:

```
1 WSGIScriptAlias /pyramidapp /Users/chris/projects/modwsgi/env/pyramid.wsgi
```

In the above configuration, we root a Pyramid application at `/pyramidapp` within the Apache configuration.

20.2 Virtual Root Support

Pyramid also supports “virtual roots”, which can be used in *traversal*-based (but not *URL dispatch*-based) applications.

Virtual root support is useful when you’d like to host some resource in a Pyramid resource tree as an application under a URL pathname that does not include the resource path itself. For example, you might want to serve the object at the traversal path `/cms` as an application reachable via `http://example.com/` (as opposed to `http://example.com/cms`).

To specify a virtual root, cause an environment variable to be inserted into the WSGI environ named `HTTP_X_VHM_ROOT` with a value that is the absolute pathname to the resource object in the resource tree that should behave as the “root” resource. As a result, the traversal machinery will respect this value during traversal (prepending it to the `PATH_INFO` before traversal starts), and the `pyramid.url.resource_url()` API will generate the “correct” virtually-rooted URLs.

An example of an Apache `mod_proxy` configuration that will host the `/cms` subobject as `http://www.example.com/` using this facility is below:

```
1 NameVirtualHost *:80
2
3 <VirtualHost *:80>
4     ServerName www.example.com
5     RewriteEngine On
6     RewriteRule ^/(.*) http://127.0.0.1:6543/$1 [L,P]
7     ProxyPreserveHost on
8     RequestHeader add X-Vhm-Root /cms
9 </VirtualHost>
```



Use of the `RequestHeader` directive requires that the Apache `mod_headers` module be available in the Apache environment you're using.

For a Pyramid application running under `mod_wsgi`, the same can be achieved using `SetEnv`:

```
1 <Location />
2     SetEnv HTTP_X_VHM_ROOT /cms
3 </Location>
```

Setting a virtual root has no effect when using an application based on *URL dispatch*.

20.3 Further Documentation and Examples

The API documentation in *pyramid.traversal* documents a `pyramid.traversal.virtual_root()` API. When called, it returns the virtual root object (or the physical root object if no virtual root has been specified).

Running a Pyramid Application under mod_wsgi has detailed information about using `mod_wsgi` to serve Pyramid applications.

USING EVENTS

An *event* is an object broadcast by the Pyramid framework at interesting points during the lifetime of an application. You don't need to use events in order to create most Pyramid applications, but they can be useful when you want to perform slightly advanced operations. For example, subscribing to an event can allow you to run some code as the result of every new request.

Events in Pyramid are always broadcast by the framework. However, they only become useful when you register a *subscriber*. A subscriber is a function that accepts a single argument named *event*:

```
1 def mysubscriber(event):  
2     print event
```

The above is a subscriber that simply prints the event to the console when it's called.

The mere existence of a subscriber function, however, is not sufficient to arrange for it to be called. To arrange for the subscriber to be called, you'll need to use the `pyramid.config.Configurator.add_subscriber()` method or you'll need to use the `pyramid.events.subscriber()` decorator to decorate a function found via a *scan*.

21.1 Configuring an Event Listener Imperatively

You can imperatively configure a subscriber function to be called for some event type via the `add_subscriber()` method (see also *Configurator*):

21. USING EVENTS

```
1 from pyramid.events import NewRequest
2
3 from subscribers import mysubscriber
4
5 # "config" below is assumed to be an instance of a
6 # pyramid.config.Configurator object
7
8 config.add_subscriber(mysubscriber, NewRequest)
```

The first argument to `add_subscriber()` is the subscriber function (or a *dotted Python name* which refers to a subscriber callable); the second argument is the event type.

21.2 Configuring an Event Listener Using a Decorator

You can configure a subscriber function to be called for some event type via the `pyramid.events.subscriber()` function.

```
1 from pyramid.events import NewRequest
2 from pyramid.events import subscriber
3
4 @subscriber(NewRequest)
5 def mysubscriber(event):
6     event.request.foo = 1
```

When the `subscriber()` decorator is used a *scan* must be performed against the package containing the decorated function for the decorator to have any effect.

Either of the above registration examples implies that every time the Pyramid framework emits an event object that supplies an `pyramid.events.NewRequest` interface, the `mysubscriber` function will be called with an *event* object.

As you can see, a subscription is made in terms of a *class* (such as `pyramid.events.NewResponse`). The event object sent to a subscriber will always be an object that possesses an *interface*. For `pyramid.events.NewResponse`, that interface is `pyramid.interfaces.INewResponse`. The interface documentation provides information about available attributes and methods of the event objects.

The return value of a subscriber function is ignored. Subscribers to the same event type are not guaranteed to be called in any particular order relative to each other.

All the concrete Pyramid event types are documented in the *pyramid.events* API documentation.

21.3 An Example

If you create event listener functions in a `subscribers.py` file in your application like so:

```
1 def handle_new_request(event):
2     print 'request', event.request
3
4 def handle_new_response(event):
5     print 'response', event.response
```

You may configure these functions to be called at the appropriate times by adding the following code to your application's configuration startup:

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_subscriber('myproject.subscribers.handle_new_request',
4                       'pyramid.events.NewRequest')
5 config.add_subscriber('myproject.subscribers.handle_new_response',
6                       'pyramid.events.NewResponse')
```

Either mechanism causes the functions in `subscribers.py` to be registered as event subscribers. Under this configuration, when the application is run, each time a new request or response is detected, a message will be printed to the console.

Each of our subscriber functions accepts an `event` object and prints an attribute of the event object. This begs the question: how can we know which attributes a particular event has?

We know that `pyramid.events.NewRequest` event objects have a `request` attribute, which is a `request` object, because the interface defined at `pyramid.interfaces.INewRequest` says it must. Likewise, we know that `pyramid.interfaces.NewResponse` events have a `response` attribute, which is a response object constructed by your application, because the interface defined at `pyramid.interfaces.INewResponse` says it must (`pyramid.events.NewResponse` objects also have a `request`).

ENVIRONMENT VARIABLES AND .INI FILE SETTINGS

Pyramid behavior can be configured through a combination of operating system environment variables and `.ini` configuration file application section settings. The meaning of the environment variables and the configuration file settings overlap.



Where a configuration file setting exists with the same meaning as an environment variable, and both are present at application startup time, the environment variable setting takes precedence.

The term “configuration file setting name” refers to a key in the `.ini` configuration for your application. The configuration file setting names documented in this chapter are reserved for Pyramid use. You should not use them to indicate application-specific configuration settings.

22.1 Reloading Templates


When this value is true, templates are automatically reloaded whenever they are modified without restarting the application, so you can see changes to templates take effect immediately during development. This flag is meaningful to Chameleon and Mako templates, as well as most third-party template rendering extensions.

Environment Variable Name	Config File Setting Name
<code>PYRAMID_RELOAD_TEMPLATES</code>	<code>reload_templates</code>

22.2 Reloading Assets

Don't cache any asset file data when this value is true. See also *Overriding Assets*.

Environment Variable Name	Config File Setting Name
PYRAMID_RELOAD_ASSETS	reload_assets

 For backwards compatibility purposes, aliases can be used for configuring asset reloading: PYRAMID_RELOAD_RESOURCES (envvar) and reload_resources (config file).

22.3 Debugging Authorization

Print view authorization failure and success information to stderr when this value is true. See also *Debugging View Authorization Failures*.

Environment Variable Name	Config File Setting Name
PYRAMID_DEBUG_AUTHORIZATION	debug_authorization

22.4 Debugging Not Found Errors

Print view-related NotFound debug messages to stderr when this value is true. See also *NotFound Errors*.

Environment Variable Name	Config File Setting Name
PYRAMID_DEBUG_NOTFOUND	debug_notfound

22.5 Debugging Route Matching

Print debugging messages related to *url dispatch* route matching when this value is true. See also *Debugging Route Matching*.

Environment Variable Name	Config File Setting Name
PYRAMID_DEBUG_ROUTE MATCH	debug_routematch

22.6 Debugging All

Turns on all `debug*` settings.

Environment Variable Name	Config File Setting Name
PYRAMID_DEBUG_ALL	debug_all

22.7 Reloading All

Turns on all `reload*` settings.

Environment Variable Name	Config File Setting Name
PYRAMID_RELOAD_ALL	reload_all

22.8 Default Locale Name

The value supplied here is used as the default locale name when a *locale negotiator* is not registered. See also *Localization-Related Deployment Settings*.

Environment Variable Name	Config File Setting Name
PYRAMID_DEFAULT_LOCALE_NAME	default_locale_name

22.9 Mako Template Render Settings

Mako derives additional settings to configure its template renderer that should be set when using it. Many of these settings are optional and only need to be set if they should be different from the default. The Mako Template Renderer uses a subclass of Mako's template lookup and accepts several arguments to configure it.

22.9.1 Mako Directories

The value(s) supplied here are passed in as the template directories. They should be in *asset specification* format, for example: `my.package:templates`.

Config File Setting Name
mako.directories

22.9.2 Mako Module Directory

The value supplied here tells Mako where to store compiled Mako templates. If omitted, compiled templates will be stored in memory. This value should be an absolute path, for example: `%(here)s/data/templates` would use a directory called `data/templates` in the same parent directory as the INI file.

Config File Setting Name
<code>mako.module_directory</code>

22.9.3 Mako Input Encoding

The encoding that Mako templates are assumed to have. By default this is set to `utf-8`. If you wish to use a different template encoding, this value should be changed accordingly.

Config File Setting Name
<code>mako.input_encoding</code>

22.9.4 Mako Error Handler

Python callable which is called whenever Mako compile or runtime exceptions occur. The callable is passed the current context as well as the exception. If the callable returns `True`, the exception is considered to be handled, else it is re-raised after the function completes. Is used to provide custom error-rendering functions.

Config File Setting Name
<code>mako.error_handler</code>

22.9.5 Mako Default Filters

List of string filter names that will be applied to all Mako expressions.

Config File Setting Name
<code>mako.default_filters</code>

22.9.6 Mako Import

String list of Python statements, typically individual “import” lines, which will be placed into the module level preamble of all generated Python modules.

Config File Setting Name
<code>mako.imports</code>

22.9.7 Mako Strict Undefined

`true` or `false`, representing the “strict undefined” behavior of Mako (see Mako Context Variables). By default, this is `false`.

Config File Setting Name
<code>mako.strict_undefined</code>

22.10 Examples

Let’s presume your configuration file is named `MyProject.ini`, and there is a section representing your application named `[app:main]` within the file that represents your Pyramid application. The configuration file settings documented in the above “Config File Setting Name” column would go in the `[app:main]` section. Here’s an example of such a section:

```

1 [app:main]
2 use = egg:MyProject#app
3 reload_templates = true
4 debug_authorization = true

```

You can also use environment variables to accomplish the same purpose for settings documented as such. For example, you might start your Pyramid application using the following command line:

```

$ PYRAMID_DEBUG_AUTHORIZATION=1 PYRAMID_RELOAD_TEMPLATES=1 \
  bin/paster serve MyProject.ini

```

If you started your application this way, your Pyramid application would behave in the same manner as if you had placed the respective settings in the `[app:main]` section of your application's `.ini` file.

If you want to turn all `debug` settings (every setting that starts with `debug_`). on in one fell swoop, you can use `PYRAMID_DEBUG_ALL=1` as an environment variable setting or you may use `debug_all=true` in the config file. Note that this does not affect settings that do not start with `debug_*` such as `reload_templates`.

If you want to turn all `reload` settings (every setting that starts with `reload_`). on in one fell swoop, you can use `PYRAMID_RELOAD_ALL=1` as an environment variable setting or you may use `reload_all=true` in the config file. Note that this does not affect settings that do not start with `reload_*` such as `debug_notfound`.



Specifying configuration settings via environment variables is generally most useful during development, where you may wish to augment or override the more permanent settings in the configuration file. This is useful because many of the reload and debug settings may have performance or security (i.e., disclosure) implications that make them undesirable in a production environment.

22.11 Understanding the Distinction Between `reload_templates` and `reload_assets`

The difference between `reload_assets` and `reload_templates` is a bit subtle. Templates are themselves also treated by Pyramid as asset files (along with other static files), so the distinction can be confusing. It's helpful to read *Overriding Assets* for some context about assets in general.

When `reload_templates` is true, Pyramid takes advantage of the underlying templating systems' ability to check for file modifications to an individual template file. When `reload_templates` is true but `reload_assets` is *not* true, the template filename returned by the `pkg_resources` package (used under the hood by asset resolution) is cached by Pyramid on the first request. Subsequent requests for the same template file will return a cached template filename. The underlying templating system checks for modifications to this particular file for every request. Setting `reload_templates` to True doesn't affect performance dramatically (although it should still not be used in production because it has some effect).

However, when `reload_assets` is true, Pyramid will not cache the template filename, meaning you can see the effect of changing the content of an overridden asset directory for templates without restarting the server after every change. Subsequent requests for the same template file may return different filenames based on the current state of overridden asset directories. Setting `reload_assets` to True affects performance *dramatically*, slowing things down by an order of magnitude for each template rendering. However, it's convenient to enable when moving files around in overridden asset directories. `reload_assets` makes the system *very slow* when templates are in use. Never set `reload_assets` to True on a production system.

UNIT, INTEGRATION, AND FUNCTIONAL TESTING

Unit testing is, not surprisingly, the act of testing a “unit” in your application. In this context, a “unit” is often a function or a method of a class instance. The unit is also referred to as a “unit under test”.

The goal of a single unit test is to test **only** some permutation of the “unit under test”. If you write a unit test that aims to verify the result of a particular codepath through a Python function, you need only be concerned about testing the code that *lives in the function body itself*. If the function accepts a parameter that represents a complex application “domain object” (such as a resource, a database connection, or an SMTP server), the argument provided to this function during a unit test *need not be* and likely *should not be* a “real” implementation object. For example, although a particular function implementation may accept an argument that represents an SMTP server object, and the function may call a method of this object when the system is operating normally that would result in an email being sent, a unit test of this codepath of the function does *not* need to test that an email is actually sent. It just needs to make sure that the function calls the method of the object provided as an argument that *would* send an email if the argument happened to be the “real” implementation of an SMTP server object.

An *integration test*, on the other hand, is a different form of testing in which the interaction between two or more “units” is explicitly tested. Integration tests verify that the components of your application work together. You *might* make sure that an email was actually sent in an integration test.

A *functional test* is a form of integration test in which the application is run “literally”. You would *have to* make sure that an email was actually sent in a functional test, because it tests your code end to end.

It is often considered best practice to write each type of tests for any given codebase. Unit testing often provides the opportunity to obtain better “coverage”: it’s usually possible to supply a unit under test with arguments and/or an environment which causes *all* of its potential codepaths to be executed. This is

usually not as easy to do with a set of integration or functional tests, but integration and functional testing provides a measure of assurance that your “units” work together, as they will be expected to when your application is run in production.

The suggested mechanism for unit and integration testing of a Pyramid application is the Python `unittest` module. Although this module is named `unittest`, it is actually capable of driving both unit and integration tests. A good `unittest` tutorial is available within *Dive Into Python* by Mark Pilgrim.

Pyramid provides a number of facilities that make unit, integration, and functional tests easier to write. The facilities become particularly useful when your code calls into Pyramid -related framework functions.

23.1 Test Set Up and Tear Down

Pyramid uses a “global” (actually *thread local*) data structure to hold on to two items: the current *request* and the current *application registry*. These data structures are available via the `pyramid.threadlocal.get_current_request()` and `pyramid.threadlocal.get_current_registry()` functions, respectively. See *Thread Locals* for information about these functions and the data structures they return.

If your code uses these `get_current_*` functions or calls Pyramid code which uses `get_current_*` functions, you will need to call `pyramid.testing.setUp()` in your test setup and you will need to call `pyramid.testing.tearDown()` in your test teardown. `setUp()` pushes a registry onto the *thread local* stack, which makes the `get_current_*` functions work. It returns a *Configurator* object which can be used to perform extra configuration required by the code under test. `tearDown()` pops the thread local stack.

Normally when a *Configurator* is used directly with the `main` block of a Pyramid application, it defers performing any “real work” until its `.commit` method is called (often implicitly by the `pyramid.config.Configurator.make_wsgi_app()` method). The *Configurator* returned by `setUp()` is an *autocommitting* *Configurator*, however, which performs all actions implied by methods called on it immediately. This is more convenient for unit-testing purposes than needing to call `pyramid.config.Configurator.commit()` in each test after adding extra configuration statements.

The use of the `setUp()` and `tearDown()` functions allows you to supply each unit test method in a test case with an environment that has an isolated registry and an isolated request for the duration of a single test. Here’s an example of using this feature:

```
1 import unittest
2 from pyramid import testing
3
4 class MyTest(unittest.TestCase):
5     def setUp(self):
6         self.config = testing.setUp()
7
8     def tearDown(self):
9         testing.tearDown()
```

The above will make sure that `get_current_registry()` called within a test case method of `MyTest` will return the *application registry* associated with the `config` `Configurator` instance. Each test case method attached to `MyTest` will use an isolated registry.

The `setUp()` and `tearDown()` functions accepts various arguments that influence the environment of the test. See the *pyramid.testing* chapter for information about the extra arguments supported by these functions.

If you also want to make `get_current_request()` return something other than `None` during the course of a single test, you can pass a *request* object into the `pyramid.testing.setUp()` within the `setUp` method of your test:

```
1 import unittest
2 from pyramid import testing
3
4 class MyTest(unittest.TestCase):
5     def setUp(self):
6         request = testing.DummyRequest()
7         self.config = testing.setUp(request=request)
8
9     def tearDown(self):
10        testing.tearDown()
```

If you pass a *request* object into `pyramid.testing.setUp()` within your test case's `setUp`, any test method attached to the `MyTest` test case that directly or indirectly calls `get_current_request()` will receive the request object. Otherwise, during testing, `get_current_request()` will return `None`. We use a “dummy” request implementation supplied by `pyramid.testing.DummyRequest` because it's easier to construct than a “real” Pyramid request object.

23.1.1 What?

Thread local data structures are always a bit confusing, especially when they're used by frameworks. Sorry. So here's a rule of thumb: if you don't *know* whether you're calling code that uses the `get_current_registry()` or `get_current_request()` functions, or you don't care about any of this, but you still want to write test code, just always call `pyramid.testing.setUp()` in your test's `setUp` method and `pyramid.testing.tearDown()` in your tests' `tearDown` method. This won't really hurt anything if the application you're testing does not call any `get_current*` function.

23.2 Using the Configurator and `pyramid.testing` APIs in Unit Tests

The `Configurator` API and the `pyramid.testing` module provide a number of functions which can be used during unit testing. These functions make *configuration declaration* calls to the current *application registry*, but typically register a “stub” or “dummy” feature in place of the “real” feature that the code would call if it was being run normally.

For example, let's imagine you want to unit test a Pyramid view function.

```
1 from pyramid.security import has_permission
2 from pyramid.exceptions import Forbidden
3
4 def view_fn(request):
5     if not has_permission('edit', request.context, request):
6         raise Forbidden
7     return {'greeting': 'hello' }
```

Without doing anything special during a unit test, the call to `has_permission()` in this view function will always return a `True` value. When a Pyramid application starts normally, it will populate a *application registry* using *configuration declaration* calls made against a *Configurator*. But if this application registry is not created and populated (e.g. by initializing the configurator with an authorization policy), like when you invoke application code via a unit test, Pyramid API functions will tend to either fail or return default results. So how do you test the branch of the code in this view function that raises `Forbidden`?

The testing API provided by Pyramid allows you to simulate various application registry registrations for use under a unit testing framework without needing to invoke the actual application configuration implied by its `main` function. For example, if you wanted to test the above `view_fn` (assuming it lived in the package named `my.package`), you could write a `unittest.TestCase` that used the testing API.


```
1 import unittest
2 from pyramid import testing
3
4 class MyTest (unittest.TestCase):
5     def setUp(self):
6         self.config = testing.setUp()
7
8     def tearDown(self):
9         testing.tearDown()
10
11    def test_view_fn_forbidden(self):
12        from pyramid.exceptions import Forbidden
13        from my.package import view_fn
14        self.config.testing_securitypolicy(userid='hank',
15                                           permissive=False)
16
17        request = testing.DummyRequest()
18        request.context = testing.DummyResource()
19        self.assertRaises(Forbidden, view_fn, request)
20
21    def test_view_fn_allowed(self):
22        from pyramid.exceptions import Forbidden
23        from my.package import view_fn
24        self.config.testing_securitypolicy(userid='hank',
25                                           permissive=True)
26
27        request = testing.DummyRequest()
28        request.context = testing.DummyResource()
29        response = view_fn(request)
30        self.assertEqual(response, {'greeting': 'hello'})
```

In the above example, we create a `MyTest` test case that inherits from `unittest.TestCase`. If it's in our Pyramid application, it will be found when `setup.py test` is run. It has two test methods.

The first test method, `test_view_fn_forbidden` tests the `view_fn` when the authentication policy forbids the current user the edit permission. Its third line registers a “dummy” “non-permissive” authorization policy using the `testing_securitypolicy()` method, which is a special helper method for unit testing.

We then create a `pyramid.testing.DummyRequest` object which simulates a `WebOb` request object API. A `pyramid.testing.DummyRequest` is a request object that requires less setup than a “real” Pyramid request. We call the function being tested with the manufactured request. When the function is called, `pyramid.security.has_permission()` will call the “dummy” authentication policy we've registered through `testing_securitypolicy()`, which denies access. We check that the view function raises a `Forbidden` error.

The second test method, named `test_view_fn_allowed` tests the alternate case, where the authentication policy allows access. Notice that we pass different values to `testing_securitypolicy()` to obtain this result. We assert at the end of this that the view function returns a value.

Note that the test calls the `pyramid.testing.setUp()` function in its `setUp` method and the `pyramid.testing.tearDown()` function in its `tearDown` method. We assign the result of `pyramid.testing.setUp()` as `config` on the `unittest` class. This is a *Configurator* object and all methods of the configurator can be called as necessary within tests. If you use any of the *Configurator* APIs during testing, be sure to use this pattern in your test case's `setUp` and `tearDown`; these methods make sure you're using a “fresh” *application registry* per test run.

See the *pyramid.testing* chapter for the entire Pyramid -specific testing API. This chapter describes APIs for registering a security policy, registering resources at paths, registering event listeners, registering views and view permissions, and classes representing “dummy” implementations of a request and a resource.

See also the various methods of the *Configurator* documented in *pyramid.config* that begin with the `testing_` prefix.

23.3 Creating Integration Tests

In Pyramid, a *unit test* typically relies on “mock” or “dummy” implementations to give the code under test only enough context to run.

“Integration testing” implies another sort of testing. In the context of a Pyramid, integration test, the test logic tests the functionality of some code *and* its integration with the rest of the Pyramid framework.

In Pyramid applications that are plugins to Pyramid, you can create an integration test by including it's `includeme` function via `pyramid.config.Configurator.include()` in the test's setup code. This causes the entire Pyramid environment to be set up and torn down as if your application was running “for real”. This is a heavy-hammer way of making sure that your tests have enough context to run properly, and it tests your code's integration with the rest of Pyramid.

Let's demonstrate this by showing an integration test for a view. The below test assumes that your application's package name is `myapp`, and that there is a `views` module in the app with a function with the name `my_view` in it that returns the response ‘Welcome to this application’ after accessing some values that require a fully set up environment.

```

1 import unittest
2
3 from pyramid import testing
4
5 class ViewIntegrationTests(unittest.TestCase):
6     def setUp(self):
7         """ This sets up the application registry with the
8             registrations your application declares in its ``includeme``
9             function.
10            """
11         import myapp
12         self.config = testing.setUp()
13         self.config.include('myapp')
14
15     def tearDown(self):
16         """ Clear out the application registry """
17         testing.tearDown()
18
19     def test_my_view(self):
20         from myapp.views import my_view
21         request = testing.DummyRequest()
22         result = my_view(request)
23         self.assertEqual(result.status, '200 OK')
24         body = result.app_iter[0]
25         self.failUnless('Welcome to' in body)
26         self.assertEqual(len(result.headerlist), 2)
27         self.assertEqual(result.headerlist[0],
28                          ('Content-Type', 'text/html; charset=UTF-8'))
29         self.assertEqual(result.headerlist[1], ('Content-Length',
30                                               str(len(body))))

```

Unless you cannot avoid it, you should prefer writing unit tests that use the Configurator API to set up the right “mock” registrations rather than creating an integration test. Unit tests will run faster (because they do less for each test) and the result of a unit test is usually easier to make assertions about.

23.4 Creating Functional Tests

Functional tests test your literal application.

The below test assumes that your application’s package name is `myapp`, and that there is `view` that returns an HTML body when the root URL is invoked. It further assumes that you’ve added a `tests_require` dependency on the `WebTest` package within your `setup.py` file. `WebTest` is a functional testing package written by Ian Bicking.

```
1 import unittest
2
3 class FunctionalTests(unittest.TestCase):
4     def setUp(self):
5         from myapp import main
6         app = main({})
7         from webtest import TestApp
8         self.testapp = TestApp(app)
9
10    def test_root(self):
11        res = self.testapp.get('/', status=200)
12        self.failUnless('Pyramid' in res.body)
```

When this test is run, each test creates a “real” WSGI application using the `main` function in your `myapp.__init__` module and uses *WebTest* to wrap that WSGI application. It assigns the result to `self.testapp`. In the test named `test_root`, we use the `testapp`’s `get` method to invoke the root URL. We then assert that the returned HTML has the string `Pyramid` in it.

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

USING HOOKS

“Hooks” can be used to influence the behavior of the Pyramid framework in various ways.

24.1 Changing the Not Found View

When Pyramid can’t map a URL to view code, it invokes a *not found view*, which is a *view callable*. A default notfound view exists. The default not found view can be overridden through application configuration.

The *not found view* callable is a view callable like any other. The *view configuration* which causes it to be a “not found” view consists only of naming the `pyramid.exceptions.NotFound` class as the context of the view configuration.

If your application uses *imperative configuration*, you can replace the Not Found view by using the `pyramid.config.Configurator.add_view()` method to register an “exception view”:


```
1 from pyramid.exceptions import NotFound
2 from helloworld.views import notfound_view
3 config.add_view(notfound_view, context=NotFound)
```


Replace `helloworld.views.notfound_view` with a reference to the *view callable* you want to use to represent the Not Found view.

Like any other view, the notfound view must accept at least a `request` parameter, or both `context` and `request`. The `request` is the current *request* representing the denied action. The `context` (if used in the call signature) will be the instance of the `NotFound` exception that caused the view to be called.

Here’s some sample code that implements a minimal `NotFound` view callable:

```
1 from pyramid.httpexceptions import HTTPNotFound
2
3 def notfound_view(request):
4     return HTTPNotFound()
```

 When a NotFound view callable is invoked, it is passed a *request*. The `exception` attribute of the request will be an instance of the `NotFound` exception that caused the not found view to be called. The value of `request.exception.args[0]` will be a value explaining why the not found error was raised. This message will be different when the `debug_notfound` environment setting is true than it is when it is false.

 When a NotFound view callable accepts an argument list as described in *Alternate View Callable Argument/Calling Conventions*, the `context` passed as the first argument to the view callable will be the `NotFound` exception instance. If available, the resource context will still be available as `request.context`.

24.2 Changing the Forbidden View

When Pyramid can't authorize execution of a view based on the *authorization policy* in use, it invokes a *forbidden view*. The default forbidden response has a 403 status code and is very plain, but the view which generates it can be overridden as necessary.

The *forbidden view* callable is a view callable like any other. The *view configuration* which causes it to be a “not found” view consists only of naming the `pyramid.exceptions.Forbidden` class as the `context` of the view configuration.

You can replace the forbidden view by using the `pyramid.config.Configurator.add_view()` method to register an “exception view”:

```
1 from helloworld.views import forbidden_view
2 from pyramid.exceptions import Forbidden
3 config.add_view(forbidden_view, context=Forbidden)
```

Replace `helloworld.views.forbidden_view` with a reference to the Python *view callable* you want to use to represent the Forbidden view.

Like any other view, the forbidden view must accept at least a `request` parameter, or both `context` and `request`. The `context` (available as `request.context` if you're using the request-only view argument pattern) is the context found by the router when the view invocation was denied. The `request` is the current *request* representing the denied action.

Here's some sample code that implements a minimal forbidden view:

```
1 from pyramid.views import view_config
2 from pyramid.response import Response
3
4 def forbidden_view(request):
5     return Response('forbidden')
```



When a forbidden view callable is invoked, it is passed a *request*. The `exception` attribute of the request will be an instance of the `Forbidden` exception that caused the forbidden view to be called. The value of `request.exception.args[0]` will be a value explaining why the forbidden was raised. This message will be different when the `debug_authorization` environment setting is true than it is when it is false.

24.3 Changing the Request Factory

Whenever Pyramid handles a *WSGI* request, it creates a *request* object based on the WSGI environment it has been passed. By default, an instance of the `pyramid.request.Request` class is created to represent the request object.

The class (aka “factory”) that Pyramid uses to create a request object instance can be changed by passing a `request_factory` argument to the constructor of the *configurator*. This argument can be either a callable or a *dotted Python name* representing a callable.

```
1 from pyramid.request import Request
2
3 class MyRequest(Request):
4     pass
5
6 config = Configurator(request_factory=MyRequest)
```

If you're doing imperative configuration, and you'd rather do it after you've already constructed a *configurator* it can also be registered via the `pyramid.config.Configurator.set_request_factory()` method:

```
1 from pyramid.config import Configurator
2 from pyramid.request import Request
3
4 class MyRequest(Request):
5     pass
6
7 config = Configurator()
8 config.set_request_factory(MyRequest)
```

24.4 Adding Renderer Globals

Whenever Pyramid handles a request to perform a rendering (after a view with a `renderer=` configuration attribute is invoked, or when the any of the methods beginning with `render` within the `pyramid.renderers` module are called), *renderer globals* can be injected into the *system* values sent to the renderer. By default, no renderer globals are injected, and the “bare” system values (such as `request`, `context`, and `renderer_name`) are the only values present in the system dictionary passed to every renderer.

A callback that Pyramid will call every time a renderer is invoked can be added by passing a `renderer_globals_factory` argument to the constructor of the *configurator*. This callback can either be a callable object or a *dotted Python name* representing such a callable.

```
1 def renderer_globals_factory(system):
2     return {'a':1}
3
4 config = Configurator(
5     renderer_globals_factory=renderer_globals_factory)
```

Such a callback must accept a single positional argument (notionally named `system`) which will contain the original system values. It must return a dictionary of values that will be merged into the system dictionary. See *System Values Used During Rendering* for discription of the values present in the system dictionary.

If you're doing imperative configuration, and you'd rather do it after you've already constructed a *configurator* it can also be registered via the `pyramid.config.Configurator.set_renderer_globals_factory()` method:


```

1 from pyramid.config import Configurator
2
3 def renderer_globals_factory(system):
4     return {'a':1}
5
6 config = Configurator()
7 config.set_renderer_globals_factory(renderer_globals_factory)

```

Another mechanism which allows event subscribers to add renderer global values exists in *Using The Before Render Event*.

24.5 Using The Before Render Event

Subscribers to the `pyramid.events.BeforeRender` event may introspect the and modify the set of *renderer globals* before they are passed to a *renderer*. This event object itself has a dictionary-like interface that can be used for this purpose. For example:

```

1 from pyramid.events import subscriber
2 from pyramid.events import BeforeRender
3
4 @subscriber(BeforeRender)
5 def add_global(event):
6     event['mykey'] = 'foo'

```

An object of this type is sent as an event just before a *renderer* is invoked (but *after* the application-level renderer globals factory added via `set_renderer_globals_factory`, if any, has injected its own keys into the renderer globals dictionary).

If a subscriber attempts to add a key that already exist in the renderer globals dictionary, a `KeyError` is raised. This limitation is enforced because event subscribers do not possess any relative ordering. The set of keys added to the renderer globals dictionary by all `pyramid.events.BeforeRender` subscribers and renderer globals factories must be unique.

See the API documentation for the `BeforeRender` event interface at `pyramid.interfaces.IBeforeRender`.

Another mechanism which allows event subscribers more control when adding renderer global values exists in *Adding Renderer Globals*.

24.6 Using Response Callbacks

Unlike many other web frameworks, Pyramid does not eagerly create a global response object. Adding a *response callback* allows an application to register an action to be performed against a response object once it is created, usually in order to mutate it.

The `pyramid.request.Request.add_response_callback()` method is used to register a response callback.

A response callback is a callable which accepts two positional parameters: `request` and `response`. For example:

```
1 def cache_callback(request, response):
2     """Set the cache_control max_age for the response"""
3     if request.exception is not None:
4         response.cache_control.max_age = 360
5     request.add_response_callback(cache_callback)
```

No response callback is called if an unhandled exception happens in application code, or if the response object returned by a *view callable* is invalid. Response callbacks *are*, however, invoked when a *exception view* is rendered successfully: in such a case, the `request.exception` attribute of the request when it enters a response callback will be an exception object instead of its default value of `None`.

Response callbacks are called in the order they're added (first-to-most-recently-added). All response callbacks are called *after* the `NewResponse` event is sent. Errors raised by response callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

A response callback has a lifetime of a *single* request. If you want a response callback to happen as the result of *every* request, you must re-register the callback into every new request (perhaps within a subscriber of a `NewRequest` event).

24.7 Using Finished Callbacks

A *finished callback* is a function that will be called unconditionally by the Pyramid *router* at the very end of request processing. A finished callback can be used to perform an action at the end of a request unconditionally.

The `pyramid.request.Request.add_finished_callback()` method is used to register a finished callback.

A finished callback is a callable which accepts a single positional parameter: `request`. For example:

```
1 import transaction
2
3 def commit_callback(request):
4     '''commit or abort the transaction associated with request'''
5     if request.exception is not None:
6         transaction.abort()
7     else:
8         transaction.commit()
9 request.add_finished_callback(commit_callback)
```

Finished callbacks are called in the order they're added (first-to-most-recently-added). Finished callbacks (unlike a *response callback*) are *always* called, even if an exception happens in application code that prevents a response from being generated.

The set of finished callbacks associated with a request are called *very late* in the processing of that request; they are essentially the very last thing called by the *router* before a request “ends”. They are called after response processing has already occurred in a top-level `finally:` block within the router request processing code. As a result, mutations performed to the `request` provided to a finished callback will have no meaningful effect, because response processing will have already occurred, and the request's scope will expire almost immediately after all finished callbacks have been processed.

It is often necessary to tell whether an exception occurred within *view callable* code from within a finished callback: in such a case, the `request.exception` attribute of the request when it enters a response callback will be an exception object instead of its default value of `None`.

Errors raised by finished callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

A finished callback has a lifetime of a *single* request. If you want a finished callback to happen as the result of *every* request, you must re-register the callback into every new request (perhaps within a subscriber of a `NewRequest` event).

24.8 Changing the Traverser

The default *traversal* algorithm that Pyramid uses is explained in *The Traversal Algorithm*. Though it is rarely necessary, this default algorithm can be swapped out selectively for a different traversal pattern via configuration.

24. USING HOOKS

```
1 from pyramid.interfaces import ITraverser
2 from zope.interface import Interface
3 from myapp.traversal import Traverser
4
5 config.registry.registerAdapter(Traverser, (Interface,), ITraverser)
```

In the example above, `myapp.traversal.Traverser` is assumed to be a class that implements the following interface:

```
1 class Traverser(object):
2     def __init__(self, root):
3         """ Accept the root object returned from the root factory """
4
5     def __call__(self, request):
6         """ Return a dictionary with (at least) the keys ``root``,
7             ``context``, ``view_name``, ``subpath``, ``traversed``,
8             ``virtual_root``, and ``virtual_root_path``. These values are
9             typically the result of a resource tree traversal. ``root``
10            is the physical root object, ``context`` will be a resource
11            object, ``view_name`` will be the view name used (a Unicode
12            name), ``subpath`` will be a sequence of Unicode names that
13            followed the view name but were not traversed, ``traversed``
14            will be a sequence of Unicode names that were traversed
15            (including the virtual root path, if any) ``virtual_root``
16            will be a resource object representing the virtual root (or the
17            physical root if traversal was not performed), and
18            ``virtual_root_path`` will be a sequence representing the
19            virtual root path (a sequence of Unicode names) or None if
20            traversal was not performed.
21
22            Extra keys for special purpose functionality can be added as
23            necessary.
24
25            All values returned in the dictionary will be made available
26            as attributes of the ``request`` object.
27            """
```

More than one traversal algorithm can be active at the same time. For instance, if your *root factory* returns more than one type of object conditionally, you could claim that an alternate traverser adapter is for only one particular class or interface. When the root factory returned an object that implemented that class or interface, a custom traverser would be used. Otherwise, the default traverser would be used. For example:

```
1 from pyramid.interfaces import ITraverser
2 from zope.interface import Interface
```

```
3 from myapp.traversal import Traverser
4 from myapp.resources import MyRoot
5
6 config.registry.registerAdapter(Traverser, (MyRoot,), ITraverser)
```

If the above stanza was added to a Pyramid `__init__.py` file's main function, Pyramid would use the `myapp.traversal.Traverser` only when the application *root factory* returned an instance of the `myapp.resources.MyRoot` object. Otherwise it would use the default Pyramid traverser to do traversal.

24.9 Changing How `pyramid.url.resource_url` Generates a URL

When you add a traverser as described in *Changing the Traverser*, it's often convenient to continue to use the `pyramid.url.resource_url()` API. However, since the way traversal is done will have been modified, the URLs it generates by default may be incorrect.

If you've added a traverser, you can change how `resource_url()` generates a URL for a specific type of resource by adding a `registerAdapter` call for `pyramid.interfaces.IContextURL` to your application:

```
1 from pyramid.interfaces import ITraverser
2 from zope.interface import Interface
3 from myapp.traversal import URLGenerator
4 from myapp.resources import MyRoot
5
6 config.registry.registerAdapter(URLGenerator, (MyRoot, Interface),
7                                 IContextURL)
```

In the above example, the `myapp.traversal.URLGenerator` class will be used to provide services to `resource_url()` any time the *context* passed to `resource_url` is of class `myapp.resources.MyRoot`. The second argument in the `(MyRoot, Interface)` tuple represents the type of interface that must be possessed by the *request* (in this case, any interface, represented by `zope.interface.Interface`).

The API that must be implemented by a class that provides `IContextURL` is as follows:

```
1 from zope.interface import Interface
2
3 class IContextURL(Interface):
4     """ An adapter which deals with URLs related to a context.
5     """
6     def __init__(self, context, request):
7         """ Accept the context and request """
8
9     def virtual_root(self):
10        """ Return the virtual root object related to a request and the
11        current context """
12
13    def __call__(self):
14        """ Return a URL that points to the context """
```

The default context URL generator is available for perusal as the class `pyramid.traversal.TraversalContextURL` in the `traversal` module of the *Pylons* GitHub Pyramid repository.

24.10 Using a View Mapper

The default calling conventions for view callables are documented in the *Views* chapter. You can change the way users define view callables by employing a *view mapper*.

A view mapper is an object that accepts a set of keyword arguments and which returns a callable. The returned callable is called with the *view callable* object. The returned callable should itself return another callable which can be called with the “internal calling protocol” (`context, request`).

You can use a view mapper in a number of ways:

- by setting a `__view_mapper__` attribute (which is the view mapper object) on the view callable itself
- by passing the mapper object to `pyramid.config.Configurator.add_view()` (or its declarative/decorator equivalents) as the `mapper` argument.
- by registering a *default* view mapper.

Here's an example of a view mapper that emulates (somewhat) a Pylons "controller". The mapper is initialized with some keyword arguments. Its `__call__` method accepts the view object (which will be a class). It uses the `attr` keyword argument it is passed to determine which attribute should be used as an action method. The wrapper method it returns accepts (`context`, `request`) and returns the result of calling the action method with keyword arguments implied by the `matchdict` after popping the `action` out of it. This somewhat emulates the Pylons style of calling action methods with routing parameters pulled out of the route matching dict as keyword arguments.

```

1  # framework
2
3  class PylonsControllerViewMapper(object):
4      def __init__(self, **kw):
5          self.kw = kw
6
7      def __call__(self, view):
8          attr = self.kw['attr']
9          def wrapper(context, request):
10             matchdict = request.matchdict.copy()
11             matchdict.pop('action', None)
12             inst = view()
13             meth = getattr(inst, attr)
14             return meth(**matchdict)
15         return wrapper
16
17  class BaseController(object):
18      __view_mapper__ = PylonsControllerViewMapper

```

A user might make use of these framework components like so:

```

1  # user application
2
3  from webob import Response
4  from pyramid.config import Configurator
5  import pyramid_handlers
6  from paste.httpserver import serve
7
8  class MyController(BaseController):
9      def index(self, id):
10         return Response(id)
11
12  if __name__ == '__main__':
13     config = Configurator()
14     config.include(pyramid_handlers)
15     config.add_handler('one', '/{id}', MyController, action='index')
16     config.add_handler('two', '/{action}/{id}', MyController)

```

```
17 | serve(config.make_wsgi_app())
```

The `pyramid.config.Configurator.set_default_mapper()` method can be used to set a *default* view mapper (overriding the superdefault view mapper used by Pyramid itself).

A *single* view registration can use a view mapper by passing the mapper as the `mapper` argument to `add_view()`.

24.11 Registering Configuration Decorators

Decorators such as `view_config` don't change the behavior of the functions or classes they're decorating. Instead, when a *scan* is performed, a modified version of the function or class is registered with Pyramid.

You may wish to have your own decorators that offer such behaviour. This is possible by using the *Venusian* package in the same way that it is used by Pyramid.

By way of example, let's suppose you want to write a decorator that registers the function it wraps with a *Zope Component Architecture* "utility" within the *application registry* provided by Pyramid. The application registry and the utility inside the registry is likely only to be available once your application's configuration is at least partially completed. A normal decorator would fail as it would be executed before the configuration had even begun.

However, using *Venusian*, the decorator could be written as follows:

```
1  import venusian
2  from pyramid.threadlocal import get_current_registry
3  from mypackage.interfaces import IMyUtility
4
5  class registerFunction(object):
6
7      def __init__(self, path):
8          self.path = path
9
10     def register(self, scanner, name, wrapped):
11         registry = scanner.config.registry
12         registry.getUtility(IMyUtility).register(
13             self.path, wrapped
14         )
15
16     def __call__(self, wrapped):
17         venusian.attach(wrapped, self.register)
18         return wrapped
```


This decorator could then be used to register functions throughout your code:

```
1 @registerFunction('/some/path')
2 def my_function():
3     do_stuff()
```

However, the utility would only be looked up when a *scan* was performed, enabling you to set up the utility in advance:

```
1 from paste.httpserver import serve
2 from pyramid.config import Configurator
3
4 class UtilityImplementation:
5
6     implements(ISomething)
7
8     def __init__(self):
9         self.registrations = {}
10
11     def register(self, path, callable_):
12         self.registrations[path]=callable_
13
14 if __name__ == '__main__':
15     config = Configurator()
16     config.registry.registerUtility(UtilityImplementation())
17     config.scan()
18     app = config.make_wsgi_app()
19     serve(app, host='0.0.0.0')
```

For full details, please read the Venusian documentation.

ADVANCED CONFIGURATION

To support application extensibility, the Pyramid *Configurator*, by default, detects configuration conflicts and allows you to include configuration imperatively from other packages or modules. It also, by default, performs configuration in two separate phases. This allows you to ignore relative configuration statement ordering in some circumstances.

25.1 Conflict Detection

Here's a familiar example of one of the simplest Pyramid applications, configured imperatively:

```
1 from paste.httpserver import serve
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 if __name__ == '__main__':
9     config = Configurator()
10    config.add_view(hello_world)
11    app = config.make_wsgi_app()
12    serve(app, host='0.0.0.0')
```

When you start this application, all will be OK. However, what happens if we try to add another view to the configuration with the same set of *predicate* arguments as one we've already added?

25. ADVANCED CONFIGURATION

```
1 from paste.httpserver import serve
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 def goodbye_world(request):
9     return Response('Goodbye world!')
10
11 if __name__ == '__main__':
12     config = Configurator()
13
14     config.add_view(hello_world, name='hello')
15
16     # conflicting view configuration
17     config.add_view(goodbye_world, name='hello')
18
19     app = config.make_wsgi_app()
20     serve(app, host='0.0.0.0')
```

The application now has two conflicting view configuration statements. When we try to start it again, it won't start. Instead, we'll receive a traceback that ends something like this:

```
1 Traceback (most recent call last):
2   File "app.py", line 12, in <module>
3     app = config.make_wsgi_app()
4   File "pyramid/config.py", line 839, in make_wsgi_app
5     self.commit()
6   File "pyramid/pyramid/config.py", line 473, in commit
7     self._ctx.execute_actions()
8   File "zope/configuration/config.py", line 600, in execute_actions
9     for action in resolveConflicts(self.actions):
10    File "zope/configuration/config.py", line 1507, in resolveConflicts
11        raise ConfigurationConflictError(conflicts)
12 zope.configuration.config.ConfigurationConflictError:
13     Conflicting configuration actions
14     For: ('view', None, '', None, <InterfaceClass pyramid.interfaces.IView>,
15          None, None, None, None, None, False, None, None, None)
16     ('app.py', 14, '<module>', 'config.add_view(hello_world)')
17     ('app.py', 17, '<module>', 'config.add_view(hello_world)')
```

This traceback is trying to tell us:

- We've got conflicting information for a set of view configuration statements (The `FOR:` line).

- There are two statements which conflict, shown beneath the `For:` line:
`config.add_view(hello_world, 'hello')` on line 14 of `app.py`, and
`config.add_view(goodbye_world, 'hello')` on line 17 of `app.py`.

These two configuration statements are in conflict because we've tried to tell the system that the set of *predicate* values for both view configurations are exactly the same. Both the `hello_world` and `goodbye_world` views are configured to respond under the same set of circumstances. This circumstance: the *view name* (represented by the `name= predicate`) is `hello`.

This presents an ambiguity that Pyramid cannot resolve. Rather than allowing the circumstance to go unreported, by default Pyramid raises a `ConfigurationConflictError` error and prevents the application from running.

Conflict detection happens for any kind of configuration: imperative configuration or configuration that results from the execution of a *scan*.

25.1.1 Manually Resolving Conflicts

There are a number of ways to manually resolve conflicts: the “right” way, by strategically using `pyramid.config.Configurator.commit()`, or by using an “autocommitting” configurator.

The Right Thing

The most correct way to resolve conflicts is to “do the needful”: change your configuration code to not have conflicting configuration statements. The details of how this is done depends entirely on the configuration statements made by your application. Use the detail provided in the `ConfigurationConflictError` to track down the offending conflicts and modify your configuration code accordingly.

If you're getting a conflict while trying to extend an existing application, and that application has a function which performs configuration like this one:

```
1 def add_routes(config):  
2     config.add_route(...)
```

Don't call this function directly with `config` as an argument. Instead, use `pyramid.config.Configuration.include()`:

```
1 config.include(add_routes)
```

Using `include()` instead of calling the function directly provides a modicum of automated conflict resolution, with the configuration statements you define in the calling code overriding those of the included function. See also *Automatic Conflict Resolution* and *Including Configuration from External Sources*.

Using `config.commit()`

You can manually commit a configuration by using the `commit()` method between configuration calls. For example, we prevent conflicts from occurring in the application we examined previously as the result of adding a `commit`. Here's the application that generates conflicts:

```
1 from paste.httpserver import serve
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
7
8 def goodbye_world(request):
9     return Response('Goodbye world!')
10
11 if __name__ == '__main__':
12     config = Configurator()
13
14     config.add_view(hello_world, name='hello')
15
16     # conflicting view configuration
17     config.add_view(goodbye_world, name='hello')
18
19     app = config.make_wsgi_app()
20     serve(app, host='0.0.0.0')
```

We can prevent the two `add_view` calls from conflicting by issuing a call to `commit()` between them:

```
1 from paste.httpserver import serve
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 def hello_world(request):
6     return Response('Hello world!')
```

```

7
8 def goodbye_world(request):
9     return Response('Goodbye world!')
10
11 if __name__ == '__main__':
12     config = Configurator()
13
14     config.add_view(hello_world, name='hello')
15
16     config.commit() # commit any pending configuration actions
17
18     # no-longer-conflicting view configuration
19     config.add_view(goodbye_world, name='hello')
20
21     app = config.make_wsgi_app()
22     serve(app, host='0.0.0.0')
```

In the above example we've issued a call to `commit()` between the two `add_view` calls. `commit()` will cause any pending configuration statements.

Calling `commit()` is safe at any time. It executes all pending configuration actions and leaves the configuration action list "clean".

Note that `commit()` has no effect when you're using an *autocommitting* configurator (see *Using An Autocommitting Configurator*).

Using An Autocommitting Configurator

You can also use a heavy hammer to circumvent conflict detection by using a configurator constructor parameter: `autocommit=True`. For example:

```

1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     config = Configurator(autocommit=True)
```

When the `autocommit` parameter passed to the `Configurator` is `True`, conflict detection (and *Two-Phase Configuration*) is disabled. Configuration statements will be executed immediately, and succeeding statements will override preceding ones.

`commit()` has no effect when `autocommit` is `True`.

If you use a `Configurator` in code that performs unit testing, it's usually a good idea to use an *autocommitting* `Configurator`, because you are usually unconcerned about conflict detection or two-phase configuration in test code.

25.1.2 Automatic Conflict Resolution

If your code uses the `include()` method to include external configuration, some conflicts are automatically resolved. Configuration statements that are made as the result of an “include” will be overridden by configuration statements that happen within the caller of the “include” method.

Automatic conflict resolution supports this goal: if a user wants to reuse a Pyramid application, and they want to customize the configuration of this application without hacking its code “from outside”, they can “include” a configuration function from the package and override only some of its configuration statements within the code that does the include. No conflicts will be generated by configuration statements within the code which does the including, even if configuration statements in the included code would conflict if it was moved “up” to the calling code.

25.1.3 Methods Which Provide Conflict Detection

These are the methods of the configurator which provide conflict detection:

```
add_view(),    add_route(),    add_renderer(),    set_request_factory(),
set_renderer_globals_factory()    set_locale_negotiator()    and
set_default_permission().
```

Some other methods of the configurator also indirectly provide conflict detection, because they’re implemented in terms of conflict-aware methods:

- `add_route()` does a second type of conflict detection when a `view` parameter is passed (it calls `add_view`).
- `static_view()`, a frontend for `add_route` and `add_view`.

25.2 Including Configuration from External Sources

Some application programmers will factor their configuration code in such a way that it is easy to reuse and override configuration statements. For example, such a developer might factor out a function used to add routes to his application:

```
1 def add_routes(config):
2     config.add_route(...)
```

Rather than calling this function directly with `config` as an argument. Instead, use `pyramid.config.Configuration.include()`:


```
1 config.include(add_routes)
```

Using `include` rather than calling the function directly will allow *Automatic Conflict Resolution* to work.

`include()` can also accept a *module* as an argument:

```
1 import myapp
2
3 config.include(myapp)
```

For this to work properly, the `myapp` module must contain a callable with the special name `includeme`, which should perform configuration (like the `add_routes` callable we showed above as an example).

`include()` can also accept a *dotted Python name* to a function or a module.

25.3 Two-Phase Configuration

When a non-autocommitting *Configurator* is used to do configuration (the default), configuration execution happens in two phases. In the first phase, “eager” configuration actions (actions that must happen before all others, such as registering a renderer) are executed, and *discriminators* are computed for each of the actions that depend on the result of the eager actions. In the second phase, the discriminators of all actions are compared to do conflict detection.

Due to this, for configuration methods that have no internal ordering constraints, execution order of configuration method calls is not important. For example, the relative ordering of `add_view()` and `add_renderer()` is unimportant when a non-autocommitting configurator is used. This code snippet:

```
1 config.add_view('some.view', renderer='path_to_custom/renderer.rn')
2 config.add_renderer('.rn', SomeCustomRendererFactory)
```

Has the same result as:

```
1 config.add_renderer('.rn', SomeCustomRendererFactory)
2 config.add_view('some.view', renderer='path_to_custom/renderer.rn')
```

Even though the view statement depends on the registration of a custom renderer, due to two-phase configuration, the order in which the configuration statements are issued is not important. `add_view` will be able to find the `.rn` renderer even if `add_renderer` is called after `add_view`.

The same is untrue when you use an *autocommitting* configurator (see *Using An Autocommitting Configurator*). When an autocommitting configurator is used, two-phase configuration is disabled, and configuration statements must be ordered in dependency order.

Some configuration methods, such as `add_route()` have internal ordering constraints: the routes they imply require relative ordering. Such ordering constraints are not absolved by two-phase configuration. Routes are still added in configuration execution order.

25.4 Adding Methods to the Configurator via `add_directive`

Framework extension writers can add arbitrary methods to a *Configurator* by using the `pyramid.config.Configurator.add_directive()` method of the configurator. This makes it possible to extend a Pyramid configurator in arbitrary ways, and allows it to perform application-specific tasks more succinctly.

The `add_directive()` method accepts two positional arguments: a method name and a callable object. The callable object is usually a function that takes the configurator instance as its first argument and accepts other arbitrary positional and keyword arguments. For example:

```
from pyramid.events import NewRequest
from pyramid.config import Configurator

def add_newrequest_subscriber(config, subscriber):
    config.add_subscriber(subscriber, NewRequest).

if __name__ == '__main__':
    config = Configurator()
    config.add_directive('add_newrequest_subscriber',
                        add_newrequest_subscriber)
```

Once `add_directive()` is called, a user can then call the method by its given name as if it were a built-in method of the *Configurator*:

```
1 def mysubscriber(event):
2     print event.request
3
4 config.add_newrequest_subscriber(mysubscriber)
```

A call to `add_directive()` is often “hidden” within an `includeme` function within a “frameworky” package meant to be included as per *Including Configuration from External Sources* via `include()`. For example, if you put this code in a package named `pyramid_subscriberhelpers`:

```
def includeme(config)
    config.add_directive('add_newrequest_subscriber',
                        add_newrequest_subscriber)
```

The user of the add-on package `pyramid_subscriberhelpers` would then be able to install it and subsequently do:

```
1 def mysubscriber(event):
2     print event.request
3
4 from pyramid.config import Configurator
5 config = Configurator()
6 config.include('pyramid_subscriberhelpers')
7 config.add_newrequest_subscriber(mysubscriber)
```


EXTENDING AN EXISTING PYRAMID APPLICATION

If a Pyramid developer has obeyed certain constraints while building an application, a third party should be able to change the application’s behavior without needing to modify its source code. The behavior of a Pyramid application that obeys certain constraints can be *overridden* or *extended* without modification.

We’ll define some jargon here for the benefit of identifying the parties involved in such an effort.

Developer The original application developer.

Integrator Another developer who wishes to reuse the application written by the original application developer in an unanticipated context. He may also wish to modify the original application without changing the original application’s source code.

26.1 The Difference Between “Extensible” and “Pluggable” Applications

Other web frameworks, such as *Django*, advertise that they allow developers to create “pluggable applications”. They claim that if you create an application in a certain way, it will be integratable in a sensible, structured way into another arbitrarily-written application or project created by a third-party developer.

Pyramid, as a platform, does not claim to provide such a feature. The platform provides no guarantee that you can create an application and package it up such that an arbitrary integrator can use it as a subcomponent in a larger Pyramid application or project. Pyramid does not mandate the constraints necessary

for such a pattern to work satisfactorily. Because Pyramid is not very “opinionated”, developers are able to use wildly different patterns and technologies to build an application. A given Pyramid application may happen to be reusable by a particular third party integrator, because the integrator and the original developer may share similar base technology choices (such as the use of a particular relational database or ORM). But the same application may not be reusable by a different developer, because he has made different technology choices which are incompatible with the original developer’s.

As a result, the concept of a “pluggable application” is left to layers built above Pyramid, such as a “CMS” layer or “application server” layer. Such layers are apt to provide the necessary “opinions” (such as mandating a storage layer, a templating system, and a structured, well-documented pattern of registering that certain URLs map to certain bits of code) which makes the concept of a “pluggable application” possible. “Pluggable applications”, thus, should not plug in to Pyramid itself but should instead plug into a system written atop Pyramid.

Although it does not provide for “pluggable applications”, Pyramid *does* provide a rich set of mechanisms which allows for the extension of a single existing application. Such features can be used by frameworks built using Pyramid as a base. All Pyramid applications may not be *pluggable*, but all Pyramid applications are *extensible*.

26.2 Rules for Building An Extensible Application

There is only one rule you need to obey if you want to build a maximally extensible Pyramid application: as a developer, you should factor any overrideable *imperative configuration* you’ve created into functions which can be used via `pyramid.config.Configurator.include()` rather than inlined as calls to methods of a *Configurator* within the main function in your application’s `__init__.py`. For example, rather than:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     config = Configurator()
5     config.add_view('myapp.views.view1', name='view1')
6     config.add_view('myapp.views.view2', name='view2')
```

You should do move the calls to `add_view` outside of the (non-reusable) `if __name__ == '__main__'` block, and into a reusable function:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
```

```
4     config = Configurator()
5     config.include(add_views)
6
7     def add_views(config):
8         config.add_view('myapp.views.view1', name='view1')
9         config.add_view('myapp.views.view2', name='view2')
```

Doing this allows an integrator to maximally reuse the configuration statements that relate to your application by allowing him to selectively include or disinclude the configuration functions you've created from an “override package”.

Alternately, you can use *ZCML* for the purpose of making configuration extensible and overrideable. *ZCML* declarations that belong to an application can be overridden and extended by integrators as necessary in a similar fashion. If you use only *ZCML* to configure your application, it will automatically be maximally extensible without any manual effort. See *pyramid_zcml* for information about using *ZCML*.

26.2.1 Fundamental Plugpoints

The fundamental “plug points” of an application developed using Pyramid are *routes*, *views*, and *assets*. Routes are declarations made using the `pyramid.config.Configurator.add_route()` method. Views are declarations made using the `pyramid.config.Configurator.add_view()` method. Assets are files that are accessed by Pyramid using the *pkg_resources* API such as static files and templates via a *asset specification*. Other directives and configurator methods also deal in routes, views, and assets. For example, `add_handler` directive of the `pyramid_handlers` package adds a single route, and some number of views.

26.3 Extending an Existing Application

The steps for extending an existing application depend largely on whether the application does or does not use configuration decorators and/or imperative code.

26.3.1 If The Application Has Configuration Decorations

You've inherited a Pyramid application which you'd like to extend or override that uses `pyramid.view.view_config` decorators or other *configuration decoration* decorators.

If you just want to *extend* the application, you can run a *scan* against the application's package, then add additional configuration that registers more views or routes.

```
1 if __name__ == '__main__':
2     config.scan('someotherpackage')
3     config.add_view('mypackage.views.myview', name='myview')
```

If you want to *override* configuration in the application, you *may* need to run `pyramid.config.Configurator.commit()` after performing the scan of the original package, then add additional configuration that registers more views or routes which performs overrides.

```
if __name__ == '__main__':
    config.scan('someotherpackage')
    config.commit()
    config.add_view('mypackage.views.myview', name='myview')
```

Once this is done, you should be able to extend or override the application like any other (see *Extending the Application*).

You can alternately just prevent a *scan* from happening (by omitting any call to the `pyramid.config.Configurator.scan()` method). This will cause the decorators attached to objects in the target application to do nothing. At this point, you will need to convert all the configuration done in decorators into equivalent imperative configuration or ZCML and add that configuration or ZCML to a separate Python package as described in *Extending the Application*.

26.3.2 Extending the Application

To extend or override the behavior of an existing application, you will need to create a new package which includes the configuration of the old package, and you'll perhaps need to create implementations of the types of things you'd like to override (such as views), which are referred to within the original package.

The general pattern for extending an existing application looks something like this:

- Create a new Python package. The easiest way to do this is to create a new Pyramid application using the “paster” template mechanism. See *Creating the Project* for more information.
- In the new package, create Python files containing views and other overridden elements, such as templates and static assets as necessary.
- Install the new package into the same Python environment as the original application (e.g. `python setup.py develop` or `python setup.py install`).

- Change the main function in the new package’s `__init__.py` to include the original Pyramid application’s configuration functions via `pyramid.config.Configurator.include()` statements or a *scan*.
- Wire the new views and assets created in the new package up using imperative registrations within the main function of the `__init__.py` file of the new application. These wiring should happen *after* including the configuration functions of the old application. These registrations will extend or override any registrations performed by the original application. See *Overriding Views*, *Overriding Routes* and *Overriding Assets*.

26.3.3 Overriding Views

The *view configuration* declarations you make which *override* application behavior will usually have the same *view predicate* attributes as the original you wish to override. These `<view>` declarations will point at “new” view code, in the override package you’ve created. The new view code itself will usually be cut-n-paste copies of view callables from the original application with slight tweaks.

For example, if the original application has the following `configure_views` configuration method:

```
1 def configure_views(config):
2     config.add_view('theoriginalapp.views.theview', name='theview')
```

You can override the first view configuration statement made by `configure_views` within the override package, after loading the original configuration function:

```
1 from pyramid.config import Configurator
2 from originalapp import configure_views
3
4 if __name__ == '__main__':
5     config = Configurator()
6     config.include(configure_views)
7     config.add_view('theoverrideapp.views.theview', name='theview')
```

In this case, the `theoriginalapp.views.theview` view will never be executed. Instead, a new view, `theoverrideapp.views.theview` will be executed instead, when request circumstances dictate.

A similar pattern can be used to *extend* the application with `add_view` declarations. Just register a new view against some other set of predicates to make sure the URLs it implies are available on some other page rendering.

26.3.4 Overriding Routes

Route setup is currently typically performed in a sequence of ordered calls to `add_route()`. Because these calls are ordered relative to each other, and because this ordering is typically important, you should retain their relative ordering when performing an override. Typically, this means *copying* all the `add_route` statements into the override package's file and changing them as necessary. Then disinclude any `add_route` statements from the original application.

26.3.5 Overriding Assets

Assets are files on the filesystem that are accessible within a Python *package*. An entire chapter is devoted to assets: *Static Assets*. Within this chapter is a section named *Overriding Assets*. This section of that chapter describes in detail how to override package assets with other assets by using the `pyramid.config.Configurator.override_asset()` method. Add such `override_asset` calls to your override package's `__init__.py` to perform overrides.

STARTUP

When you cause a Pyramid application to start up in a console window, you'll see something much like this show up on the console:

```
$ paster serve myproject/MyProject.ini
Starting server in PID 16601.
serving on 0.0.0.0:6543 view at http://127.0.0.1:6543
```

This chapter explains what happens between the time you press the “Return” key on your keyboard after typing `paster serve myproject/MyProject.ini` and the time the line `serving on 0.0.0.0:6543 ...` is output to your console.

27.1 The Startup Process

The easiest and best-documented way to start and serve a Pyramid application is to use the `paster serve` command against a *PasteDeploy* `.ini` file. This uses the `.ini` file to infer settings and starts a server listening on a port. For the purposes of this discussion, we'll assume that you are using this command to run your Pyramid application.

Here's a high-level time-ordered overview of what happens when you press `return` after running `paster serve development.ini`.

1. The *PasteDeploy* `paster` command is invoked under your shell with the arguments `serve` and `development.ini`. As a result, the *PasteDeploy* framework recognizes that it is meant to begin to run and serve an application using the information contained within the `development.ini` file.

2. The PasteDeploy framework finds a section named either `[app:main]`, `[pipeline:main]`, or `[composite:main]` in the `.ini` file. This section represents the configuration of a *WSGI* application that will be served. If you're using a simple application (e.g. `[app:main]`), the application *entry point* or *dotted Python name* will be named on the `use=` line within the section's configuration. If, instead of a simple application, you're using a *WSGI pipeline* (e.g. a `[pipeline:main]` section), the application named on the "last" element will refer to your Pyramid application. If instead of a simple application or a pipeline, you're using a Paste "composite" (e.g. `[composite:main]`), refer to the documentation for that particular composite to understand how to make it refer to your Pyramid application.
3. The application's *constructor* (named by the entry point reference or dotted Python name on the `use=` line of the section representing your Pyramid application) is passed the key/value parameters mentioned within the section in which it's defined. The constructor is meant to return a *router* instance, which is a *WSGI* application.

For Pyramid applications, the constructor will be a function named `main` in the `__init__.py` file within the *package* in which your application lives. If this function succeeds, it will return a Pyramid *router* instance. Here's the contents of an example `__init__.py` module:

```
1 from pyramid.config import Configurator
2 from myproject.resources import Root
3
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6         """
7     config = Configurator(root_factory=Root, settings=settings)
8     config.add_view('myproject.views.my_view',
9                    context='myproject.resources.Root',
10                   renderer='myproject:templates/mytemplate.pt')
11     config.add_static_view('static', 'myproject:static')
12     return config.make_wsgi_app()
```

Note that the constructor function accepts a `global_config` argument, which is a dictionary of key/value pairs mentioned in the `[DEFAULT]` section of an `.ini` file. It also accepts a `**settings` argument, which collects another set of arbitrary key/value pairs. The arbitrary key/value pairs received by this function in `**settings` will be composed of all the key/value pairs that are present in the `[app:MyProject]` section (except for the `use=` setting) when this function is called by the *PasteDeploy* framework when you run `paster serve`.

Our generated `development.ini` file looks like so:

```
1 [app:MyProject]
2 use = egg:MyProject
```

```
3 reload_templates = true
4 debug_authorization = false
5 debug_notfound = false
6 debug_routematch = false
7 debug_templates = true
8 default_locale_name = en
9
10 [pipeline:main]
11 pipeline =
12     egg:WebError#evalerror
13     MyProject
14
15 [server:main]
16 use = egg:Paste#http
17 host = 0.0.0.0
18 port = 6543
19
20 # Begin logging configuration
21
22 [loggers]
23 keys = root, myproject
24
25 [handlers]
26 keys = console
27
28 [formatters]
29 keys = generic
30
31 [logger_root]
32 level = INFO
33 handlers = console
34
35 [logger_myproject]
36 level = DEBUG
37 handlers =
38 qualname = myproject
39
40 [handler_console]
41 class = StreamHandler
42 args = (sys.stderr,)
43 level = NOTSET
44 formatter = generic
45
46 [formatter_generic]
47 format = %(asctime)s %(levelname)-5.5s [% (name)s] %(message)s
48
```

```
49 | # End logging configuration
```

In this case, the `myproject.__init__:main` function referred to by the entry point URI `egg:MyProject` (see *development.ini* for more information about entry point URIs, and how they relate to callables), will receive the key/value pairs `{'reload_templates':'true', 'debug_authorization':'false', 'debug_notfound':'false', 'debug_routematch':'false', 'debug_templates':'true', 'default_locale_name':'en'}`.

4. The main function first constructs a `Configurator` instance, passing a root resource factory (constructor) to it as its `root_factory` argument, and `settings` dictionary captured via the `**settings` kwarg as its `settings` argument.

The root resource factory is invoked on every request to retrieve the application's root resource. It is not called during startup, only when a request is handled.

The `settings` dictionary contains all the options in the `[app:MyProject]` section of our `.ini` file except the `use` option (which is internal to Paste) such as `reload_templates`, `debug_authorization`, etc.

5. The main function then calls various methods on the an instance of the class `Configurator` method. The intent of calling these methods is to populate an *application registry*, which represents the Pyramid configuration related to the application.
6. The `make_wsgi_app()` method is called. The result is a *router* instance. The router is associated with the *application registry* implied by the configurator previously populated by other methods run against the `Configurator`. The router is a WSGI application.
7. A `ApplicationCreated` event is emitted (see *Using Events* for more information about events).
8. Assuming there were no errors, the main function in `myproject` returns the router instance created by `make_wsgi_app` back to `PasteDeploy`. As far as `PasteDeploy` is concerned, it is "just another WSGI application".
9. `PasteDeploy` starts the WSGI *server* defined within the `[server:main]` section. In our case, this is the `Paste#http` server (`use = egg:Paste#http`), and it will listen on all interfaces (`host = 0.0.0.0`), on port number `6543` (`port = 6543`). The server code itself is what prints serving on `0.0.0.0:6543` view at `http://127.0.0.1:6543`. The server serves the application, and the application is running, waiting to receive requests.

27.2 Deployment Settings

Note that an augmented version of the values passed as `**settings` to the `Configurator` constructor will be available in Pyramid *view callable* code as `request.registry.settings`. You can create objects you wish to access later from view code, and put them into the dictionary you pass to the configurator as `settings`. They will then be present in the `request.registry.settings` dictionary at application runtime.

THREAD LOCALS

A *thread local* variable is a variable that appears to be a “global” variable to an application which uses it. However, unlike a true global variable, one thread or process serving the application may receive a different value than another thread or process when that variable is “thread local”.

When a request is processed, Pyramid makes two *thread local* variables available to the application: a “registry” and a “request”.

28.1 Why and How Pyramid Uses Thread Local Variables

How are thread locals beneficial to Pyramid and application developers who use Pyramid? Well, usually they’re decidedly **not**. Using a global or a thread local variable in any application usually makes it a lot harder to understand for a casual reader. Use of a thread local or a global is usually just a way to avoid passing some value around between functions, which is itself usually a very bad idea, at least if code readability counts as an important concern.

For historical reasons, however, thread local variables are indeed consulted by various Pyramid API functions. For example, the implementation of the `pyramid.security` function named `authenticated_userid()` retrieves the thread local *application registry* as a matter of course to find an *authentication policy*. It uses the `pyramid.threadlocal.get_current_registry()` function to retrieve the application registry, from which it looks up the authentication policy; it then uses the authentication policy to retrieve the authenticated user id. This is how Pyramid allows arbitrary authentication policies to be “plugged in”.

When they need to do so, Pyramid internals use two API functions to retrieve the *request* and *application registry*: `get_current_request()` and `get_current_registry()`. The former returns the

“current” request; the latter returns the “current” registry. Both `get_current_*` functions retrieve an object from a thread-local data structure. These API functions are documented in `pyramid.threadlocal`.

These values are thread locals rather than true globals because one Python process may be handling multiple simultaneous requests or even multiple Pyramid applications. If they were true globals, Pyramid could not handle multiple simultaneous requests or allow more than one Pyramid application instance to exist in a single Python process.

Because one Pyramid application is permitted to call *another* Pyramid application from its own *view* code (perhaps as a *WSGI* app with help from the `pyramid.wsgi.wsgiapp2()` decorator), these variables are managed in a *stack* during normal system operations. The stack instance itself is a `threading.local`.

During normal operations, the thread locals stack is managed by a *Router* object. At the beginning of a request, the Router pushes the application’s registry and the request on to the stack. At the end of a request, the stack is popped. The topmost request and registry on the stack are considered “current”. Therefore, when the system is operating normally, the very definition of “current” is defined entirely by the behavior of a pyramid *Router*.

However, during unit testing, no Router code is ever invoked, and the definition of “current” is defined by the boundary between calls to the `pyramid.config.Configurator.begin()` and `pyramid.config.Configurator.end()` methods (or between calls to the `pyramid.testing.setUp()` and `pyramid.testing.tearDown()` functions). These functions push and pop the threadlocal stack when the system is under test. See *Test Set Up and Tear Down* for the definitions of these functions.

Scripts which use Pyramid machinery but never actually start a WSGI server or receive requests via HTTP such as scripts which use the `pyramid.scripting` API will never cause any Router code to be executed. However, the `pyramid.scripting` APIs also push some values on to the thread locals stack as a matter of course. Such scripts should expect the `get_current_request()` function to always return `None`, and should expect the `get_current_registry()` function to return exactly the same *application registry* for every request.

28.2 Why You Shouldn’t Abuse Thread Locals

You probably should almost never use the `get_current_request()` or `get_current_registry()` functions, except perhaps in tests. In particular, it’s almost always a mistake to use `get_current_request` or `get_current_registry` in application code because its usage makes it possible to write code that can be neither easily tested nor scripted. Inappropriate usage is defined as follows:

- `get_current_request` should never be called within the body of a *view callable*, or within code called by a view callable. View callables already have access to the request (it's passed in to each as `request`).
- `get_current_request` should never be called in *resource* code. If a resource needs access to the request, it should be passed the request by a *view callable*.
- `get_current_request` function should never be called because it's "easier" or "more elegant" to think about calling it than to pass a request through a series of function calls when creating some API design. Your application should instead almost certainly pass data derived from the request around rather than relying on being able to call this function to obtain the request in places that actually have no business knowing about it. Parameters are *meant* to be passed around as function arguments, this is why they exist. Don't try to "save typing" or create "nicer APIs" by using this function in the place where a request is required; this will only lead to sadness later.
- Neither `get_current_request` nor `get_current_registry` should ever be called within application-specific forks of third-party library code. The library you've forked almost certainly has nothing to do with Pyramid, and making it dependent on Pyramid (rather than making your pyramid application depend upon it) means you're forming a dependency in the wrong direction.

Use of the `get_current_request()` function in application code *is* still useful in very limited circumstances. As a rule of thumb, usage of `get_current_request` is useful **within code which is meant to eventually be removed**. For instance, you may find yourself wanting to deprecate some API that expects to be passed a request object in favor of one that does not expect to be passed a request object. But you need to keep implementations of the old API working for some period of time while you deprecate the older API. So you write a "facade" implementation of the new API which calls into the code which implements the older API. Since the new API does not require the request, your facade implementation doesn't have local access to the request when it needs to pass it into the older API implementation. After some period of time, the older implementation code is disused and the hack that uses `get_current_request` is removed. This would be an appropriate place to use the `get_current_request`.

Use of the `get_current_registry()` function should be limited to testing scenarios. The registry made current by use of the `pyramid.config.Configurator.begin()` method during a test (or via `pyramid.testing.setUp()`) when you do not pass one in is available to you via this API.

USING THE ZOPE COMPONENT ARCHITECTURE IN PYRAMID

Under the hood, Pyramid uses a *Zope Component Architecture* component registry as its *application registry*. The Zope Component Architecture is referred to colloquially as the “ZCA.”

The `zope.component` API used to access data in a traditional Zope application can be opaque. For example, here is a typical “unnamed utility” lookup using the `zope.component.getUtility()` global API as it might appear in a traditional Zope application:

```
1 from pyramid.interfaces import ISettings
2 from zope.component import getUtility
3 settings = getUtility(ISettings)
```

After this code runs, `settings` will be a Python dictionary. But it’s unlikely that any “civilian” will be able to figure this out just by reading the code casually. When the `zope.component.getUtility` API is used by a developer, the conceptual load on a casual reader of code is high.

While the ZCA is an excellent tool with which to build a *framework* such as Pyramid, it is not always the best tool with which to build an *application* due to the opacity of the `zope.component` APIs. Accordingly, Pyramid tends to hide the the presence of the ZCA from application developers. You needn’t understand the ZCA to create a Pyramid application; its use is effectively only a framework implementation detail.

However, developers who are already used to writing *Zope* applications often still wish to use the ZCA while building a Pyramid application; `pyramid` makes this possible.

29.1 Using the ZCA Global API in a Pyramid Application

Zope uses a single ZCA registry – the “global” ZCA registry – for all *Zope* applications that run in the same Python process, effectively making it impossible to run more than one *Zope* application in a single process.

However, for ease of deployment, it’s often useful to be able to run more than a single application per process. For example, use of a *Paste* “composite” allows you to run separate individual WSGI applications in the same process, each answering requests for some URL prefix. This makes it possible to run, for example, a TurboGears application at `/turbogears` and a BFG application at `/bfg`, both served up using the same WSGI server within a single Python process.

Most production *Zope* applications are relatively large, making it impractical due to memory constraints to run more than one *Zope* application per Python process. However, a Pyramid application may be very small and consume very little memory, so it’s a reasonable goal to be able to run more than one BFG application per process.

In order to make it possible to run more than one Pyramid application in a single process, Pyramid defaults to using a separate ZCA registry *per application*.

While this services a reasonable goal, it causes some issues when trying to use patterns which you might use to build a typical *Zope* application to build a Pyramid application. Without special help, ZCA “global” APIs such as `zope.component.getUtility` and `zope.component.getSiteManager` will use the ZCA “global” registry. Therefore, these APIs will appear to fail when used in a Pyramid application, because they’ll be consulting the ZCA global registry rather than the component registry associated with your Pyramid application.

There are three ways to fix this: by disusing the ZCA global API entirely, by using `pyramid.config.Configurator.hook_zca()` or by passing the ZCA global registry to the *Configurator* constructor at startup time. We’ll describe all three methods in this section.

29.1.1 Disusing the Global ZCA API

ZCA “global” API functions such as `zope.component.getSiteManager`, `zope.component.getUtility`, `zope.component.getAdapter`, and `zope.component.getMultiAdapter` aren’t strictly necessary. Every component registry has a method API that offers the same functionality; it can be used instead. For example, presuming the `registry` value below is a Zope Component Architecture component registry, the following bit of code is equivalent to `zope.component.getUtility(IFoo)`:

```
1 registry.getUtility(IFoo)
```

The full method API is documented in the `zope.component` package, but it largely mirrors the “global” API almost exactly.

If you are willing to disuse the “global” ZCA APIs and use the method interface of a registry instead, you need only know how to obtain the Pyramid component registry.

There are two ways of doing so:

- use the `pyramid.threadlocal.get_current_registry()` function within Pyramid view or resource code. This will always return the “current” Pyramid application registry.
- use the attribute of the `request` object named `registry` in your Pyramid view code, eg. `request.registry`. This is the ZCA component registry related to the running Pyramid application.

See *Thread Locals* for more information about `pyramid.threadlocal.get_current_registry()`.

29.1.2 Enabling the ZCA Global API by Using `hook_zca`

Consider the following bit of idiomatic Pyramid startup code:

```
1 from zope.component import getGlobalSiteManager
2 from pyramid.config import Configurator
3
4 def app(global_settings, **settings):
5     config = Configurator(settings=settings)
6     config.include('some.other.package')
7     return config.make_wsgi_app()
```

When the `app` function above is run, a *Configurator* is constructed. When the configurator is created, it creates a *new application registry* (a ZCA component registry). A new registry is constructed whenever the `registry` argument is omitted when a *Configurator* constructor is called, or when a `registry` argument with a value of `None` is passed to a *Configurator* constructor.

During a request, the application registry created by the *Configurator* is “made current”. This means calls to `get_current_registry()` in the thread handling the request will return the component registry associated with the application.

29. USING THE ZOPE COMPONENT ARCHITECTURE IN PYRAMID

As a result, application developers can use `get_current_registry` to get the registry and thus get access to utilities and such, as per *Disusing the Global ZCA API*. But they still cannot use the global ZCA API. Without special treatment, the ZCA global APIs will always return the global ZCA registry (the one in `zope.component.globalregistry.base`).

To “fix” this and make the ZCA global APIs use the “current” BFG registry, you need to call `hook_zca()` within your setup code. For example:

```
1 from zope.component import getGlobalSiteManager
2 from pyramid.config import Configurator
3
4 def app(global_settings, **settings):
5     config = Configurator(settings=settings)
6     config.hook_zca()
7     config.include('some.other.application')
8     return config.make_wsgi_app()
```

We’ve added a line to our original startup code, line number 6, which calls `config.hook_zca()`. The effect of this line under the hood is that an analogue of the following code is executed:

```
1 from zope.component import getSiteManager
2 from pyramid.threadlocal import get_current_registry
3 getSiteManager.sethook(get_current_registry)
```

This causes the ZCA global API to start using the Pyramid application registry in threads which are running a Pyramid request.

Calling `hook_zca` is usually sufficient to “fix” the problem of being able to use the global ZCA API within a Pyramid application. However, it also means that a Zope application that is running in the same process may start using the Pyramid global registry instead of the Zope global registry, effectively inverting the original problem. In such a case, follow the steps in the next section, *Enabling the ZCA Global API by Using The ZCA Global Registry*.

29.1.3 Enabling the ZCA Global API by Using The ZCA Global Registry

You can tell your Pyramid application to use the ZCA global registry at startup time instead of constructing a new one:


```
1 from zope.component import getGlobalSiteManager
2 from pyramid.config import Configurator
3
4 def app(global_settings, **settings):
5     globalreg = getGlobalSiteManager()
6     config = Configurator(registry=globalreg)
7     config.setup_registry(settings=settings)
8     config.hook_zca()
9     config.include('some.other.application')
10    return config.make_wsgi_app()
```

Lines 5, 6, and 7 above are the interesting ones. Line 5 retrieves the global ZCA component registry. Line 6 creates a *Configurator*, passing the global ZCA registry into its constructor as the `registry` argument. Line 7 “sets up” the global registry with BFG-specific registrations; this is code that is normally executed when a registry is constructed rather than created, but we must call it “by hand” when we pass an explicit registry.

At this point, Pyramid will use the ZCA global registry rather than creating a new application-specific registry; since by default the ZCA global API will use this registry, things will work as you might expect a Zope app to when you use the global ZCA API.

Part II

Tutorials

ZODB + TRAVERSAL WIKI TUTORIAL

This tutorial introduces a *traversal*-based Pyramid application to a developer familiar with Python. It will be most familiar to developers with previous *Zope* experience. When we're done with the tutorial, the developer will have created a basic Wiki application with authentication.

For cut and paste purposes, the source code for all stages of this tutorial can be browsed at <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki>.

30.1 Background

This version of the Pyramid wiki tutorial presents a Pyramid application that uses technologies which will be familiar to someone with *Zope* experience. It uses *ZODB* as a persistence mechanism and *traversal* to map URLs to code. It can also be followed by people without any prior Python web framework experience.

To code along with this tutorial, the developer will need a UNIX machine with development tools (Mac OS X with XCode, any Linux or BSD variant, etc) *or* he will need a Windows system of any kind.

This tutorial targets Pyramid version 1.0.

Have fun!

30.2 Installation

For the most part, the installation process for this tutorial duplicates the steps described in *Installing Pyramid* and *Creating a Pyramid Project*, however it also explains how to install additional libraries for tutorial purposes.

30.2.1 Preparation

Please take the following steps to prepare for the tutorial. The steps to prepare for the tutorial are slightly different depending on whether you're using UNIX or Windows.

Preparation, UNIX

1. If you don't already have a Python 2.6 interpreter installed on your system, obtain, install, or find Python 2.6 for your system.
2. Install the latest *setuptools* into the Python you obtained/installed/found in the step above: download `ez_setup.py` and run it using the `python` interpreter of your Python 2.6 installation:

```
$ /path/to/my/Python-2.6/bin/python ez_setup.py
```

3. Use that Python's `bin/easy_install` to install *virtualenv*:

```
$ /path/to/my/Python-2.6/bin/easy_install virtualenv
```

4. Use that Python's *virtualenv* to make a workspace:

```
$ path/to/my/Python-2.6/bin/virtualenv --no-site-packages \  
pyramidtut
```

5. Switch to the `pyramidtut` directory:

```
$ cd pyramidtut
```

6. (Optional) Consider using `source bin/activate` to make your shell environment wired to use the *virtualenv*.
7. Use `easy_install` to get *Pyramid* and its direct dependencies installed:

```
$ bin/easy_install pyramid
```

8. Use `easy_install` to install `docutils`, `repoze.tm`, `repoze.zodbconn`, `nose` and `coverage`:

```
$ bin/easy_install docutils repoze.tm repoze.zodbconn \  
nose coverage
```

Preparation, Windows

1. Install, or find Python 2.6 for your system.
2. Install the latest `setuptools` into the Python you obtained/installed/found in the step above: download `ez_setup.py` and run it using the `python` interpreter of your Python 2.6 installation using a command prompt:

```
c:\> c:\Python26\python ez_setup.py
```

3. Use that Python's `bin/easy_install` to install `virtualenv`:

```
c:\> c:\Python26\Scripts\easy_install virtualenv
```

4. Use that Python's `virtualenv` to make a workspace:

```
c:\> c:\Python26\Scripts\virtualenv --no-site-packages pyramidtut
```

5. Switch to the `pyramidtut` directory:

```
c:\> cd pyramidtut
```

6. (Optional) Consider using `bin\activate.bat` to make your shell environment wired to use the `virtualenv`.
7. Use `easy_install` to get Pyramid and its direct dependencies installed:

```
c:\pyramidtut> Scripts\easy_install pyramid
```

8. Use `easy_install` to install `docutils`, `repoze.tm`, `repoze.zodbconn`, `nose` and `coverage`:

```
c:\pyramidtut> Scripts\easy_install docutils repoze.tm \  
repoze.zodbconn nose coverage
```

30.2.2 Making a Project

Your next step is to create a project. Pyramid supplies a variety of templates to generate sample projects. For this tutorial, we will use the *ZODB*-oriented template named `pyramid_zodb`.

The below instructions assume your current working directory is the “virtualenv” named “pyramidtut”.

On UNIX:

```
$ bin/paster create -t pyramid_zodb tutorial
```

On Windows:

```
c:\pyramidtut> Scripts\paster create -t pyramid_zodb tutorial
```

i If you are using Windows, the `pyramid_zodb` Paster template doesn’t currently deal gracefully with installation into a location that contains spaces in the path. If you experience startup problems, try putting both the virtualenv and the project into directories that do not contain spaces in their paths.

30.2.3 Installing the Project in “Development Mode”

In order to do development on the project easily, you must “register” the project as a development egg in your workspace using the `setup.py develop` command. In order to do so, `cd` to the “tutorial” directory you created in *Making a Project*, and run the “`setup.py develop`” command using virtualenv Python interpreter.

On UNIX:


```
$ cd tutorial
$ ../bin/python setup.py develop
```

On Windows:

```
C:\pyramidtut> cd tutorial
C:\pyramidtut\tutorial> ..\Scripts\python setup.py develop
```

30.2.4 Running the Tests

After you've installed the project in development mode, you may run the tests for the project.

On UNIX:

```
$ ../bin/python setup.py test -q
```

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\python setup.py test -q
```

30.2.5 Starting the Application

Start the application.

On UNIX:

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

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\paster serve development.ini --reload
```

30.2.6 Exposing Test Coverage Information

You can run the `nosetests` command to see test coverage information. This runs the tests in the same way that `setup.py test` does but provides additional “coverage” information, exposing which lines of your project are “covered” (or not covered) by the tests.

On UNIX:

```
$ ../bin/nosetests --cover-package=tutorial --cover-erase --with-coverage
```

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\nosetests --cover-package=tutorial \  
--cover-erase --with-coverage
```

Looks like the code in the `pyramid_zodb` template for ZODB projects is missing some test coverage, particularly in the file named `models.py`.


30.2.7 Visit the Application in a Browser

In a browser, visit `http://localhost:6543/`. You will see the generated application’s default page.

30.2.8 Decisions the `pyramid_zodb` Template Has Made For You

Creating a project using the `pyramid_zodb` template makes the following assumptions:

- you are willing to use *ZODB* as persistent storage
- you are willing to use *traversal* to map URLs to code.
- you want to use imperative code plus a *scan* to perform configuration.

 Pyramid supports any persistent storage mechanism (e.g. a SQL database or filesystem files, etc). Pyramid also supports an additional mechanism to map URLs to code (*URL dispatch*). However, for the purposes of this tutorial, we’ll only be using traversal and ZODB.

30.3 Basic Layout

The starter files generated by the `pyramid_zodb` template are basic, but they provide a good orientation for the high-level patterns common to most *traversal*-based Pyramid (and *ZODB* based) projects.

The source code for this tutorial stage can be browsed via <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki/src/basiclayout/>.

30.3.1 App Startup with `__init__.py`

A directory on disk can be turned into a Python *package* by containing an `__init__.py` file. Even if empty, this marks a directory as a Python package. Our application uses `__init__.py` as both a package marker, as well as to contain application configuration code.

When you run the application using the `paster` command using the `development.ini` generated config file, the application configuration points at an *Setuptools entry point* described as `egg:tutorial`. In our application, because the application's `setup.py` file says so, this entry point happens to be the main function within the file named `__init__.py`:

```

1 from pyramid.config import Configurator
2 from repoze.zodbconn.finder import PersistentApplicationFinder
3 from tutorial.models import appmaker
4
5 def main(global_config, **settings):
6     """ This function returns a Pyramid WSGI application.
7     """
8     zodb_uri = settings.get('zodb_uri')
9     if zodb_uri is None:
10        raise ValueError("No 'zodb_uri' in application configuration.")
11
12    finder = PersistentApplicationFinder(zodb_uri, appmaker)
13    def get_root(request):
14        return finder(request.environ)
15    config = Configurator(root_factory=get_root, settings=settings)
16    config.add_static_view('static', 'tutorial:static')
17    config.scan('tutorial')
18    return config.make_wsgi_app()

```

1. *Lines 1-3.* Perform some dependency imports.
2. *Line 8.* Get the *ZODB* configuration from the `development.ini` file's `[app:main]` section represented by the `settings` dictionary passed to our app function. This will be a URI (something like `file:///path/to/Data.fs`).

3. *Line 12.* We create a “finder” object using the `PersistentApplicationFinder` helper class, passing it the ZODB URI and the “appmaker” we’ve imported from `models.py`.
4. *Lines 13 - 14.* We create a *root factory* which uses the finder to return a ZODB root object.
5. *Line 15.* We construct a *Configurator* with a *root factory* and the settings keywords parsed by PasteDeploy. The root factory is named `get_root`.
6. *Line 16.* Register a ‘static view’ which answers requests which start with with URL path `/static` using the `pyramid.config.Configurator.add_static_view` method(). This statement registers a view that will serve up static assets, such as CSS and image files, for us, in this case, at `http://localhost:6543/static/` and below. The first argument is the “name” `static`, which indicates that the URL path prefix of the view will be `/static`. The second argument of this tag is the “path”, which is an *asset specification*, so it finds the resources it should serve within the `static` directory inside the `tutorial` package.
7. *Line 17.* Perform a *scan*. A scan will find *configuration decoration*, such as view configuration decorators (e.g. `@view_config`) in the source code of the `tutorial` package and will take actions based on these decorators. The argument to `scan()` is the package name to scan, which is `tutorial`.
8. *Line 18.* Use the `pyramid.config.Configurator.make_wsgi_app()` method to return a WSGI application.

30.3.2 Resources and Models with `models.py`

Pyramid uses the word *resource* to describe objects arranged hierarchically in a *resource tree*. This tree is consulted by *traversal* to map URLs to code. In this application, the resource tree represents the site structure, but it *also* represents the *domain model* of the application, because each resource is a node stored persistently in a ZODB database. The `models.py` file is where the `pyramid_zodb` Paster template put the classes that implement our resource objects, each of which happens also to be a domain model object.

Here is the source for `models.py`:

```
1 from persistent.mapping import PersistentMapping
2
3 class MyModel(PersistentMapping):
4     __parent__ = __name__ = None
5
6 def appmaker(zodb_root):
7     if not 'app_root' in zodb_root:
```

```

8         app_root = MyModel()
9         zodb_root['app_root'] = app_root
10        import transaction
11        transaction.commit()
12        return zodb_root['app_root']

```

1. *Lines 3-4.* The `MyModel` *resource* class is implemented here. Instances of this class will be capable of being persisted in *ZODB* because the class inherits from the `persistent.mapping.PersistentMapping` class. The `__parent__` and `__name__` are important parts of the *traversal* protocol. By default, have these as `None` indicating that this is the *root* object.
2. *Lines 6-12.* `appmaker` is used to return the *application root* object. It is called on *every request* to the Pyramid application. It also performs bootstrapping by *creating* an application root (inside the *ZODB* root object) if one does not already exist.

We do so by first seeing if the database has the persistent application root. If not, we make an instance, store it, and commit the transaction. We then return the application root object.

30.3.3 Views With `views.py`

Our paster template generated a default `views.py` on our behalf. It contains a single view, which is used to render the page shown when you visit the URL `http://localhost:6543/`.

Here is the source for `views.py`:

```

1 from pyramid.view import view_config
2 from tutorial.models import MyModel
3
4 @view_config(context=MyModel,
5             renderer='tutorial:templates/mytemplate.pt')
6 def my_view(request):
7     return {'project': 'tutorial'}

```

Let's try to understand the components in this module:

1. *Lines 1-2.* Perform some dependency imports.

2. *Line 4.* Use the `pyramid.view.view_config()` *configuration decoration* to perform a *view configuration* registration. This view configuration registration will be activated when the application is started. It will be activated by virtue of it being found as the result of a *scan* (when Line 17 of `__init__.py` is run).

The `@view_config` decorator accepts a number of keyword arguments. We use two keyword arguments here: `context` and `renderer`.

The `context` argument signifies that the decorated view callable should only be run when *traversal* finds the `tutorial.models.MyModel` *resource* to be the *context* of a request. In English, this means that when the URL `/` is visited, because `MyModel` is the root model, this view callable will be invoked.

The `renderer` argument names an *asset specification* of `tutorial:templates/mytemplate.pt`. This asset specification points at a *Chameleon* template which lives in the `mytemplate.pt` file within the `templates` directory of the `tutorial` package. And indeed if you look in the `templates` directory of this package, you'll see a `mytemplate.pt` template file, which renders the default home page of the generated project.

Since this call to `@view_config` doesn't pass a `name` argument, the `my_view` function which it decorates represents the "default" view callable used when the context is of the type `MyModel`.

3. *Lines 5-6.* We define a *view callable* named `my_view`, which we decorated in the step above. This view callable is a *function* we write generated by the `pyramid_zodb` template that is given a *request* and which returns a dictionary. The `mytemplate.pt` *renderer* named by the asset specification in the step above will convert this dictionary to a *response* on our behalf.

The function returns the dictionary `{ 'project' : 'tutorial' }`. This dictionary is used by the template named by the `mytemplate.pt` asset specification to fill in certain values on the page.

30.3.4 The WSGI Pipeline in `development.ini`

The `development.ini` (in the `tutorial project` directory, as opposed to the `tutorial package` directory) looks like this:

```
[app:tutorial]
use = egg:tutorial
reload_templates = true
debug_authorization = false
debug_notfound = false
debug_routematch = false
```

```
debug_templates = true
default_locale_name = en
zodb_uri = file://%(here)s/Data.fs?connection_cache_size=20000

[pipeline:main]
pipeline =
    egg:WebError#evalerror
    egg:repoze.zodbconn#closer
    egg:repoze.retry#retry
    tm
    tutorial

[filter:tm]
use = egg:repoze.tm2#tm
commit_veto = repoze.tm:default_commit_veto

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 6543

# Begin logging configuration

[loggers]
keys = root

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

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

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

# End logging configuration
```

Note the existence of a `[pipeline:main]` section which specifies our WSGI pipeline. This “pipeline” will be served up as our WSGI application. As far as the WSGI server is concerned the pipeline *is* our application. Simpler configurations don’t use a pipeline: instead they expose a single WSGI application as “main”. Our setup is more complicated, so we use a pipeline composed of *middleware*.

The `egg:WebError#evalerror` middleware is at the “top” of the pipeline. This is middleware which displays debuggable errors in the browser while you’re developing (not recommended for deployment).

The `egg:repoze.zodbconn#closer` middleware is in the middle of the pipeline. This is a piece of middleware which closes the ZODB connection opened by the `PersistentApplicationFinder` at the end of the request.

The `egg:repoze.retry#retry` middleware catches `ConflictError` exceptions from ZODB and retries the request up to three times (ZODB is an optimistic concurrency database that relies on application-level transaction retries when a conflict occurs).

The `tm` middleware is the last piece of middleware in the pipeline. This commits a transaction near the end of the request unless there’s an exception raised or the HTTP response code is an error code. The `tm` refers to the `[filter:tm]` section beneath the pipeline declaration, which configures the transaction manager.

The final line in the `[pipeline:main]` section is `tutorial`, which refers to the `[app:tutorial]` section above it. The `[app:tutorial]` section is the section which actually defines our application settings. The values within this section are passed as `**settings` to the main function we defined in `__init__.py` when the server is started via `paster serve`.

30.4 Defining the Domain Model

The first change we’ll make to our bone-stock `paster`-generated application will be to define a number of *resource* constructors. Remember that, because we’re using *ZODB* to represent our *resource tree*, each of these resource constructors represents a *domain model* object, so we’ll call these constructors “model constructors”. For this application, which will be a Wiki, we will need two kinds of model constructors: a “Wiki” model constructor, and a “Page” model constructor. Both our Page and Wiki constructors will be class objects. A single instance of the “Wiki” class will serve as a container for “Page” objects, which will be instances of the “Page” class.

The source code for this tutorial stage can be browsed via <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki/src/models/>.

30.4.1 Deleting the Database

In a subsequent step, we're going to remove the `MyModel` Python model class from our `models.py` file. Since this class is referred to within our persistent storage (represented on disk as a file named `Data.fs`), we'll have strange things happen the next time we want to visit the application in a browser. Remove the `Data.fs` from the `tutorial` directory before proceeding any further. It's always fine to do this as long as you don't care about the content of the database; the database itself will be recreated as necessary.

30.4.2 Adding Model Classes

The next thing we want to do is remove the `MyModel` class from the generated `models.py` file. The `MyModel` class is only a sample and we're not going to use it.



There is nothing automatically special about the filename `models.py`. A project may have many models throughout its codebase in arbitrarily-named files. Files implementing models often have `model` in their filenames, or they may live in a Python subpackage of your application package named `models`, but this is only by convention.

Then, we'll add a `Wiki` class. Because this is a ZODB application, this class should inherit from `persistent.mapping.PersistentMapping`. We want it to inherit from the `persistent.mapping.PersistentMapping` class because our `Wiki` class will be a mapping of wiki page names to `Page` objects. The `persistent.mapping.PersistentMapping` class provides our class with mapping behavior, and makes sure that our `Wiki` page is stored as a "first-class" persistent object in our ZODB database.

Our `Wiki` class should also have a `__name__` attribute set to `None` at class scope, and should have a `__parent__` attribute set to `None` at class scope as well. If a model has a `__parent__` attribute of `None` in a traversal-based Pyramid application, it means that it's the *root* model. The `__name__` of the root model is also always `None`.

Then we'll add a `Page` class. This class should inherit from the `persistent.Persistent` class. We'll also give it an `__init__` method that accepts a single parameter named `data`. This parameter will contain the *ReStructuredText* body representing the wiki page content. Note that `Page` objects don't have an initial `__name__` or `__parent__` attribute. All objects in a traversal graph must have a `__name__` and a `__parent__` attribute. We don't specify these here because both `__name__` and `__parent__` will be set by by a *view* function when a `Page` is added to our `Wiki` mapping.

As a last step, we want to change the `appmaker` function in our `models.py` file so that the *root resource* of our application is a `Wiki` instance. We'll also slot a single page object (the front page) into the

Wiki within the appmaker. This will provide *traversal* a *resource tree* to work against when it attempts to resolve URLs to resources.

We're using a mini-framework callable named `PersistentApplicationFinder` in our application (see `__init__.py`). A `PersistentApplicationFinder` accepts a ZODB URL as well as an "appmaker" callback. This callback typically lives in the `models.py` file. We'll just change this function, making the necessary edits.

30.4.3 Looking at the Result of Our Edits to `models.py`

The result of all of our edits to `models.py` will end up looking something like this:

```
1 from persistent import Persistent
2 from persistent.mapping import PersistentMapping
3
4 class Wiki(PersistentMapping):
5     __name__ = None
6     __parent__ = None
7
8 class Page(Persistent):
9     def __init__(self, data):
10         self.data = data
11
12 def appmaker(zodb_root):
13     if not 'app_root' in zodb_root:
14         app_root = Wiki()
15         frontpage = Page('This is the front page')
16         app_root['FrontPage'] = frontpage
17         frontpage.__name__ = 'FrontPage'
18         frontpage.__parent__ = app_root
19         zodb_root['app_root'] = app_root
20         import transaction
21         transaction.commit()
22     return zodb_root['app_root']
```

30.4.4 Removing View Configuration

In a previous step in this chapter, we removed the `tutorial.models.MyModel` class. However, our `views.py` module still attempts to import this class. Temporarily, we'll change `views.py` so that it no longer references `MyModel` by removing its imports and removing a reference to it from the arguments

passed to the `@view_config` *configuration decoration* decorator which sits atop the `my_view` view callable.

The result of all of our edits to `views.py` will end up looking something like this:

```

1 from pyramid.view import view_config
2
3 @view_config(renderer='tutorial:templates/mytemplate.pt')
4 def my_view(request):
5     return {'project': 'tutorial'}
```

30.4.5 Testing the Models

To make sure the code we just wrote works, we write tests for the model classes and the appmaker. Changing `tests.py`, we'll write a separate test class for each model class, and we'll write a test class for the appmaker.

To do so, we'll retain the `tutorial.tests.ViewTests` class provided as a result of the `pyramid_zodb` project generator. We'll add three test classes: one for the `Page` model named `PageModelTests`, one for the `Wiki` model named `WikiModelTests`, and one for the appmaker named `AppmakerTests`.

When we're done changing `tests.py`, it will look something like so:

```

1 import unittest
2
3 from pyramid import testing
4
5 class PageModelTests(unittest.TestCase):
6
7     def _getTargetClass(self):
8         from tutorial.models import Page
9         return Page
10
11     def _makeOne(self, data=u'some data'):
12         return self._getTargetClass()(data=data)
13
14     def test_constructor(self):
15         instance = self._makeOne()
16         self.assertEqual(instance.data, u'some data')
17
18 class WikiModelTests(unittest.TestCase):
```

```
19
20     def _getTargetClass(self):
21         from tutorial.models import Wiki
22         return Wiki
23
24     def _makeOne(self):
25         return self._getTargetClass()()
26
27     def test_it(self):
28         wiki = self._makeOne()
29         self.assertEqual(wiki.__parent__, None)
30         self.assertEqual(wiki.__name__, None)
31
32 class AppmakerTests(unittest.TestCase):
33
34     def _callFUT(self, zodb_root):
35         from tutorial.models import appmaker
36         return appmaker(zodb_root)
37
38     def test_no_app_root(self):
39         root = {}
40         self._callFUT(root)
41         self.assertEqual(root['app_root']['FrontPage'].data,
42                          'This is the front page')
43
44     def test_w_app_root(self):
45         app_root = object()
46         root = {'app_root': app_root}
47         self._callFUT(root)
48         self.failUnless(root['app_root'] is app_root)
49
50 class ViewTests(unittest.TestCase):
51     def setUp(self):
52         self.config = testing.setUp()
53
54     def tearDown(self):
55         testing.tearDown()
56
57     def test_my_view(self):
58         from tutorial.views import my_view
59         request = testing.DummyRequest()
60         info = my_view(request)
61         self.assertEqual(info['project'], 'tutorial')
```

30.4.6 Declaring Dependencies in Our `setup.py` File

Our application now depends on packages which are not dependencies of the original “tutorial” application as it was generated by the `paster create` command. We’ll add these dependencies to our tutorial package’s `setup.py` file by assigning these dependencies to both the `install_requires` and the `tests_require` parameters to the `setup` function. In particular, we require the `docutils` package.

Our resulting `setup.py` should look like so:

```
1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 README = open(os.path.join(here, 'README.txt')).read()
7 CHANGES = open(os.path.join(here, 'CHANGES.txt')).read()
8
9 requires = [
10     'pyramid',
11     'repoze.zodbconn',
12     'repoze.tm2>=1.0b1', # default_commit_veto
13     'repoze.retry',
14     'ZODB3',
15     'WebError',
16     'docutils',
17 ]
18
19 setup(name='tutorial',
20       version='0.0',
21       description='tutorial',
22       long_description=README + '\n\n' + CHANGES,
23       classifiers=[
24         "Intended Audience :: Developers",
25         "Framework :: Pylons",
26         "Programming Language :: Python",
27         "Topic :: Internet :: WWW/HTTP",
28         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
29     ],
30       author='',
31       author_email='',
32       url='',
33       keywords='web pylons pyramid',
34       packages=find_packages(),
35       include_package_data=True,
36       zip_safe=False,
```

```
37     install_requires=requires,
38     tests_require=requires,
39     test_suite="tutorial",
40     entry_points = """\
41     [paste.app_factory]
42     main = tutorial:main
43     """ ,
44     paster_plugins=['pyramid'],
45     )
```

30.4.7 Running the Tests

We can run these tests by using `setup.py test` in the same way we did in *Running the Tests*. Assuming our shell's current working directory is the “tutorial” distribution directory:

On UNIX:

```
$ ../bin/python setup.py test -q
```

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\python setup.py test -q
```

The expected output is something like this:

```
.....
-----
Ran 5 tests in 0.008s

OK
```

30.5 Defining Views

Conventionally, *view callable* objects are defined within a `views.py` module in an Pyramid application. There is nothing automatically special about the filename `views.py`. Files implementing views often have `view` in their filenames (or may live in a Python subpackage of your application package named `views`), but this is only by convention. A project may have many views throughout its codebase in

arbitrarily-named files. In this application, however, we'll be continuing to use the `views.py` module, because there's no reason to break convention.

A *view callable* in a Pyramid application is typically a simple Python function that accepts a single parameter: *request*. A view callable is assumed to return a *response* object.

However, a Pyramid view can also be defined as callable which accepts *two* arguments: a *context* and a *request*. In *url dispatch* based applications, the context resource is rarely used in the view body itself, so within code that uses URL-dispatch-only, it's common to define views as callables that accept only a *request* to avoid the visual “noise” of a *context* argument. This application, however, uses *traversal* to map URLs to a context *resource*, and since our *resource tree* also represents our application's “domain model”, we're often interested in the context, because it represents the persistent storage of our application. For this reason, having *context* in the callable argument list is not “noise” to us; instead it's actually rather important within the view code we'll define in this application.

The single-arg (*request* -only) or two-arg (*context* and *request*) calling conventions will work in any Pyramid application for any view; they can be used interchangeably as necessary. We'll be using the two-argument (*context*, *request*) view callable argument list syntax in this application.

We're going to define several *view callable* functions then wire them into Pyramid using some *view configuration*.

The source code for this tutorial stage can be browsed via <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki/src/views/>.

30.5.1 Adding View Functions

We're going to add four *view callable* functions to our `views.py` module. One view named `view_wiki` will display the wiki itself (it will answer on the root URL), another named `view_page` will display an individual page, another named `add_page` will allow a page to be added, and a final view named `edit_page` will allow a page to be edited.

The `view_wiki` view function

The `view_wiki` function will be configured to respond as the default view callable for a Wiki resource. We'll provide it with a `@view_config` decorator which names the class `tutorial.models.Wiki` as its context. This means that when a Wiki resource is the context, and no *view name* exists in the request, this view will be used. The view configuration associated with `view_wiki` does not use a `renderer` because the view callable always returns a *response* object rather than a dictionary. No `renderer` is necessary when a view returns a response object.

The `view_wiki` view callable always redirects to the URL of a Page resource named “FrontPage”. To do so, it returns an instance of the `pyramid.httpexceptions.HTTPFound` class (instances of which implement the `WebOb response` interface). The `pyramid.url.resource_url()` API. `pyramid.url.resource_url()` constructs a URL to the `FrontPage` page resource (e.g. `http://localhost:6543/FrontPage`), and uses it as the “location” of the `HTTPFound` response, forming an HTTP redirect.

The `view_page` view function

The `view_page` function will be configured to respond as the default view of a Page resource. We’ll provide it with a `@view_config` decorator which names the class `tutorial.models.Wiki` as its context. This means that when a Page resource is the context, and no *view name* exists in the request, this view will be used. We inform Pyramid this view will use the `templates/view.pt` template file as a `renderer`.

The `view_page` function generates the *ReStructuredText* body of a page (stored as the `data` attribute of the context passed to the view; the context will be a Page resource) as HTML. Then it substitutes an HTML anchor for each *WikiWord* reference in the rendered HTML using a compiled regular expression.

The curried function named `check` is used as the first argument to `wikiwords.sub`, indicating that it should be called to provide a value for each WikiWord match found in the content. If the wiki (our page’s `__parent__`) already contains a page with the matched WikiWord name, the `check` function generates a view link to be used as the substitution value and returns it. If the wiki does not already contain a page with with the matched WikiWord name, the function generates an “add” link as the substitution value and returns it.

As a result, the `content` variable is now a fully formed bit of HTML containing various view and add links for WikiWords based on the content of our current page resource.

We then generate an edit URL (because it’s easier to do here than in the template), and we wrap up a number of arguments in a dictionary and return it.

The arguments we wrap into a dictionary include `page`, `content`, and `edit_url`. As a result, the *template* associated with this view callable (via `renderer=` in its configuration) will be able to use these names to perform various rendering tasks. The template associated with this view callable will be a template which lives in `templates/view.pt`.

Note the contrast between this view callable and the `view_wiki` view callable. In the `view_wiki` view callable, we unconditionally return a *response* object. In the `view_page` view callable, we return a *dictionary*. It is *always* fine to return a *response* object from a Pyramid view. Returning a dictionary is allowed only when there is a *renderer* associated with the view callable in the view configuration.

The `add_page` view function

The `add_page` function will be configured to respond when the context resource is a Wiki and the *view name* is `add_page`. We'll provide it with a `@view_config` decorator which names the string `add_page` as its *view name* (via `name=`), the class `tutorial.models.Wiki` as its context, and the renderer named `templates/edit.pt`. This means that when a Wiki resource is the context, and a *view name* named `add_page` exists as the result of traversal, this view will be used. We inform Pyramid this view will use the `templates/edit.pt` template file as a renderer. We share the same template between add and edit views, thus `edit.pt` instead of `add.pt`.

The `add_page` function will be invoked when a user clicks on a WikiWord which isn't yet represented as a page in the system. The `check` function within the `view_page` view generates URLs to this view. It also acts as a handler for the form that is generated when we want to add a page resource. The context of the `add_page` view is always a Wiki resource (*not* a Page resource).

The request *subpath* in Pyramid is the sequence of names that are found *after* the *view name* in the URL segments given in the `PATH_INFO` of the WSGI request as the result of *traversal*. If our add view is invoked via, e.g. `http://localhost:6543/add_page/SomeName`, the *subpath* will be a tuple: `('SomeName',)`.

The add view takes the zeroth element of the subpath (the wiki page name), and aliases it to the `name` attribute in order to know the name of the page we're trying to add.

If the view rendering is *not* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `False`), the view renders a template. To do so, it generates a "save url" which the template use as the form post URL during rendering. We're lazy here, so we're trying to use the same template (`templates/edit.pt`) for the add view as well as the page edit view. To do so, we create a dummy Page resource object in order to satisfy the edit form's desire to have *some* page object exposed as page, and we'll render the template to a response.

If the view rendering *is* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `True`), we scrape the page body from the form data, create a Page object using the name in the subpath and the page body, and save it into "our context" (the Wiki) using the `__setitem__` method of the context. We then redirect back to the `view_page` view (the default view for a page) for the newly created page.

The `edit_page` view function

The `edit_page` function will be configured to respond when the context is a Page resource and the *view name* is `edit_page`. We'll provide it with a `@view_config` decorator which names the string `edit_page` as its *view name* (via `name=`), the class `tutorial.models.Page` as its context, and

the renderer named `templates/edit.pt`. This means that when a `Page` resource is the context, and a *view name* exists as the result of traversal named `edit_page`, this view will be used. We inform Pyramid this view will use the `templates/edit.pt` template file as a renderer.

The `edit_page` function will be invoked when a user clicks the “Edit this Page” button on the view form. It renders an edit form but it also acts as the form post view callable for the form it renders. The context of the `edit_page` view will *always* be a `Page` resource (never a `Wiki` resource).

If the view execution is *not* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `False`), the view simply renders the edit form, passing the request, the page resource, and a `save_url` which will be used as the action of the generated form.

If the view execution *is* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `True`), the view grabs the `body` element of the request parameter and sets it as the `data` attribute of the page context. It then redirects to the default view of the context (the page), which will always be the `view_page` view.

30.5.2 Viewing the Result of Our Edits to `views.py`

The result of all of our edits to `views.py` will leave it looking like this:

```
1 from docutils.core import publish_parts
2 import re
3
4 from pyramid.httpexceptions import HTTPFound
5 from pyramid.url import resource_url
6 from pyramid.view import view_config
7
8 from tutorial.models import Page
9
10 # regular expression used to find WikiWords
11 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+) ")
12
13 @view_config(context='tutorial.models.Wiki')
14 def view_wiki(context, request):
15     return HTTPFound(location=resource_url(context, request, 'FrontPage'))
16
17 @view_config(context='tutorial.models.Page',
18             renderer='tutorial:templates/view.pt')
19 def view_page(context, request):
20     wiki = context.__parent__
21
22     def check(match):
```

```

23     word = match.group(1)
24     if word in wiki:
25         page = wiki[word]
26         view_url = resource_url(page, request)
27         return '<a href="%s">%s</a>' % (view_url, word)
28     else:
29         add_url = request.application_url + '/add_page/' + word
30         return '<a href="%s">%s</a>' % (add_url, word)
31
32     content = publish_parts(context.data, writer_name='html')['html_body']
33     content = wikiwords.sub(check, content)
34     edit_url = resource_url(context, request, 'edit_page')
35     return dict(page = context, content = content, edit_url = edit_url)
36
37 @view_config(name='add_page', context='tutorial.models.Wiki',
38             renderer='tutorial:templates/edit.pt')
39 def add_page(context, request):
40     name = request.subpath[0]
41     if 'form.submitted' in request.params:
42         body = request.params['body']
43         page = Page(body)
44         page.__name__ = name
45         page.__parent__ = context
46         context[name] = page
47         return HTTPFound(location = resource_url(page, request))
48     save_url = resource_url(context, request, 'add_page', name)
49     page = Page('')
50     page.__name__ = name
51     page.__parent__ = context
52     return dict(page = page, save_url = save_url)
53
54 @view_config(name='edit_page', context='tutorial.models.Page',
55             renderer='tutorial:templates/edit.pt')
56 def edit_page(context, request):
57     if 'form.submitted' in request.params:
58         context.data = request.params['body']
59         return HTTPFound(location = resource_url(context, request))
60
61     return dict(page = context,
62               save_url = resource_url(context, request, 'edit_page'))

```

30.5.3 Adding Templates

Most view callables we've added expected to be rendered via a *template*. The default templating systems in Pyramid are *Chameleon* and *Mako*. Chameleon is a variant of *ZPT*, which is an XML-based templating

language. Mako is a non-XML-based templating language. Because we had to pick one, we chose Chameleon for this tutorial.

The templates we create will live in the `templates` directory of our tutorial package. Chameleon templates must have a `.pt` extension to be recognized as such.

The `view.pt` Template

The `view.pt` template is used for viewing a single Page. It is used by the `view_page` view function. It should have a div that is “structure replaced” with the `content` value provided by the view. It should also have a link on the rendered page that points at the “edit” URL (the URL which invokes the `edit_page` view for the page being viewed).

Once we’re done with the `view.pt` template, it will look a lot like the below:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
  <title>${page.__name__} - Pyramid tutorial wiki (based on
    TurboGears 20-Minute Wiki)</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <meta name="keywords" content="python web application" />
  <meta name="description" content="pyramid web application" />
  <link rel="shortcut icon"
    href="${request.static_url('tutorial:static/favicon.ico')}" />
  <link rel="stylesheet"
    href="${request.static_url('tutorial:static/pylons.css')}"
    type="text/css" media="screen" charset="utf-8" />
  <!--[if lte IE 6]>
  <link rel="stylesheet"
    href="${request.static_url('tutorial:static/ie6.css')}"
    type="text/css" media="screen" charset="utf-8" />
  <![endif]-->
</head>
<body>
  <div id="wrap">
    <div id="top-small">
      <div class="top-small align-center">
        <div>
          
        </div>
```

```

    </div>
</div>
<div id="middle">
  <div class="middle align-right">
    <div id="left" class="app-welcome align-left">
      Viewing <b><span tal:replace="page.__name__">Page Name Goes
      Here</span></b><br/>
      You can return to the
      <a href="{request.application_url}">FrontPage</a>.<br/>
    </div>
    <div id="right" class="app-welcome align-right"></div>
  </div>
</div>
<div id="bottom">
  <div class="bottom">
    <div tal:replace="structure content">
      Page text goes here.
    </div>
    <p>
      <a tal:attributes="href edit_url" href="">
        Edit this page
      </a>
    </p>
  </div>
</div>
<div id="footer">
  <div class="footer"
    >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
</div>
</body>
</html>

```

i The names available for our use in a template are always those that are present in the dictionary returned by the view callable. But our templates make use of a `request` object that none of our tutorial views return in their dictionary. This value appears as if “by magic”. However, `request` is one of several names that are available “by default” in a template when a template renderer is used. See **.pt* or **.txt: Chameleon Template Renderers* for more information about other names that are available by default in a template when a template is used as a renderer.

The edit.pt Template

The edit.pt template is used for adding and editing a Page. It is used by the add_page and edit_page view functions. It should display a page containing a form that POSTs back to the “save_url” argument supplied by the view. The form should have a “body” textarea field (the page data), and a submit button that has the name “form.submitted”. The textarea in the form should be filled with any existing page data when it is rendered.

Once we’re done with the edit.pt template, it will look a lot like the below:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
  <title>${page.__name__} - Pyramid tutorial wiki (based on
    TurboGears 20-Minute Wiki)</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <meta name="keywords" content="python web application" />
  <meta name="description" content="pyramid web application" />
  <link rel="shortcut icon"
        href="${request.static_url('tutorial:static/favicon.ico')}" />
  <link rel="stylesheet"
        href="${request.static_url('tutorial:static/pylons.css')}"
        type="text/css" media="screen" charset="utf-8" />
  <!--[if lte IE 6]>
  <link rel="stylesheet"
        href="${request.static_url('tutorial:static/ie6.css')}"
        type="text/css" media="screen" charset="utf-8" />
  <![endif]-->
</head>
<body>
  <div id="wrap">
    <div id="top-small">
      <div class="top-small align-center">
        <div>
          
        </div>
      </div>
    </div>
    <div id="middle">
      <div class="middle align-right">
        <div id="left" class="app-welcome align-left">
          Editing <b><span tal:replace="page.__name__">Page Name Goes
            Here</span></b><br/>

```

```

        You can return to the
        <a href="{request.application_url}">FrontPage</a>.<br/>
    </div>
    <div id="right" class="app-welcome align-right"></div>
</div>
<div id="bottom">
    <div class="bottom">
        <form action="{save_url}" method="post">
            <textarea name="body" tal:content="page.data" rows="10"
                cols="60"/><br/>
            <input type="submit" name="form.submitted" value="Save"/>
        </form>
    </div>
</div>
<div id="footer">
    <div class="footer">
        >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
    </div>
</body>
</html>

```

Static Assets

Our templates name a single static asset named `pylons.css`. We don't need to create this file within our package's `static` directory because it was provided at the time we created the project. This file is a little too long to replicate within the body of this guide, however it is available online.

This CSS file will be accessed via e.g. `http://localhost:6543/static/pylons.css` by virtue of the call to `add_static_view` directive we've made in the `__init__` file. Any number and type of static assets can be placed in this directory (or subdirectories) and are just referred to by URL or by using the convenience method `static_url` e.g. `request.static_url('{package}:static/foo.css')` within templates.

30.5.4 Testing the Views

We'll modify our `tests.py` file, adding tests for each view function we added above. As a result, we'll *delete* the `ViewTests` test in the file, and add four other test classes: `ViewWikiTests`, `ViewPageTests`, `AddPageTests`, and `EditPageTests`. These test the `view_wiki`, `view_page`, `add_page`, and `edit_page` views respectively.

Once we're done with the `tests.py` module, it will look a lot like the below:

```
1 import unittest
2
3 from pyramid import testing
4
5 class PageModelTests(unittest.TestCase):
6
7     def _getTargetClass(self):
8         from tutorial.models import Page
9         return Page
10
11     def _makeOne(self, data=u'some data'):
12         return self._getTargetClass()(data=data)
13
14     def test_constructor(self):
15         instance = self._makeOne()
16         self.assertEqual(instance.data, u'some data')
17
18 class WikiModelTests(unittest.TestCase):
19
20     def _getTargetClass(self):
21         from tutorial.models import Wiki
22         return Wiki
23
24     def _makeOne(self):
25         return self._getTargetClass()()
26
27     def test_it(self):
28         wiki = self._makeOne()
29         self.assertEqual(wiki.__parent__, None)
30         self.assertEqual(wiki.__name__, None)
31
32 class AppmakerTests(unittest.TestCase):
33     def _callFUT(self, zodb_root):
34         from tutorial.models import appmaker
35         return appmaker(zodb_root)
36
37     def test_it(self):
38         root = {}
39         self._callFUT(root)
40         self.assertEqual(root['app_root']['FrontPage'].data,
41                          'This is the front page')
42
43 class ViewWikiTests(unittest.TestCase):
44     def test_it(self):
45         from tutorial.views import view_wiki
46         context = testing.DummyResource()
```



```

47     request = testing.DummyRequest()
48     response = view_wiki(context, request)
49     self.assertEqual(response.location, 'http://example.com/FrontPage')
50
51 class ViewPageTests(unittest.TestCase):
52     def _callFUT(self, context, request):
53         from tutorial.views import view_page
54         return view_page(context, request)
55
56     def test_it(self):
57         wiki = testing.DummyResource()
58         wiki['IDoExist'] = testing.DummyResource()
59         context = testing.DummyResource(data='Hello CruelWorld IDoExist')
60         context.__parent__ = wiki
61         context.__name__ = 'thepage'
62         request = testing.DummyRequest()
63         info = self._callFUT(context, request)
64         self.assertEqual(info['page'], context)
65         self.assertEqual(
66             info['content'],
67             '<div class="document">\n'
68             '<p>Hello <a href="http://example.com/add_page/CruelWorld">'
69             'CruelWorld</a> '
70             '<a href="http://example.com/IDoExist/">'
71             'IDoExist</a>'
72             '</p>\n</div>\n')
73         self.assertEqual(info['edit_url'],
74             'http://example.com/thepage/edit_page')
75
76
77 class AddPageTests(unittest.TestCase):
78     def _callFUT(self, context, request):
79         from tutorial.views import add_page
80         return add_page(context, request)
81
82     def test_it_notsubmitted(self):
83         from pyramid.url import resource_url
84         context = testing.DummyResource()
85         request = testing.DummyRequest()
86         request.subpath = ['AnotherPage']
87         info = self._callFUT(context, request)
88         self.assertEqual(info['page'].data, '')
89         self.assertEqual(
90             info['save_url'],
91             resource_url(context, request, 'add_page', 'AnotherPage'))
92

```

```
93     def test_it_submitted(self):
94         context = testing.DummyResource()
95         request = testing.DummyRequest({'form.submitted': True,
96                                         'body': 'Hello yo!'})
97         request.subpath = ['AnotherPage']
98         self._callFUT(context, request)
99         page = context['AnotherPage']
100        self.assertEqual(page.data, 'Hello yo!')
101        self.assertEqual(page.__name__, 'AnotherPage')
102        self.assertEqual(page.__parent__, context)
103
104    class EditPageTests(unittest.TestCase):
105        def _callFUT(self, context, request):
106            from tutorial.views import edit_page
107            return edit_page(context, request)
108
109        def test_it_notsubmitted(self):
110            from pyramid.url import resource_url
111            context = testing.DummyResource()
112            request = testing.DummyRequest()
113            info = self._callFUT(context, request)
114            self.assertEqual(info['page'], context)
115            self.assertEqual(info['save_url'],
116                             resource_url(context, request, 'edit_page'))
117
118        def test_it_submitted(self):
119            context = testing.DummyResource()
120            request = testing.DummyRequest({'form.submitted': True,
121                                           'body': 'Hello yo!'})
122            response = self._callFUT(context, request)
123            self.assertEqual(response.location, 'http://example.com/')
124            self.assertEqual(context.data, 'Hello yo!')
```

30.5.5 Running the Tests

We can run these tests by using `setup.py test` in the same way we did in *Running the Tests*. Assuming our shell's current working directory is the “tutorial” distribution directory:

On UNIX:

```
$ ../bin/python setup.py test -q
```

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\python setup.py test -q
```

The expected result looks something like:

```
.....
-----
Ran 9 tests in 0.203s

OK
```

30.5.6 Viewing the Application in a Browser

Once we've completed our edits, we can finally examine our application in a browser. The views we'll try are as follows:

- Visiting `http://localhost:6543/` in a browser invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage Page` resource.
- Visiting `http://localhost:6543/FrontPage/` in a browser invokes the `view_page` view of the front page resource. This is because it's the *default view* (a view without a name) for `Page` resources.
- Visiting `http://localhost:6543/FrontPage/edit_page` in a browser invokes the `edit` view for the `FrontPage Page` resource.
- Visiting `http://localhost:6543/add_page/SomePageName` in a browser invokes the `add` view for a `Page`.
- To generate an error, visit `http://localhost:6543/add_page` which will generate an `IndexError` for the expression `request.subpath[0]`. You'll see an interactive traceback facility provided by *WebError*.

30.6 Adding Authorization

Our application currently allows anyone with access to the server to view, edit, and add pages to our wiki. For purposes of demonstration we'll change our application to allow people whom are members of a *group* named `group:editors` to add and edit wiki pages but we'll continue allowing anyone with access to the server to view pages. Pyramid provides facilities for *authorization* and *authentication*. We'll make use of both features to provide security to our application.

The source code for this tutorial stage can be browsed via <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki/src/authorization/>.

30.6.1 Configuring a pyramid Authentication Policy

For any Pyramid application to perform authorization, we need to add a `security.py` module and we'll need to change our *application registry* to add an *authentication policy* and a *authorization policy*.

Adding Authentication and Authorization Policies

We'll change our package's `__init__.py` file to enable an `AuthTktAuthenticationPolicy` and an `ACLAuthorizationPolicy` to enable declarative security checking. When you're done, your `__init__.py` will look like so:

```
1 from repoze.zodbconn.finder import PersistentApplicationFinder
2
3 from pyramid.config import Configurator
4 from pyramid.authentication import AuthTktAuthenticationPolicy
5 from pyramid.authorization import ACLAuthorizationPolicy
6
7 from tutorial.models import appmaker
8 from tutorial.security import groupfinder
9
10 def main(global_config, **settings):
11     """ This function returns a WSGI application.
12
13     It is usually called by the PasteDeploy framework during
14     'paster serve'.
15     """
16     authn_policy = AuthTktAuthenticationPolicy(secret='sosecret',
17                                             callback=groupfinder)
18     authz_policy = ACLAuthorizationPolicy()
19     zodb_uri = settings.get('zodb_uri')
20     if zodb_uri is None:
21         raise ValueError("No 'zodb_uri' in application configuration.")
22
23     finder = PersistentApplicationFinder(zodb_uri, appmaker)
24     def get_root(request):
25         return finder(request.environ)
26     config = Configurator(root_factory=get_root, settings=settings,
27                         authentication_policy=authn_policy,
28                         authorization_policy=authz_policy)
29     config.add_static_view('static', 'tutorial:static')
30     config.scan('tutorial')
31     return config.make_wsgi_app()
```

Note that the creation of an `AuthTktAuthenticationPolicy` requires two arguments: `secret` and `callback`. `secret` is a string representing an encryption key used by the “authentication ticket” machinery represented by this policy: it is required. The `callback` is a reference to a `groupfinder` function in the `tutorial` package’s `security.py` file. We haven’t added that module yet, but we’re about to.

Adding `security.py`

Add a `security.py` module within your package (in the same directory as `__init__.py`, `views.py`, etc) with the following content:

```
1 USERS = {'editor': 'editor',
2         'viewer': 'viewer'}
3 GROUPS = {'editor': ['group:editors']}
4
5 def groupfinder(userid, request):
6     if userid in USERS:
7         return GROUPS.get(userid, [])
```

The `groupfinder` function defined here is an authorization policy “callback”; it is a callable that accepts a `userid` and a `request`. If the `userid` exists in the set of users known by the system, the callback will return a sequence of group identifiers (or an empty sequence if the user isn’t a member of any groups). If the `userid` *does not* exist in the system, the callback will return `None`. In a production system this data will most often come from a database, but here we use “dummy” data to represent user and groups sources. Note that the `editor` user is a member of the `group:editors` group in our dummy group data (the `GROUPS` data structure).

Adding Login and Logout Views

We’ll add a `login` view which renders a login form and processes the post from the login form, checking credentials.

We’ll also add a `logout` view to our application and provide a link to it. This view will clear the credentials of the logged in user and redirect back to the front page.

We’ll add a different file (for presentation convenience) to add login and logout views. Add a file named `login.py` to your application (in the same directory as `views.py`) with the following content:

```
1 from pyramid.httpexceptions import HTTPFound
2
3 from pyramid.security import remember
4 from pyramid.security import forget
5 from pyramid.view import view_config
6 from pyramid.url import resource_url
7
8 from tutorial.security import USERS
9
10 @view_config(context='tutorial.models.Wiki', name='login',
11             renderer='templates/login.pt')
12 @view_config(context='pyramid.exceptions.Forbidden',
13             renderer='templates/login.pt')
14 def login(request):
15     login_url = resource_url(request.context, request, 'login')
16     referrer = request.url
17     if referrer == login_url:
18         referrer = '/' # never use the login form itself as came_from
19     came_from = request.params.get('came_from', referrer)
20     message = ''
21     login = ''
22     password = ''
23     if 'form.submitted' in request.params:
24         login = request.params['login']
25         password = request.params['password']
26         if USERS.get(login) == password:
27             headers = remember(request, login)
28             return HTTPFound(location = came_from,
29                             headers = headers)
30         message = 'Failed login'
31
32     return dict(
33         message = message,
34         url = request.application_url + '/login',
35         came_from = came_from,
36         login = login,
37         password = password,
38     )
39
40 @view_config(context='tutorial.models.Wiki', name='logout')
41 def logout(request):
42     headers = forget(request)
43     return HTTPFound(location = resource_url(request.context, request),
44                     headers = headers)
```

Note that the login view callable in the login.py file has *two* view configuration decorators. The

order of these decorators is unimportant. Each just adds a different *view configuration* for the `login` view callable.

The first view configuration decorator configures the `login` view callable so it will be invoked when someone visits `/login` (when the context is a `Wiki` and the view name is `login`). The second decorator (with context of `pyramid.exceptions.Forbidden`) specifies a *forbidden view*. This configures our `login` view to be presented to the user when Pyramid detects that a view invocation can not be authorized. Because we've configured a forbidden view, the `login` view callable will be invoked whenever one of our users tries to execute a view callable that they are not allowed to invoke as determined by the *authorization policy* in use. In our application, for example, this means that if a user has not logged in, and he tries to add or edit a Wiki page, he will be shown the login form. Before being allowed to continue on to the add or edit form, he will have to provide credentials that give him permission to add or edit via this login form.

Changing Existing Views

Then we need to change each of our `view_page`, `edit_page` and `add_page` views in `views.py` to pass a "logged in" parameter into its template. We'll add something like this to each view body:

```
1 from pyramid.security import authenticated_userid
2 logged_in = authenticated_userid(request)
```

We'll then change the return value of each view that has an associated `renderer` to pass the *resulting* `'logged_in'` value to the template. For example:

```
1 return dict(page = context,
2             content = content,
3             logged_in = logged_in,
4             edit_url = edit_url)
```

Adding the `login.pt` Template

Add a `login.pt` template to your templates directory. It's referred to within the login view we just added to `login.py`.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
<title>Login - Pyramid tutorial wiki (based on TurboGears
20-Minute Wiki)</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<meta name="keywords" content="python web application" />
<meta name="description" content="pyramid web application" />
<link rel="shortcut icon"
href="{request.static_url('tutorial:static/favicon.ico')}" />
<link rel="stylesheet"
href="{request.static_url('tutorial:static/pylons.css')}"
type="text/css" media="screen" charset="utf-8" />
<!--[if lte IE 6]>
<link rel="stylesheet"
href="{request.static_url('tutorial:static/ie6.css')}"
type="text/css" media="screen" charset="utf-8" />
<![endif]-->
</head>
<body>
<div id="wrap">
<div id="top-small">
<div class="top-small align-center">
<div>

</div>
</div>
</div>
<div id="middle">
<div class="middle align-right">
<div id="left" class="app-welcome align-left">
<b>Login</b><br/>
<span tal:replace="message"/>
</div>
<div id="right" class="app-welcome align-right"></div>
</div>
</div>
<div id="bottom">
<div class="bottom">
<form action="{url}" method="post">
<input type="hidden" name="came_from" value="{came_from}"/>
<input type="text" name="login" value="{login}"/><br/>
<input type="password" name="password"

```



```

        value="{password}"/><br/>
        <input type="submit" name="form.submitted" value="Log In"/>
    </form>
</div>
</div>
</div>
<div id="footer">
    <div class="footer"
        >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
</div>
</body>
</html>

```

Change `view.pt` and `edit.pt`

We'll also need to change our `edit.pt` and `view.pt` templates to display a "Logout" link if someone is logged in. This link will invoke the `logout` view.

To do so we'll add this to both templates within the `<div id="right" class="app-welcome align-right">` `div`:

```

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

```

30.6.2 Giving Our Root Resource an ACL

We need to give our root resource object an *ACL*. This ACL will be sufficient to provide enough information to the Pyramid security machinery to challenge a user who doesn't have appropriate credentials when he attempts to invoke the `add_page` or `edit_page` views.

We need to perform some imports at module scope in our `models.py` file:

```

1 from pyramid.security import Allow
2 from pyramid.security import Everyone

```

Our root resource object is a `Wiki` instance. We'll add the following line at class scope to our `Wiki` class:

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

It's only happenstance that we're assigning this ACL at class scope. An ACL can be attached to an object *instance* too; this is how “row level security” can be achieved in Pyramid applications. We actually only need *one* ACL for the entire system, however, because our security requirements are simple, so this feature is not demonstrated.

Our resulting `models.py` file will now look like so:

```
1 from persistent import Persistent  
2 from persistent.mapping import PersistentMapping  
3  
4 from pyramid.security import Allow  
5 from pyramid.security import Everyone  
6  
7 class Wiki(PersistentMapping):  
8     __name__ = None  
9     __parent__ = None  
10    __acl__ = [ (Allow, Everyone, 'view'),  
11               (Allow, 'group:editors', 'edit') ]  
12  
13 class Page(Persistent):  
14     def __init__(self, data):  
15         self.data = data  
16  
17 def appmaker(zodb_root):  
18     if not 'app_root' in zodb_root:  
19         app_root = Wiki()  
20         frontpage = Page('This is the front page')  
21         app_root['FrontPage'] = frontpage  
22         frontpage.__name__ = 'FrontPage'  
23         frontpage.__parent__ = app_root  
24         zodb_root['app_root'] = app_root  
25         import transaction  
26         transaction.commit()  
27     return zodb_root['app_root']
```

30.6.3 Adding permission Declarations to our `view_config` Decorators

To protect each of our views with a particular permission, we need to pass a `permission` argument to each of our `pyramid.view.view_config` decorators. To do so, within `views.py`:

- We add `permission='view'` to the decorator attached to the `view_wiki` view function. This makes the assertion that only users who possess the `view` permission against the context resource at the time of the request may invoke this view. We've granted `pyramid.security.Everyone` the `view` permission at the root model via its ACL, so everyone will be able to invoke the `view_wiki` view.
- We add `permission='view'` to the decorator attached to the `view_page` view function. This makes the assertion that only users who possess the effective `view` permission against the context resource at the time of the request may invoke this view. We've granted `pyramid.security.Everyone` the `view` permission at the root model via its ACL, so everyone will be able to invoke the `view_page` view.
- We add `permission='edit'` to the decorator attached to the `add_page` view function. This makes the assertion that only users who possess the effective `edit` permission against the context resource at the time of the request may invoke this view. We've granted the `group:editors` principal the `edit` permission at the root model via its ACL, so only the a user whom is a member of the group named `group:editors` will able to invoke the `add_page` view. We've likewise given the `editor` user membership to this group via the `security.py` file by mapping him to the `group:editors` group in the `GROUPS` data structure (`GROUPS = {'editor': ['group:editors']}`); the `groupfinder` function consults the `GROUPS` data structure. This means that the `editor` user can add pages.
- We add `permission='edit'` to the decorator attached to the `edit_page` view function. This makes the assertion that only users who possess the effective `edit` permission against the context resource at the time of the request may invoke this view. We've granted the `group:editors` principal the `edit` permission at the root model via its ACL, so only the a user whom is a member of the group named `group:editors` will able to invoke the `edit_page` view. We've likewise given the `editor` user membership to this group via the `security.py` file by mapping him to the `group:editors` group in the `GROUPS` data structure (`GROUPS = {'editor': ['group:editors']}`); the `groupfinder` function consults the `GROUPS` data structure. This means that the `editor` user can edit pages.

30.6.4 Viewing the Application in a Browser

We can finally examine our application in a browser. The views we'll try are as follows:

- Visiting `http://localhost:6543/` in a browser invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` page resource. It is executable by any user.
- Visiting `http://localhost:6543/FrontPage/` in a browser invokes the `view_page` view of the `FrontPage` Page resource. This is because it's the *default view* (a view without a name) for Page resources. It is executable by any user.

- Visiting `http://localhost:6543/FrontPage/edit_page` in a browser invokes the edit view for the `FrontPage Page` resource. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, a login form will be displayed. Supplying the credentials with the username `editor`, password `editor` will show the edit page form being displayed.
- Visiting `http://localhost:6543/add_page/SomePageName` in a browser invokes the add view for a page. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, a login form will be displayed. Supplying the credentials with the username `editor`, password `editor` will show the edit page form being displayed.

30.6.5 Seeing Our Changes To `views.py` and our Templates

Our `views.py` module will look something like this when we're done:

```
1 from docutils.core import publish_parts
2 import re
3
4 from pyramid.httpexceptions import HTTPFound
5 from pyramid.url import resource_url
6 from pyramid.view import view_config
7 from pyramid.security import authenticated_userid
8
9 from tutorial.models import Page
10
11 # regular expression used to find WikiWords
12 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+)")
13
14 @view_config(context='tutorial.models.Wiki', permission='view')
15 def view_wiki(context, request):
16     return HTTPFound(location=resource_url(context, request, 'FrontPage'))
17
18 @view_config(context='tutorial.models.Page',
19             renderer='templates/view.pt', permission='view')
20 def view_page(context, request):
21     wiki = context.__parent__
22
23     def check(match):
24         word = match.group(1)
25         if word in wiki:
26             page = wiki[word]
27             view_url = resource_url(page, request)
28             return '<a href="%s">%s</a>' % (view_url, word)
29     else:
30         add_url = request.application_url + '/add_page/' + word
```

```

31         return '<a href="%s">%s</a>' % (add_url, word)
32
33     content = publish_parts(context.data, writer_name='html')['html_body']
34     content = wikiwords.sub(check, content)
35     edit_url = resource_url(context, request, 'edit_page')
36
37     logged_in = authenticated_userid(request)
38
39     return dict(page = context, content = content, edit_url = edit_url,
40               logged_in = logged_in)
41
42 @view_config(name='add_page', context='tutorial.models.Wiki',
43             renderer='templates/edit.pt',
44             permission='edit')
45 def add_page(context, request):
46     name = request.subpath[0]
47     if 'form.submitted' in request.params:
48         body = request.params['body']
49         page = Page(body)
50         page.__name__ = name
51         page.__parent__ = context
52         context[name] = page
53         return HTTPFound(location = resource_url(page, request))
54     save_url = resource_url(context, request, 'add_page', name)
55     page = Page('')
56     page.__name__ = name
57     page.__parent__ = context
58
59     logged_in = authenticated_userid(request)
60
61     return dict(page = page, save_url = save_url, logged_in = logged_in)
62
63 @view_config(name='edit_page', context='tutorial.models.Page',
64             renderer='templates/edit.pt',
65             permission='edit')
66 def edit_page(context, request):
67     if 'form.submitted' in request.params:
68         context.data = request.params['body']
69         return HTTPFound(location = resource_url(context, request))
70
71     logged_in = authenticated_userid(request)
72
73     return dict(page = context,
74               save_url = resource_url(context, request, 'edit_page'),
75               logged_in = logged_in)

```

Our edit .pt template will look something like this when we're done:

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
4     xmlns:tal="http://xml.zope.org/namespaces/tal">
5  <head>
6     <title>${page.__name__} - Pyramid tutorial wiki (based on
7         TurboGears 20-Minute Wiki)</title>
8     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
9     <meta name="keywords" content="python web application" />
10    <meta name="description" content="pyramid web application" />
11    <link rel="shortcut icon"
12         href="${request.static_url('tutorial:static/favicon.ico')}" />
13    <link rel="stylesheet"
14         href="${request.static_url('tutorial:static/pylons.css')}"
15         type="text/css" media="screen" charset="utf-8" />
16    <!--[if lte IE 6]>
17    <link rel="stylesheet"
18         href="${request.static_url('tutorial:static/ie6.css')}"
19         type="text/css" media="screen" charset="utf-8" />
20    <![endif]-->
21 </head>
22 <body>
23     <div id="wrap">
24         <div id="top-small">
25             <div class="top-small align-center">
26                 <div>
27                     
29                 </div>
30             </div>
31         </div>
32         <div id="middle">
33             <div class="middle align-right">
34                 <div id="left" class="app-welcome align-left">
35                     Editing <b><span tal:replace="page.__name__">Page Name
36                         Goes Here</span></b><br/>
37                     You can return to the
38                     <a href="${request.application_url}">FrontPage</a>. <br/>
39                 </div>
40                 <div id="right" class="app-welcome align-right">
41                     <span tal:condition="logged_in">
42                         <a href="${request.application_url}/logout">Logout</a>
43                     </span>
44                 </div>
45             </div>

```

```

46     </div>
47     <div id="bottom">
48         <div class="bottom">
49             <form action="{save_url}" method="post">
50                 <textarea name="body" tal:content="page.data" rows="10"
51                     cols="60"/><br/>
52                 <input type="submit" name="form.submitted" value="Save"/>
53             </form>
54         </div>
55     </div>
56 </div>
57 <div id="footer">
58     <div class="footer"
59         >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
60 </div>
61 </body>
62 </html>

```

Our view.pt template will look something like this when we're done:

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
4     xmlns:tal="http://xml.zope.org/namespaces/tal">
5  <head>
6     <title>${page.__name__} - Pyramid tutorial wiki (based on
7         TurboGears 20-Minute Wiki)</title>
8     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
9     <meta name="keywords" content="python web application" />
10    <meta name="description" content="pyramid web application" />
11    <link rel="shortcut icon"
12        href="{request.static_url('tutorial:static/favicon.ico')}" />
13    <link rel="stylesheet"
14        href="{request.static_url('tutorial:static/pylons.css')}"
15        type="text/css" media="screen" charset="utf-8" />
16    <!--[if lte IE 6]>
17    <link rel="stylesheet"
18        href="{request.static_url('tutorial:static/ie6.css')}"
19        type="text/css" media="screen" charset="utf-8" />
20    <![endif]-->
21 </head>
22 <body>
23     <div id="wrap">
24         <div id="top-small">
25             <div class="top-small align-center">
26                 <div>

```

```
27     
29     </div>
30 </div>
31 </div>
32 <div id="middle">
33     <div class="middle align-right">
34         <div id="left" class="app-welcome align-left">
35             Viewing <b><span tal:replace="page.__name__">Page Name
36             Goes Here</span></b><br/>
37             You can return to the
38             <a href="{request.application_url}">FrontPage</a>. <br/>
39         </div>
40         <div id="right" class="app-welcome align-right">
41             <span tal:condition="logged_in">
42                 <a href="{request.application_url}/logout">Logout</a>
43             </span>
44         </div>
45     </div>
46 </div>
47 <div id="bottom">
48     <div class="bottom">
49         <div tal:replace="structure content">
50             Page text goes here.
51         </div>
52         <p>
53             <a tal:attributes="href edit_url" href="">
54                 Edit this page
55             </a>
56         </p>
57     </div>
58 </div>
59 <div id="footer">
60     <div class="footer"
61         >&copy; Copyright 2008–2011, Agendaless Consulting.</div>
62 </div>
63 </body>
64 </html>
```

30.6.6 Revisiting the Application

When we revisit the application in a browser, and log in (as a result of hitting an edit or add page and submitting the login form with the `editor` credentials), we'll see a Logout link in the upper right hand corner. When we click it, we're logged out, and redirected back to the front page.

30.7 Distributing Your Application

Once your application works properly, you can create a “tarball” from it by using the `setup.py sdist` command. The following commands assume your current working directory is the `tutorial` package we’ve created and that the parent directory of the `tutorial` package is a `virtualenv` representing a Pyramid environment.

On UNIX:

```
$ ../bin/python setup.py sdist
```

On Windows:

```
c:\pyramidtut> ..\Scripts\python setup.py sdist
```

The output of such a command will be something like:

```
running sdist
# .. more output ..
creating dist
tar -cf dist/tutorial-0.1.tar tutorial-0.1
gzip -f9 dist/tutorial-0.1.tar
removing 'tutorial-0.1' (and everything under it)
```

Note that this command creates a tarball in the “dist” subdirectory named `tutorial-0.1.tar.gz`. You can send this file to your friends to show them your cool new application. They should be able to install it by pointing the `easy_install` command directly at it. Or you can upload it to PyPI and share it with the rest of the world, where it can be downloaded via `easy_install` remotely like any other package people download from PyPI.

SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

This tutorial introduces a *SQLAlchemy* and *url dispatch* -based Pyramid application to a developer familiar with Python, and will be most familiar to developers who have used the *Pylons 1.X* web framework. When the tutorial is finished, the developer will have created a basic Wiki application with authentication.

For cut and paste purposes, the source code for all stages of this tutorial can be browsed at <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki2/>.

31.1 Background

This tutorial presents a Pyramid application that uses technologies which will be familiar to someone with *Pylons* experience. It uses *SQLAlchemy* as a persistence mechanism and *url dispatch* to map URLs to code. It can also be followed by people without any prior Python web framework experience.

To code along with this tutorial, the developer will need a UNIX machine with development tools (Mac OS X with XCode, any Linux or BSD variant, etc) *or* he will need a Windows system of any kind.

This tutorial is targeted at Pyramid version 1.0.

Have fun!

31.2 Installation

For the most part, the installation process for this tutorial duplicates the steps described in *Installing Pyramid* and *Creating a Pyramid Project*, however it also explains how to install additional libraries for tutorial purposes.

31.2.1 Preparation

Please take the following steps to prepare for the tutorial. The steps are slightly different depending on whether you're using UNIX or Windows.

Preparation, UNIX

1. Install SQLite3 and its development packages if you don't already have them installed. Usually this is via your system's package manager. For example, on a Debian Linux system, do `sudo apt-get install libsqlite3-dev`.
2. If you don't already have a Python 2.6 interpreter installed on your system, obtain, install, or find Python 2.6 for your system.
3. Install the latest *setuptools* into the Python you obtained/installed/found in the step above: download `ez_setup.py` and run it using the `python` interpreter of your Python 2.6 installation:

```
$ /path/to/my/Python-2.6/bin/python ez_setup.py
```

4. Use that Python's `bin/easy_install` to install *virtualenv*:

```
$ /path/to/my/Python-2.6/bin/easy_install virtualenv
```

5. Use that Python's *virtualenv* to make a workspace:

```
$ path/to/my/Python-2.6/bin/virtualenv --no-site-packages pyramidtut
```

6. Switch to the `pyramidtut` directory:

```
$ cd pyramidtut
```

7. (Optional) Consider using `source bin/activate` to make your shell environment wired to use the `virtualenv`.
8. Use `easy_install` to get Pyramid and its direct dependencies installed:

```
$ bin/easy_install pyramid
```

9. Use `easy_install` to install various packages from PyPI.

```
$ bin/easy_install docutils nose coverage zope.sqlalchemy \
    SQLAlchemy repoze.tm2
```

Preparation, Windows

1. Install, or find Python 2.6.6 for your system.
2. Install the latest `setuptools` into the Python you obtained/installed/found in the step above: download `ez_setup.py` and run it using the `python` interpreter of your Python 2.6 installation using a command prompt:

```
c:\> c:\Python26\python ez_setup.py
```

3. Use that Python's `bin/easy_install` to install `virtualenv`:

```
c:\> c:\Python26\Scripts\easy_install virtualenv
```

4. Use that Python's `virtualenv` to make a workspace:

```
c:\> c:\Python26\Scripts\virtualenv --no-site-packages pyramidtut
```

5. Switch to the `pyramidtut` directory:

```
c:\> cd pyramidtut
```

6. (Optional) Consider using `bin\activate.bat` to make your shell environment wired to use the `virtualenv`.
7. Use `easy_install` to get Pyramid and its direct dependencies installed:

31. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
c:\pyramidtut> Scripts\easy_install pyramid
```

8. Use `easy_install` to install various packages from PyPI.

```
c:\pyramidtut> Scripts\easy_install -i docutils \  
nose coverage zope.sqlalchemy SQLAlchemy repoze.tm2
```

31.2.2 Making a Project

Your next step is to create a project. Pyramid supplies a variety of templates to generate sample projects. We will use the `pyramid_routesalchemy` template, which generates an application that uses *SQLAlchemy* and *URL dispatch*.

The below instructions assume your current working directory is the “virtualenv” named “pyramidtut”.

On UNIX:

```
$ bin/paster create -t pyramid_routesalchemy tutorial
```

On Windows:

```
c:\pyramidtut> Scripts\paster create -t pyramid_routesalchemy tutorial
```

i If you are using Windows, the `pyramid_routesalchemy` Paster template may not deal gracefully with installation into a location that contains spaces in the path. If you experience startup problems, try putting both the virtualenv and the project into directories that do not contain spaces in their paths.

31.2.3 Installing the Project in “Development Mode”

In order to do development on the project easily, you must “register” the project as a development egg in your workspace using the `setup.py develop` command. In order to do so, `cd` to the “tutorial” directory you created in *Making a Project*, and run the “`setup.py develop`” command using virtualenv Python interpreter.

On UNIX:

```
$ cd tutorial
$ ../bin/python setup.py develop
```

On Windows:

```
c:\pyramidtut> cd tutorial
c:\pyramidtut\tutorial> ..\Scripts\python setup.py develop
```

31.2.4 Running the Tests

After you've installed the project in development mode, you may run the tests for the project.

On UNIX:

```
$ ../bin/python setup.py test -q
```

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\python setup.py test -q
```

31.2.5 Starting the Application

Start the application.

On UNIX:

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

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\paster serve development.ini --reload
```

31.2.6 Exposing Test Coverage Information

You can run the `nosetests` command to see test coverage information. This runs the tests in the same way that `setup.py test` does but provides additional “coverage” information, exposing which lines of your project are “covered” (or not covered) by the tests.

To get this functionality working, we’ll need to install a couple of other packages into our `virtualenv`: `nose` and `coverage`:

On UNIX:

```
$ ../bin/easy_install nose coverage
```

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\easy_install nose coverage
```

Once `nose` and `coverage` are installed, we can actually run the coverage tests.

On UNIX:

```
$ ../bin/nosetests --cover-package=tutorial --cover-erase --with-coverage
```

On Windows:

```
c:\pyramidtut\tutorial> ..\Scripts\nosetests --cover-package=tutorial \  
--cover-erase --with-coverage
```

Looks like our package’s `models` module doesn’t quite have 100% test coverage.


31.2.7 Visit the Application in a Browser

In a browser, visit `http://localhost:6543/`. You will see the generated application’s default page.

31.2.8 Decisions the `pyramid_routesalchemy` Template Has Made For You

Creating a project using the `pyramid_routesalchemy` template makes the following assumptions:

- you are willing to use *SQLAlchemy* as a database access tool
- you are willing to use *url dispatch* to map URLs to code.
- you want to configure your application *imperatively* (no *declarative configuration* such as ZCML).

 Pyramid supports any persistent storage mechanism (e.g. object database or filesystem files, etc). It also supports an additional mechanism to map URLs to code (*traversal*). However, for the purposes of this tutorial, we'll only be using *url dispatch* and *SQLAlchemy*.

31.3 Basic Layout

The starter files generated by the `pyramid_routesalchemy` template are basic, but they provide a good orientation for the high-level patterns common to most *url dispatch* -based Pyramid projects.

The source code for this tutorial stage can be browsed at <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki2/src/basiclayout/>.

31.3.1 App Startup with `__init__.py`

A directory on disk can be turned into a Python *package* by containing an `__init__.py` file. Even if empty, this marks a directory as a Python package. We use `__init__.py` both as a package marker and to contain configuration code.

The generated `development.ini` file is read by `paster` which looks for the application module in the `use` variable of the `app:tutorial` section. The *entry point* is defined in the `Setuptools` configuration of this module, specifically in the `setup.py` file. For this tutorial, the *entry point* is defined as `tutorial:main` and points to the following `main` function:

```
1 from pyramid.config import Configurator
2 from sqlalchemy import engine_from_config
3
4 from tutorial.models import initialize_sql
5
6 def main(global_config, **settings):
7     """ This function returns a Pyramid WSGI application.
8         """
9     engine = engine_from_config(settings, 'sqlalchemy.')
10    initialize_sql(engine)
11    config = Configurator(settings=settings)
12    config.add_static_view('static', 'tutorial:static')
13    config.add_route('home', '/', view='tutorial.views.my_view',
14                    view_renderer='templates/mytemplate.pt')
15    return config.make_wsgi_app()
```

1. *Lines 1-4.* Imports to support later code.
2. *Line 9.* Create a SQLAlchemy database engine from the `sqlalchemy.` prefixed settings in the `development.ini` file's `[app:tutorial]` section. This will be a URI (something like `sqlite://`).
3. *Line 10.* We initialize our SQL database using SQLAlchemy, passing it the engine
4. *Line 11.* We construct a *Configurator*. `settings` is passed as a keyword argument with the dictionary values passed by PasteDeploy as the `settings` argument. This will be a dictionary of settings parsed by PasteDeploy, which contains deployment-related values such as `reload_templates`, `db_string`, etc.
5. *Line 12.* We call `pyramid.config.Configurator.add_static_view()` with the arguments `static` (the name), and `tutorial:static` (the path). This registers a static resource view which will match any URL that starts with `/static/`. This will serve up static resources for us from within the `static` directory of our `tutorial` package, in this case, via `http://localhost:6543/static/` and below. With this declaration, we're saying that any URL that starts with `/static` should go to the static view; any remainder of its path (e.g. the `/foo` in `/static/foo`) will be used to compose a path to a static file resource, such as a CSS file.
6. *Lines 13-14.* Register a *route configuration* via the `pyramid.config.Configurator.add_route()` method that will be used when the URL is `/`. Since this route has a pattern equalling `/` it is the "default" route. The argument named `view` with the value `tutorial.views.my_view` is the dotted name to a *function* we write (generated by the `pyramid_routesalchemy` template) that is given a `request` object and which returns a response or a dictionary. You

will use `pyramid.config.Configurator.add_route()` statements in a *URL dispatch* based application to map URLs to code. This route also names a `view_renderer`, which is a template which lives in the `templates` subdirectory of the package. When the `tutorial.views.my_view` view returns a dictionary, a *renderer* will use this template to create a response.

7. *Line 15.* We use the `pyramid.config.Configurator.make_wsgi_app()` method to return a *WSGI* application.

31.3.2 Content Models with `models.py`

In a SQLAlchemy-based application, a *model* object is an object composed by querying the SQL database which backs an application. SQLAlchemy is an “object relational mapper” (an ORM). The `models.py` file is where the `pyramid_routesalchemy` Paster template put the classes that implement our models.

Here is the source for `models.py`:

```

1 import transaction
2
3 from sqlalchemy import Column
4 from sqlalchemy import Integer
5 from sqlalchemy import Unicode
6
7 from sqlalchemy.exc import IntegrityError
8 from sqlalchemy.ext.declarative import declarative_base
9
10 from sqlalchemy.orm import scoped_session
11 from sqlalchemy.orm import sessionmaker
12
13 from zope.sqlalchemy import ZopeTransactionExtension
14
15 DBSession = scoped_session(sessionmaker(
16                             extension=ZopeTransactionExtension()))
17 Base = declarative_base()
18
19 class MyModel(Base):
20     __tablename__ = 'models'
21     id = Column(Integer, primary_key=True)
22     name = Column(Unicode(255), unique=True)
23     value = Column(Integer)
24
25     def __init__(self, name, value):

```

```
26         self.name = name
27         self.value = value
28
29     def populate():
30         session = DBSession()
31         model = MyModel(name=u'root', value=55)
32         session.add(model)
33         session.flush()
34         transaction.commit()
35
36     def initialize_sql(engine):
37         DBSession.configure(bind=engine)
38         Base.metadata.bind = engine
39         Base.metadata.create_all(engine)
40         try:
41             populate()
42         except IntegrityError:
43             pass
```

1. *Lines 1-13.* Imports to support later code.
2. *Line 15.* We set up a SQLAlchemy “DBSession” object here. We specify that we’d like to use the “ZopeTransactionExtension”. This extension is an extension which allows us to use a *transaction manager* instead of controlling commits and aborts to database operations by hand.
3. *Line 16.* We create a declarative Base object to use as a base class for our model.
4. *Lines 18-26.* A model class named MyModel. It has an `__init__` that takes a two arguments (name, and value). It stores these values as `self.name` and `self.value` within the `__init__` function itself. The MyModel class also has a `__tablename__` attribute. This informs SQLAlchemy which table to use to store the data representing instances of this class.
5. *Lines 28-33.* A function named `populate` which adds a single model instance into our SQL storage and commits a transaction.
6. *Lines 35-42.* A function named `initialize_sql` which receives a SQL database engine and binds it to our SQLAlchemy DBSession object. It also calls the `populate` function, to do initial database population.

31.4 Defining the Domain Model

The first change we’ll make to our stock paster-generated application will be to define a *domain model* constructor representing a wiki page. We’ll do this inside our `models.py` file.

The source code for this tutorial stage can be browsed at <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki2/src/models/>.

31.4.1 Making Edits to `models.py`

i There is nothing automatically special about the filename `models.py`. A project may have many models throughout its codebase in arbitrarily-named files. Files implementing models often have `model` in their filenames (or they may live in a Python subpackage of your application package named `models`), but this is only by convention.

The first thing we want to do is remove the stock `MyModel` class from the generated `models.py` file. The `MyModel` class is only a sample and we're not going to use it.

Then, we'll add a `Page` class. Because this is a SQLAlchemy application, this class should inherit from an instance of `sqlalchemy.ext.declarative.declarative_base`. Declarative SQLAlchemy models are easier to use than directly-mapped ones.

Our `Page` class will have a class level attribute `__tablename__` which equals the string `pages`. This means that SQLAlchemy will store our wiki data in a SQL table named `pages`. Our `Page` class will also have class-level attributes named `id`, `pagename` and `data` (all instances of `sqlalchemy.Column`). These will map to columns in the `pages` table. The `id` attribute will be the primary key in the table. The `name` attribute will be a text attribute, each value of which needs to be unique within the column. The `data` attribute is a text attribute that will hold the body of each page.

We'll also remove our `populate` function. We'll inline the `populate` step into `initialize_sql`, changing our `initialize_sql` function to add a `FrontPage` object to our database at startup time. We're also going to use slightly different binding syntax. It will otherwise largely be the same as the `initialize_sql` in the paster-generated `models.py`.

Our `DBSession` assignment stays the same as the original generated `models.py`.

31.4.2 Looking at the Result of Our Edits to `models.py`

The result of all of our edits to `models.py` will end up looking something like this:

```
1 import transaction
2
3 from sqlalchemy import Column
4 from sqlalchemy import Integer
5 from sqlalchemy import Text
6
7 from sqlalchemy.exc import IntegrityError
```

```
8 from sqlalchemy.ext.declarative import declarative_base
9
10 from sqlalchemy.orm import scoped_session
11 from sqlalchemy.orm import sessionmaker
12
13 from zope.sqlalchemy import ZopeTransactionExtension
14
15 DBSession = scoped_session(sessionmaker(
16                             extension=ZopeTransactionExtension()))
17 Base = declarative_base()
18
19 class Page(Base):
20     """ The SQLAlchemy declarative model class for a Page object. """
21     __tablename__ = 'pages'
22     id = Column(Integer, primary_key=True)
23     name = Column(Text, unique=True)
24     data = Column(Text)
25
26     def __init__(self, name, data):
27         self.name = name
28         self.data = data
29
30 def initialize_sql(engine):
31     DBSession.configure(bind=engine)
32     Base.metadata.bind = engine
33     Base.metadata.create_all(engine)
34     try:
35         session = DBSession()
36         page = Page('FrontPage', 'initial data')
37         session.add(page)
38         transaction.commit()
39     except IntegrityError:
40         # already created
41         pass
```

31.4.3 Viewing the Application in a Browser


We can't. At this point, our system is in a “non-runnable” state; we'll need to change view-related files in the next chapter to be able to start the application successfully. If you try to start the application, you'll wind up with a Python traceback on your console that ends with this exception:

```
ImportError: cannot import name MyModel
```

This will also happen if you attempt to run the tests.

31.5 Defining Views

A *view callable* in a *url dispatch*-based Pyramid application is typically a simple Python function that accepts a single parameter named *request*. A view callable is assumed to return a *response* object.

 A Pyramid view can also be defined as callable which accepts *two* arguments: a *context* and a *request*. You’ll see this two-argument pattern used in other Pyramid tutorials and applications. Either calling convention will work in any Pyramid application; the calling conventions can be used interchangeably as necessary. In *url dispatch* based applications, however, the context object is rarely used in the view body itself, so within this tutorial we define views as callables that accept only a request to avoid the visual “noise”. If you do need the `context` within a view function that only takes the request as a single argument, you can obtain it via `request.context`.

The request passed to every view that is called as the result of a route match has an attribute named `matchdict` that contains the elements placed into the URL by the pattern of a route statement. For instance, if a call to `pyramid.config.Configurator.add_route()` in `__init__.py` had the pattern `{one}/{two}`, and the URL at `http://example.com/foo/bar` was invoked, matching this pattern, the `matchdict` dictionary attached to the request passed to the view would have a `one` key with the value `foo` and a `two` key with the value `bar`.

The source code for this tutorial stage can be browsed at <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki2/src/views/>.

31.5.1 Declaring Dependencies in Our `setup.py` File

The view code in our application will depend on a package which is not a dependency of the original “tutorial” application. The original “tutorial” application was generated by the `paster create` command; it doesn’t know about our custom application requirements. We need to add a dependency on the `docutils` package to our tutorial package’s `setup.py` file by assigning this dependency to the `install_requires` parameter in the `setup` function.

Our resulting `setup.py` should look like so:

```

1 import os
2 import sys
3
4 from setuptools import setup, find_packages
5
6 here = os.path.abspath(os.path.dirname(__file__))

```

31. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
7 README = open(os.path.join(here, 'README.txt')).read()
8 CHANGES = open(os.path.join(here, 'CHANGES.txt')).read()
9
10 requires = [
11     'pyramid',
12     'SQLAlchemy',
13     'transaction',
14     'repoze.tm2>=1.0b1', # default_commit_veto
15     'zope.sqlalchemy',
16     'WebError',
17     'docutils',
18 ]
19
20 if sys.version_info[:3] < (2,5,0):
21     requires.append('pysqlite')
22
23 setup(name='tutorial',
24       version='0.0',
25       description='tutorial',
26       long_description=README + '\n\n' + CHANGES,
27       classifiers=[
28         "Programming Language :: Python",
29         "Framework :: Pylons",
30         "Topic :: Internet :: WWW/HTTP",
31         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
32     ],
33       author='',
34       author_email='',
35       url='',
36       keywords='web wsgi bfg pylons pyramid',
37       packages=find_packages(),
38       include_package_data=True,
39       zip_safe=False,
40       test_suite='tutorial',
41       install_requires = requires,
42       entry_points = """\
43 [paste.app_factory]
44 main = tutorial:main
45 """ ,
46       paster_plugins=['pyramid'],
47 )
```



After these new dependencies are added, you will need to rerun `python setup.py develop` inside the root of the tutorial package to obtain and register the newly added dependency package.

31.5.2 Adding View Functions

We'll get rid of our `my_view` view function in our `views.py` file. It's only an example and isn't relevant to our application.

Then we're going to add four *view callable* functions to our `views.py` module. One view callable (named `view_wiki`) will display the wiki itself (it will answer on the root URL), another named `view_page` will display an individual page, another named `add_page` will allow a page to be added, and a final view callable named `edit_page` will allow a page to be edited. We'll describe each one briefly and show the resulting `views.py` file afterward.



There is nothing special about the filename `views.py`. A project may have many view callables throughout its codebase in arbitrarily-named files. Files implementing view callables often have `view` in their filenames (or may live in a Python subpackage of your application package named `views`), but this is only by convention.

The `view_wiki` view function

The `view_wiki` function will respond as the *default view* of a `Wiki` model object. It always redirects to a URL which represents the path to our “FrontPage”. It returns an instance of the `pyramid.httpexceptions.HTTPFound` class (instances of which implement the `WebOb response` interface), It will use the `pyramid.url.route_url()` API to construct a URL to the `FrontPage` page (e.g. `http://localhost:6543/FrontPage`), and will use it as the “location” of the `HTTPFound` response, forming an HTTP redirect.

The `view_page` view function

The `view_page` function will respond as the *default view* of a `Page` object. The `view_page` function renders the *ReStructuredText* body of a page (stored as the `data` attribute of a `Page` object) as HTML. Then it substitutes an HTML anchor for each *WikiWord* reference in the rendered HTML using a compiled regular expression.

The curried function named `check` is used as the first argument to `wikiwords.sub`, indicating that it should be called to provide a value for each `WikiWord` match found in the content. If the wiki already contains a page with the matched `WikiWord` name, the `check` function generates a view link to be used

as the substitution value and returns it. If the wiki does not already contain a page with with the matched WikiWord name, the function generates an “add” link as the substitution value and returns it.

As a result, the `content` variable is now a fully formed bit of HTML containing various view and add links for WikiWords based on the content of our current page object.

We then generate an edit URL (because it’s easier to do here than in the template), and we return a dictionary with a number of arguments. The fact that this view returns a dictionary (as opposed to a *response* object) is a cue to Pyramid that it should try to use a *renderer* associated with the view configuration to render a template. In our case, the template which will be rendered will be the `templates/view.pt` template, as per the configuration put into effect in `__init__.py`.

The `add_page` view function

The `add_page` function will be invoked when a user clicks on a *WikiWord* which isn’t yet represented as a page in the system. The `check` function within the `view_page` view generates URLs to this view. It also acts as a handler for the form that is generated when we want to add a page object. The `matchdict` attribute of the request passed to the `add_page` view will have the values we need to construct URLs and find model objects.

The `matchdict` will have a `pagename` key that matches the name of the page we’d like to add. If our add view is invoked via, e.g. `http://localhost:6543/add_page/SomeName`, the `pagename` value in the `matchdict` will be `SomeName`.

If the view execution is *not* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `False`), the view callable renders a template. To do so, it generates a “save url” which the template use as the form post URL during rendering. We’re lazy here, so we’re trying to use the same template (`templates/edit.pt`) for the add view as well as the page edit view, so we create a dummy Page object in order to satisfy the edit form’s desire to have *some* page object exposed as `page`, and Pyramid will render the template associated with this view to a response.

If the view execution *is* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `True`), we scrape the page body from the form data, create a Page object using the name in the `matchdict` `pagename`, and obtain the page body from the request, and save it into the database using `session.add`. We then redirect back to the `view_page` view (the *default view* for a Page) for the newly created page.

The `edit_page` view function

The `edit_page` function will be invoked when a user clicks the “Edit this Page” button on the view form. It renders an edit form but it also acts as the handler for the form it renders. The `matchdict` attribute of the request passed to the `add_page` view will have a `pagename` key matching the name of the page the user wants to edit.

If the view execution is *not* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `False`), the view simply renders the edit form, passing the request, the page object, and a `save_url` which will be used as the action of the generated form.

If the view execution *is* a result of a form submission (if the expression `'form.submitted'` in `request.params` is `True`), the view grabs the `body` element of the request parameter and sets it as the `data` key in the `matchdict`. It then redirects to the default view of the wiki page, which will always be the `view_page` view.

31.5.3 Viewing the Result of Our Edits to `views.py`

The result of all of our edits to `views.py` will leave it looking like this:

```

1  import re
2
3  from docutils.core import publish_parts
4
5  from pyramid.httpexceptions import HTTPFound
6  from pyramid.url import route_url
7
8  from tutorial.models import DBSession
9  from tutorial.models import Page
10
11 # regular expression used to find WikiWords
12 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+) ")
13
14 def view_wiki(request):
15     return HTTPFound(location = route_url('view_page', request,
16                                         pagename='FrontPage'))
17
18 def view_page(request):
19     matchdict = request.matchdict
20     session = DBSession()
21     page = session.query(Page).filter_by(name=matchdict['pagename']).one()
22
23     def check(match):

```

31. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
24     word = match.group(1)
25     exists = session.query(Page).filter_by(name=word).all()
26     if exists:
27         view_url = route_url('view_page', request, pagename=word)
28         return '<a href="%s">%s</a>' % (view_url, word)
29     else:
30         add_url = route_url('add_page', request, pagename=word)
31         return '<a href="%s">%s</a>' % (add_url, word)
32
33     content = publish_parts(page.data, writer_name='html')['html_body']
34     content = wikiwords.sub(check, content)
35     edit_url = route_url('edit_page', request,
36                         pagename=matchdict['pagename'])
37     return dict(page=page, content=content, edit_url=edit_url)
38
39 def add_page(request):
40     name = request.matchdict['pagename']
41     if 'form.submitted' in request.params:
42         session = DBSession()
43         body = request.params['body']
44         page = Page(name, body)
45         session.add(page)
46         return HTTPFound(location = route_url('view_page', request,
47                                             pagename=name))
48     save_url = route_url('add_page', request, pagename=name)
49     page = Page('', '')
50     return dict(page=page, save_url=save_url)
51
52 def edit_page(request):
53     name = request.matchdict['pagename']
54     session = DBSession()
55     page = session.query(Page).filter_by(name=name).one()
56     if 'form.submitted' in request.params:
57         page.data = request.params['body']
58         session.add(page)
59         return HTTPFound(location = route_url('view_page', request,
60                                             pagename=name))
61     return dict(
62         page=page,
63         save_url = route_url('edit_page', request, pagename=name),
64     )
```

31.5.4 Adding Templates

The views we've added all reference a *template*. Each template is a *Chameleon ZPT* template. These templates will live in the `templates` directory of our tutorial package.

The `view.pt` Template

The `view.pt` template is used for viewing a single wiki page. It is used by the `view_page` view function. It should have a `div` that is “structure replaced” with the `content` value provided by the view. It should also have a link on the rendered page that points at the “edit” URL (the URL which invokes the `edit_page` view for the page being viewed).

Once we're done with the `view.pt` template, it will look a lot like the below:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
  <title>${page.name} - Pyramid tutorial wiki (based on
    TurboGears 20-Minute Wiki)</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <meta name="keywords" content="python web application" />
  <meta name="description" content="pyramid web application" />
  <link rel="shortcut icon"
        href="${request.static_url('tutorial:static/favicon.ico')}" />
  <link rel="stylesheet"
        href="${request.static_url('tutorial:static/pylons.css')}"
        type="text/css" media="screen" charset="utf-8" />
  <!--[if lte IE 6]>
  <link rel="stylesheet"
        href="${request.static_url('tutorial:static/ie6.css')}"
        type="text/css" media="screen" charset="utf-8" />
  <![endif]-->
</head>
<body>
  <div id="wrap">
    <div id="top-small">
      <div class="top-small align-center">
        <div>
          
        </div>

```

```
    </div>
</div>
<div id="middle">
  <div class="middle align-right">
    <div id="left" class="app-welcome align-left">
      Viewing <b><span tal:replace="page.name">Page Name
        Goes Here</span></b><br/>
      You can return to the
      <a href="{request.application_url}">FrontPage</a>. <br/>
    </div>
    <div id="right" class="app-welcome align-right"></div>
  </div>
</div>
<div id="bottom">
  <div class="bottom">
    <div tal:replace="structure content">
      Page text goes here.
    </div>
    <p>
      <a tal:attributes="href edit_url" href="">
        Edit this page
      </a>
    </p>
  </div>
</div>
</div>
<div id="footer">
  <div class="footer"
    >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
</div>
</body>
</html>
```

i The names available for our use in a template are always those that are present in the dictionary returned by the view callable. But our templates make use of a `request` object that none of our tutorial views return in their dictionary. This value appears as if “by magic”. However, `request` is one of several names that are available “by default” in a template when a template renderer is used. See **.pt* or **.txt*: *Chameleon Template Renderers* for more information about other names that are available by default in a template when a Chameleon template is used as a renderer.

The edit .pt Template

The `edit.pt` template is used for adding and editing a wiki page. It is used by the `add_page` and `edit_page` view functions. It should display a page containing a form that POSTs back to the “`save_url`” argument supplied by the view. The form should have a “body” textarea field (the page data), and a submit button that has the name “`form.submitted`”. The textarea in the form should be filled with any existing page data when it is rendered.

Once we’re done with the `edit.pt` template, it will look a lot like the below:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
  <title>${page.name} - Pyramid tutorial wiki (based on
    TurboGears 20-Minute Wiki)</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <meta name="keywords" content="python web application" />
  <meta name="description" content="pyramid web application" />
  <link rel="shortcut icon"
        href="${request.static_url('tutorial:static/favicon.ico')}" />
  <link rel="stylesheet"
        href="${request.static_url('tutorial:static/pylons.css')}"
        type="text/css" media="screen" charset="utf-8" />
  <!--[if lte IE 6]>
  <link rel="stylesheet"
        href="${request.static_url('tutorial:static/ie6.css')}"
        type="text/css" media="screen" charset="utf-8" />
  <![endif]-->
</head>
<body>
  <div id="wrap">
    <div id="top-small">
      <div class="top-small align-center">
        <div>
          
        </div>
      </div>
    </div>
    <div id="middle">
      <div class="middle align-right">
        <div id="left" class="app-welcome align-left">
          Editing <b><span tal:replace="page.name">Page Name Goes
            Here</span></b><br/>

```

```
        You can return to the
        <a href="{request.application_url}">FrontPage</a>.<br/>
    </div>
    <div id="right" class="app-welcome align-right"></div>
</div>
</div>
<div id="bottom">
    <div class="bottom">
        <form action="{save_url}" method="post">
            <textarea name="body" tal:content="page.data" rows="10"
                cols="60"/><br/>
            <input type="submit" name="form.submitted" value="Save"/>
        </form>
    </div>
</div>
</div>
<div id="footer">
    <div class="footer"
        >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
</div>
</body>
</html>
```

Static Assets

Our templates name a single static asset named `pylons.css`. We don't need to create this file within our package's `static` directory because it was provided at the time we created the project. This file is a little too long to replicate within the body of this guide, however it is available online.

This CSS file will be accessed via e.g. `http://localhost:6543/static/pylons.css` by virtue of the call to `add_static_view` directive we've made in the `__init__` file. Any number and type of static assets can be placed in this directory (or subdirectories) and are just referred to by URL or by using the convenience method `static_url` e.g. `request.static_url('{package}:static/foo.css')` within templates.

31.5.5 Mapping Views to URLs in `__init__.py`

The `__init__.py` file contains `pyramid.config.Configurator.add_route()` calls which serve to map URLs via *url dispatch* to view functions. First, we'll get rid of the existing route created by the template using the name `home`. It's only an example and isn't relevant to our application.

We then need to add four calls to `add_route`. Note that the *ordering* of these declarations is very important. `route` declarations are matched in the order they're found in the `__init__.py` file.

1. Add a declaration which maps the pattern / (signifying the root URL) to the view named `view_wiki` in our `views.py` file with the name `view_wiki`. This is the *default view* for the wiki.
2. Add a declaration which maps the pattern `/ {pagename}` to the view named `view_page` in our `views.py` file with the view name `view_page`. This is the regular view for a page.
3. Add a declaration which maps the pattern `/add_page/ {pagename}` to the view named `add_page` in our `views.py` file with the name `add_page`. This is the add view for a new page.
4. Add a declaration which maps the pattern `/ {pagename}/edit_page` to the view named `edit_page` in our `views.py` file with the name `edit_page`. This is the edit view for a page.

As a result of our edits, the `__init__.py` file should look something like so:

```
1 from pyramid.config import Configurator
2 from sqlalchemy import engine_from_config
3
4 from tutorial.models import initialize_sql
5
6 def main(global_config, **settings):
7     """ This function returns a WSGI application.
8     """
9     engine = engine_from_config(settings, 'sqlalchemy.')
10    initialize_sql(engine)
11    config = Configurator(settings=settings)
12    config.add_static_view('static', 'tutorial:static')
13    config.add_route('view_wiki', '/', view='tutorial.views.view_wiki')
14    config.add_route('view_page', '/ {pagename}',
15                    view='tutorial.views.view_page',
16                    view_renderer='tutorial:templates/view.pt')
17    config.add_route('add_page', '/add_page/ {pagename}',
18                    view='tutorial.views.add_page',
19                    view_renderer='tutorial:templates/edit.pt')
20    config.add_route('edit_page', '/ {pagename}/edit_page',
21                    view='tutorial.views.edit_page',
22                    view_renderer='tutorial:templates/edit.pt')
23    return config.make_wsgi_app()
```

31.5.6 Viewing the Application in a Browser

We can finally examine our application in a browser. The views we'll try are as follows:

- Visiting `http://localhost:6543` in a browser invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` page object.
- Visiting `http://localhost:6543/FrontPage` in a browser invokes the `view_page` view of the front page page object.
- Visiting `http://localhost:6543/FrontPage/edit_page` in a browser invokes the `edit` view for the front page object.
- Visiting `http://localhost:6543/add_page/SomePageName` in a browser invokes the `add` view for a page.

Try generating an error within the body of a view by adding code to the top of it that generates an exception (e.g. `raise Exception('Forced Exception')`). Then visit the error-raising view in a browser. You should see an interactive exception handler in the browser which allows you to examine values in a post-mortem mode.

31.5.7 Adding Tests

Since we've added a good bit of imperative code here, it's useful to define tests for the views we've created. We'll change our `tests.py` module to look like this:

```
1 import unittest
2
3 from pyramid import testing
4
5 def _initTestingDB():
6     from tutorial.models import DBSession
7     from tutorial.models import Base
8     from sqlalchemy import create_engine
9     engine = create_engine('sqlite://')
10    DBSession.configure(bind=engine)
11    Base.metadata.bind = engine
12    Base.metadata.create_all(engine)
13    return DBSession
14
15 def _registerRoutes(config):
16    config.add_route('view_page', '{pagename}')
17    config.add_route('edit_page', '{pagename}/edit_page')
18    config.add_route('add_page', 'add_page/{pagename}')
19
20 class ViewWikiTests(unittest.TestCase):
21     def setUp(self):
```

```

22     self.config = testing.setUp()
23
24     def tearDown(self):
25         testing.tearDown()
26
27     def test_it(self):
28         from tutorial.views import view_wiki
29         self.config.add_route('view_page', '{pagename}')
30         request = testing.DummyRequest()
31         response = view_wiki(request)
32         self.assertEqual(response.location, 'http://example.com/FrontPage')
33
34 class ViewPageTests(unittest.TestCase):
35     def setUp(self):
36         self.session = _initTestingDB()
37         self.config = testing.setUp()
38
39     def tearDown(self):
40         self.session.remove()
41         testing.tearDown()
42
43     def _callFUT(self, request):
44         from tutorial.views import view_page
45         return view_page(request)
46
47     def test_it(self):
48         from tutorial.models import Page
49         request = testing.DummyRequest()
50         request.matchdict['pagename'] = 'IDoExist'
51         page = Page('IDoExist', 'Hello CruelWorld IDoExist')
52         self.session.add(page)
53         _registerRoutes(self.config)
54         info = self._callFUT(request)
55         self.assertEqual(info['page'], page)
56         self.assertEqual(
57             info['content'],
58             '<div class="document">\n'
59             '<p>Hello <a href="http://example.com/add_page/CruelWorld">'
60             'CruelWorld</a> '
61             '<a href="http://example.com/IDoExist">'
62             'IDoExist</a>'
63             '</p>\n</div>\n')
64         self.assertEqual(info['edit_url'],
65             'http://example.com/IDoExist/edit_page')
66
67

```

```
68 class AddPageTests (unittest.TestCase):
69     def setUp(self):
70         self.session = _initTestingDB()
71         self.config = testing.setUp()
72         self.config.begin()
73
74     def tearDown(self):
75         self.session.remove()
76         testing.tearDown()
77
78     def _callFUT(self, request):
79         from tutorial.views import add_page
80         return add_page(request)
81
82     def test_it_notsubmitted(self):
83         _registerRoutes(self.config)
84         request = testing.DummyRequest()
85         request.matchdict = {'pagename': 'AnotherPage'}
86         info = self._callFUT(request)
87         self.assertEqual(info['page'].data, '')
88         self.assertEqual(info['save_url'],
89                          'http://example.com/add_page/AnotherPage')
90
91     def test_it_submitted(self):
92         from tutorial.models import Page
93         _registerRoutes(self.config)
94         request = testing.DummyRequest({'form.submitted': True,
95                                         'body': 'Hello yo!'})
96         request.matchdict = {'pagename': 'AnotherPage'}
97         self._callFUT(request)
98         page = self.session.query(Page).filter_by(name='AnotherPage').one()
99         self.assertEqual(page.data, 'Hello yo!')
100
101 class EditPageTests (unittest.TestCase):
102     def setUp(self):
103         self.session = _initTestingDB()
104         self.config = testing.setUp()
105
106     def tearDown(self):
107         self.session.remove()
108         testing.tearDown()
109
110     def _callFUT(self, request):
111         from tutorial.views import edit_page
112         return edit_page(request)
113
```

```

114     def test_it_notsubmitted(self):
115         from tutorial.models import Page
116         _registerRoutes(self.config)
117         request = testing.DummyRequest()
118         request.matchdict = {'pagename': 'abc'}
119         page = Page('abc', 'hello')
120         self.session.add(page)
121         info = self._callFUT(request)
122         self.assertEqual(info['page'], page)
123         self.assertEqual(info['save_url'],
124                          'http://example.com/abc/edit_page')
125
126     def test_it_submitted(self):
127         from tutorial.models import Page
128         _registerRoutes(self.config)
129         request = testing.DummyRequest({'form.submitted': True,
130                                         'body': 'Hello yo!'})
131         request.matchdict = {'pagename': 'abc'}
132         page = Page('abc', 'hello')
133         self.session.add(page)
134         response = self._callFUT(request)
135         self.assertEqual(response.location, 'http://example.com/abc')
136         self.assertEqual(page.data, 'Hello yo!')

```

We can then run the tests using something like:

```
1 $ python setup.py test -q
```

The expected output is something like:

```

1 running test
2 running egg_info
3 writing requirements to tutorial.egg-info/requirements.txt
4 writing tutorial.egg-info/PKG-INFO
5 writing top-level names to tutorial.egg-info/top_level.txt
6 writing dependency_links to tutorial.egg-info/dependency_links.txt
7 writing entry points to tutorial.egg-info/entry_points.txt
8 unrecognized .svn/entries format in
9 reading manifest file 'tutorial.egg-info/SOURCES.txt'
10 writing manifest file 'tutorial.egg-info/SOURCES.txt'
11 running build_ext
12 .....
13 -----
14 Ran 6 tests in 0.181s

```

```
15 |  
16 | OK
```

31.6 Adding Authorization

Our application currently allows anyone with access to the server to view, edit, and add pages to our wiki. For purposes of demonstration we'll change our application to allow only people whom possess a specific username (*editor*) to add and edit wiki pages but we'll continue allowing anyone with access to the server to view pages. Pyramid provides facilities for *authorization* and *authentication*. We'll make use of both features to provide security to our application.

The source code for this tutorial stage can be browsed at <http://github.com/Pylons/pyramid/tree/master/docs/tutorials/wiki2/src/authorization/>.

31.6.1 Changing `__init__.py` For Authorization

We're going to be making several changes to our `__init__.py` file which will help us configure an authorization policy.

Adding A Root Factory

We're going to start to use a custom *root factory* within our `__init__.py` file. The objects generated by the root factory will be used as the *context* of each request to our application. In order for Pyramid declarative security to work properly, the context object generated during a request must be decorated with security declarations; when we begin to use a custom root factory to generate our contexts, we can begin to make use of the declarative security features of Pyramid.

We'll modify our `__init__.py`, passing in a *root factory* to our *Configurator* constructor. We'll point it at a new class we create inside our `models.py` file. Add the following statements to your `models.py` file:

```
from pyramid.security import Allow  
from pyramid.security import Everyone  
  
class RootFactory(object):  
    __acl__ = [ (Allow, Everyone, 'view'),  
               (Allow, 'group:editors', 'edit') ]  
    def __init__(self, request):  
        pass
```

The `RootFactory` class we've just added will be used by Pyramid to construct a `context` object. The context is attached to the request object passed to our view callables as the `context` attribute.

All of our context objects will possess an `__acl__` attribute that allows `pyramid.security.Everyone` (a special principal) to view all pages, while allowing only a *principal* named `group:editors` to edit and add pages. The `__acl__` attribute attached to a context is interpreted specially by Pyramid as an access control list during view callable execution. See *Assigning ACLs to your Resource Objects* for more information about what an *ACL* represents.

We'll pass the `RootFactory` we created in the step above in as the `root_factory` argument to a *Configurator*.

Configuring an Authorization Policy

For any Pyramid application to perform authorization, we need to add a `security.py` module (we'll do that shortly) and we'll need to change our `__init__.py` file to add an *authentication policy* and an *authorization policy* which uses the `security.py` file for a *callback*.

We'll change our `__init__.py` file to enable an `AuthTktAuthenticationPolicy` and an `ACLAuthorizationPolicy` to enable declarative security checking. We'll also change `__init__.py` to add a `pyramid.config.Configurator.add_view()` call to points at our *login view callable*, also known as a *forbidden view*. This configures our newly created login view to show up when Pyramid detects that a view invocation can not be authorized. Also, we'll add `view_permission` arguments with the value `edit` to the `edit_page` and `add_page` routes. This indicates that the view callables which these routes reference cannot be invoked without the authenticated user possessing the `edit` permission with respect to the current context.

This makes the assertion that only users who possess the effective `edit` permission at the time of the request may invoke those two views. We've granted the `group:editors` principal the `edit` permission at the root model via its ACL, so only the a user whom is a member of the group named `group:editors` will able to invoke the views associated with the `add_page` or `edit_page` routes.

Viewing Your Changes

When we're done configuring a root factory, adding an authorization policy, and adding views, your application's `__init__.py` will look like this:

31. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
1 from pyramid.config import Configurator
2 from pyramid.authentication import AuthTktAuthenticationPolicy
3 from pyramid.authorization import ACLAuthorizationPolicy
4
5 from sqlalchemy import engine_from_config
6
7 from tutorial.models import initialize_sql
8 from tutorial.security import groupfinder
9
10 def main(global_config, **settings):
11     """ This function returns a WSGI application.
12     """
13     engine = engine_from_config(settings, 'sqlalchemy.')
14     initialize_sql(engine)
15     authn_policy = AuthTktAuthenticationPolicy(
16         'sosecret', callback=groupfinder)
17     authz_policy = ACLAuthorizationPolicy()
18     config = Configurator(settings=settings,
19                           root_factory='tutorial.models.RootFactory',
20                           authentication_policy=authn_policy,
21                           authorization_policy=authz_policy)
22     config.add_static_view('static', 'tutorial:static')
23     config.add_route('view_wiki', '/', view='tutorial.views.view_wiki')
24     config.add_route('login', '/login',
25                     view='tutorial.login.login',
26                     view_renderer='tutorial:templates/login.pt')
27     config.add_route('logout', '/logout',
28                     view='tutorial.login.logout')
29     config.add_route('view_page', '/{pagename}',
30                     view='tutorial.views.view_page',
31                     view_renderer='tutorial:templates/view.pt')
32     config.add_route('add_page', '/add_page/{pagename}',
33                     view='tutorial.views.add_page',
34                     view_renderer='tutorial:templates/edit.pt',
35                     view_permission='edit')
36     config.add_route('edit_page', '/{pagename}/edit_page',
37                     view='tutorial.views.edit_page',
38                     view_renderer='tutorial:templates/edit.pt',
39                     view_permission='edit')
40     config.add_view('tutorial.login.login',
41                   renderer='tutorial:templates/login.pt',
42                   context='pyramid.exceptions.Forbidden')
43     return config.make_wsgi_app()
```

Note that that the `pyramid.authentication.AuthTktAuthenticationPolicy` constructor accepts two arguments: `secret` and `callback`. `secret` is a string representing an encryption key

used by the “authentication ticket” machinery represented by this policy: it is required. The `callback` is a string, representing a *dotted Python name*, which points at the `groupfinder` function in the current directory’s `security.py` file. We haven’t added that module yet, but we’re about to.

Adding `security.py`

Add a `security.py` module within your package (in the same directory as `__init__.py`, `views.py`, etc) with the following content:

```
1 USERS = {'editor': 'editor',
2         'viewer': 'viewer'}
3 GROUPS = {'editor': ['group:editors']}
4
5 def groupfinder(userid, request):
6     if userid in USERS:
7         return GROUPS.get(userid, [])
```

The `groupfinder` defined here is an *authentication policy* “callback”; it is a callable that accepts a `userid` and a `request`. If the `userid` exists in the system, the callback will return a sequence of group identifiers (or an empty sequence if the user isn’t a member of any groups). If the `userid` *does not* exist in the system, the callback will return `None`. In a production system, user and group data will most often come from a database, but here we use “dummy” data to represent user and groups sources. Note that the `editor` user is a member of the `group:editors` group in our dummy group data (the `GROUPS` data structure).

We’ve given the `editor` user membership to the `group:editors` by mapping him to this group in the `GROUPS` data structure (`GROUPS = {'editor': ['group:editors']}`). Since the `groupfinder` function consults the `GROUPS` data structure, this will mean that, as a result of the ACL attached to the root returned by the root factory, and the permission associated with the `add_page` and `edit_page` views, the `editor` user should be able to add and edit pages.

Adding Login and Logout Views

We’ll add a `login` view callable which renders a login form and processes the post from the login form, checking credentials.

We’ll also add a `logout` view callable to our application and provide a link to it. This view will clear the credentials of the logged in user and redirect back to the front page.

We’ll add a different file (for presentation convenience) to add login and logout view callables. Add a file named `login.py` to your application (in the same directory as `views.py`) with the following content:

```
1 from pyramid.httpexceptions import HTTPFound
2 from pyramid.security import remember
3 from pyramid.security import forget
4 from pyramid.url import route_url
5
6 from tutorial.security import USERS
7
8 def login(request):
9     login_url = route_url('login', request)
10    referrer = request.url
11    if referrer == login_url:
12        referrer = '/' # never use the login form itself as came_from
13    came_from = request.params.get('came_from', referrer)
14    message = ''
15    login = ''
16    password = ''
17    if 'form.submitted' in request.params:
18        login = request.params['login']
19        password = request.params['password']
20        if USERS.get(login) == password:
21            headers = remember(request, login)
22            return HTTPFound(location = came_from,
23                             headers = headers)
24        message = 'Failed login'
25
26    return dict(
27        message = message,
28        url = request.application_url + '/login',
29        came_from = came_from,
30        login = login,
31        password = password,
32    )
33
34 def logout(request):
35     headers = forget(request)
36     return HTTPFound(location = route_url('view_wiki', request),
37                     headers = headers)
```

Changing Existing Views

Then we need to change each of our `view_page`, `edit_page` and `add_page` views in `views.py` to pass a “logged in” parameter to its template. We’ll add something like this to each view body:

```

1 from pyramid.security import authenticated_userid
2 logged_in = authenticated_userid(request)

```

We'll then change the return value of these views to pass the *resulting* 'logged_in' value to the template, e.g.:

```

1 return dict(page = context,
2             content = content,
3             logged_in = logged_in,
4             edit_url = edit_url)

```

Adding the login.pt Template

Add a login.pt template to your templates directory. It's referred to within the login view we just added to login.py.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
  <title>Login - Pyramid tutorial wiki (based on TurboGears
    20-Minute Wiki)</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <meta name="keywords" content="python web application" />
  <meta name="description" content="pyramid web application" />
  <link rel="shortcut icon"
    href="${request.static_url('tutorial:static/favicon.ico')}" />
  <link rel="stylesheet"
    href="${request.static_url('tutorial:static/pylons.css')}"
    type="text/css" media="screen" charset="utf-8" />
  <!--[if lte IE 6]>
  <link rel="stylesheet"
    href="${request.static_url('tutorial:static/ie6.css')}"
    type="text/css" media="screen" charset="utf-8" />
  <![endif]-->
</head>
<body>
  <div id="wrap">
    <div id="top-small">
      <div class="top-small align-center">
        <div>

```

```
        
    </div>
</div>
</div>
<div id="middle">
    <div class="middle align-right">
        <div id="left" class="app-welcome align-left">
            <b>Login</b><br/>
            <span tal:replace="message" />
        </div>
        <div id="right" class="app-welcome align-right"></div>
    </div>
</div>
<div id="bottom">
    <div class="bottom">
        <form action="{url}" method="post">
            <input type="hidden" name="came_from" value="{came_from}" />
            <input type="text" name="login" value="{login}" /><br/>
            <input type="password" name="password"
            value="{password}" /><br/>
            <input type="submit" name="form.submitted" value="Log In" />
        </form>
    </div>
</div>
</div>
<div id="footer">
    <div class="footer"
    >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
</div>
</body>
</html>
```

Change view.pt and edit.pt

We'll also need to change our edit.pt and view.pt templates to display a "Logout" link if someone is logged in. This link will invoke the logout view.

To do so we'll add this to both templates within the <div id="right" class="app-welcome align-right"> div:

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

31.6.2 Viewing the Application in a Browser

We can finally examine our application in a browser. The views we'll try are as follows:

- Visiting `http://localhost:6543/` in a browser invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` page object. It is executable by any user.
- Visiting `http://localhost:6543/FrontPage` in a browser invokes the `view_page` view of the `FrontPage` page object.
- Visiting `http://localhost:6543/FrontPage/edit_page` in a browser invokes the `edit` view for the `FrontPage` object. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, a login form will be displayed. Supplying the credentials with the username `editor`, password `editor` will display the edit page form.
- Visiting `http://localhost:6543/add_page/SomePageName` in a browser invokes the `add` view for a page. It is executable by only the `editor` user. If a different user (or the anonymous user) invokes it, a login form will be displayed. Supplying the credentials with the username `editor`, password `editor` will display the edit page form.

31.6.3 Seeing Our Changes To `views.py` and our Templates

Our `views.py` module will look something like this when we're done:

```
1 import re
2
3 from docutils.core import publish_parts
4
5 from pyramid.httpexceptions import HTTPFound
6 from pyramid.security import authenticated_userid
7 from pyramid.url import route_url
8
9 from tutorial.models import DBSession
10 from tutorial.models import Page
11
12 # regular expression used to find WikiWords
13 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+) ")
14
15 def view_wiki(request):
16     return HTTPFound(location = route_url('view_page', request,
17                                           pagename='FrontPage'))
18
```

31. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
19 def view_page(request):
20     pagename = request.matchdict['pagename']
21     session = DBSession()
22     page = session.query(Page).filter_by(name=pagename).one()
23
24     def check(match):
25         word = match.group(1)
26         exists = session.query(Page).filter_by(name=word).all()
27         if exists:
28             view_url = route_url('view_page', request, pagename=word)
29             return '<a href="%s">%s</a>' % (view_url, word)
30         else:
31             add_url = route_url('add_page', request, pagename=word)
32             return '<a href="%s">%s</a>' % (add_url, word)
33
34     content = publish_parts(page.data, writer_name='html')['html_body']
35     content = wikiwords.sub(check, content)
36     edit_url = route_url('edit_page', request, pagename=pagename)
37     logged_in = authenticated_userid(request)
38     return dict(page=page, content=content, edit_url=edit_url,
39                logged_in=logged_in)
40
41 def add_page(request):
42     name = request.matchdict['pagename']
43     if 'form.submitted' in request.params:
44         session = DBSession()
45         body = request.params['body']
46         page = Page(name, body)
47         session.add(page)
48         return HTTPFound(location = route_url('view_page', request,
49                                                pagename=name))
50     save_url = route_url('add_page', request, pagename=name)
51     page = Page('', '')
52     logged_in = authenticated_userid(request)
53     return dict(page=page, save_url=save_url, logged_in=logged_in)
54
55 def edit_page(request):
56     name = request.matchdict['pagename']
57     session = DBSession()
58     page = session.query(Page).filter_by(name=name).one()
59     if 'form.submitted' in request.params:
60         page.data = request.params['body']
61         session.add(page)
62         return HTTPFound(location = route_url('view_page', request,
63                                                pagename=name))
64
```

```

65     logged_in = authenticated_userid(request)
66     return dict(
67         page=page,
68         save_url = route_url('edit_page', request, pagename=name),
69         logged_in = logged_in,
70     )

```

Our edit.pt template will look something like this when we're done:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
    xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
  <title>${page.name} - Pyramid tutorial wiki (based on
    TurboGears 20-Minute Wiki)</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <meta name="keywords" content="python web application" />
  <meta name="description" content="pyramid web application" />
  <link rel="shortcut icon"
    href="${request.static_url('tutorial:static/favicon.ico')}" />
  <link rel="stylesheet"
    href="${request.static_url('tutorial:static/pylons.css')}"
    type="text/css" media="screen" charset="utf-8" />
  <!--[if lte IE 6]>
  <link rel="stylesheet"
    href="${request.static_url('tutorial:static/ie6.css')}"
    type="text/css" media="screen" charset="utf-8" />
  <![endif]-->
</head>
<body>
  <div id="wrap">
    <div id="top-small">
      <div class="top-small align-center">
        <div>
          
        </div>
      </div>
    </div>
    <div id="middle">
      <div class="middle align-right">
        <div id="left" class="app-welcome align-left">
          Editing <b><span tal:replace="page.name">Page Name
            Goes Here</span></b><br/>
          You can return to the

```

```

        <a href="${request.application_url}">FrontPage</a>.<br/>
    </div>
    <div id="right" class="app-welcome align-right">
        <span tal:condition="logged_in">
            <a href="${request.application_url}/logout">Logout</a>
        </span>
    </div>
</div>
</div>
<div id="bottom">
    <div class="bottom">
        <form action="${save_url}" method="post">
            <textarea name="body" tal:content="page.data" rows="10"
                cols="60"/><br/>
            <input type="submit" name="form.submitted" value="Save"/>
        </form>
    </div>
</div>
</div>
<div id="footer">
    <div class="footer">
        >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
    </div>
</body>
</html>

```

Our view.pt template will look something like this when we're done:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
    xmlns:tal="http://xml.zope.org/namespaces/tal">
<head>
    <title>${page.name} - Pyramid tutorial wiki (based on
        TurboGears 20-Minute Wiki)</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <meta name="keywords" content="python web application" />
    <meta name="description" content="pyramid web application" />
    <link rel="shortcut icon"
        href="${request.static_url('tutorial:static/favicon.ico')}" />
    <link rel="stylesheet"
        href="${request.static_url('tutorial:static/pylons.css')}"
        type="text/css" media="screen" charset="utf-8" />
    <!--[if lte IE 6]>
    <link rel="stylesheet"
        href="${request.static_url('tutorial:static/ie6.css')}"

```



```

        type="text/css" media="screen" charset="utf-8" />
    <![endif]-->
</head>
<body>
    <div id="wrap">
        <div id="top-small">
            <div class="top-small align-center">
                <div>
                    
                </div>
            </div>
        </div>
        <div id="middle">
            <div class="middle align-right">
                <div id="left" class="app-welcome align-left">
                    Viewing <b><span tal:replace="page.name">Page Name
                        Goes Here</span></b><br/>
                    You can return to the
                    <a href="{request.application_url}">FrontPage</a>.<br/>
                </div>
                <div id="right" class="app-welcome align-right">
                    <span tal:condition="logged_in">
                        <a href="{request.application_url}/logout">Logout</a>
                    </span>
                </div>
            </div>
        </div>
        <div id="bottom">
            <div class="bottom">
                <div tal:replace="structure content">
                    Page text goes here.
                </div>
                <p>
                    <a tal:attributes="href edit_url" href="">
                        Edit this page
                    </a>
                </p>
            </div>
        </div>
        <div id="footer">
            <div class="footer">
                >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
        </div>
    </body>

```

```
</html>
```

31.6.4 Revisiting the Application

When we revisit the application in a browser, and log in (as a result of hitting an edit or add page and submitting the login form with the `editor` credentials), we'll see a Logout link in the upper right hand corner. When we click it, we're logged out, and redirected back to the front page.

31.7 Distributing Your Application

Once your application works properly, you can create a “tarball” from it by using the `setup.py sdist` command. The following commands assume your current working directory is the `tutorial` package we've created and that the parent directory of the `tutorial` package is a `virtualenv` representing a Pyramid environment.

On UNIX:

```
$ ../bin/python setup.py sdist
```

On Windows:

```
c:\pyramidtut> ..\Scripts\python setup.py sdist
```

The output of such a command will be something like:

```
running sdist
# ... more output ...
creating dist
tar -cf dist/tutorial-0.1.tar tutorial-0.1
gzip -f9 dist/tutorial-0.1.tar
removing 'tutorial-0.1' (and everything under it)
```

Note that this command creates a tarball in the “dist” subdirectory named `tutorial-0.1.tar.gz`. You can send this file to your friends to show them your cool new application. They should be able to install it by pointing the `easy_install` command directly at it. Or you can upload it to PyPI and share it with the rest of the world, where it can be downloaded via `easy_install` remotely like any other package people download from PyPI.

CONVERTING A REPOZE . BFG APPLICATION TO PYRAMID

Prior iterations of Pyramid were released as a package named `repoze.bfg`. `repoze.bfg` users are encouraged to upgrade their deployments to Pyramid, as, after the first final release of Pyramid, further feature development on `repoze.bfg` will cease.

Most existing `repoze.bfg` applications can be converted to a Pyramid application in a completely automated fashion. However, if your application depends on packages which are not “core” parts of `repoze.bfg` but which nonetheless have `repoze.bfg` in their names (e.g. `repoze.bfg.skins`, `repoze.bfg.traversalwrapper`, `repoze.bfg.jinja2`), you will need to find an analogue for each. For example, by the time you read this, there will be a `pyramid_jinja2` package, which can be used instead of `repoze.bfg.jinja2`. If an analogue does not seem to exist for a `repoze.bfg` add-on package that your application uses, please email the Pylons-devel maillist; we’ll convert the package to a Pyramid analogue for you.

Here’s how to convert a `repoze.bfg` application to a Pyramid application:

1. Ensure that your application works under `repoze.bfg version 1.3 or better`. See <http://docs.repoze.org/bfg/1.3/narr/install.html> for `repoze.bfg 1.3` installation instructions. If your application has an automated test suite, run it while your application is using `repoze.bfg 1.3+`. Otherwise, test it manually. It is only safe to proceed to the next step once your application works under `repoze.bfg 1.3+`.

If your application has a proper set of dependencies, and a standard automated test suite, you might test your `repoze.bfg` application against `repoze.bfg 1.3` like so:

32. CONVERTING A REPOZE.BFG APPLICATION TO PYRAMID

```
$ bfgenv/bin/python setup.py test
```

bfgenv above will be the virtualenv into which you've installed `repoze.bfg` 1.3.

2. Install Pyramid into a *separate* virtualenv as per the instructions in *Installing Pyramid*. The Pyramid virtualenv should be separate from the one you've used to install `repoze.bfg`. A quick way to do this:

```
$ cd ~
$ virtualenv --no-site-packages pyramidenv
$ cd pyramidenv
$ bin/easy_install pyramid
```

3. Put a *copy* of your `repoze.bfg` application into a temporary location (perhaps by checking a fresh copy of the application out of a version control repository). For example:

```
$ cd /tmp
$ svn co http://my.server/my/bfg/application/trunk bfgapp
```

4. Use the `bfg2pyramid` script present in the `bin` directory of the Pyramid virtualenv to convert all `repoze.bfg` Python import statements into compatible Pyramid import statements. `bfg2pyramid` will also fix ZCML directive usages of common `repoze.bfg` directives. You invoke `bfg2pyramid` by passing it the *path* of the copy of your application. The path passed should contain a "setup.py" file, representing your `repoze.bfg` application's setup script. `bfg2pyramid` will change the copy of the application *in place*.

```
$ ~/pyramidenv/bfg2pyramid /tmp/bfgapp
```

`bfg2pyramid` will convert the following `repoze.bfg` application aspects to Pyramid compatible analogues:

- Python import statements naming `repoze.bfg` APIs will be converted to Pyramid compatible import statements. Every Python file beneath the top-level path will be visited and converted recursively, except Python files which live in directories which start with a `.` (dot).
- Each ZCML file found (recursively) within the path will have the default `xmlns` attribute attached to the `configure` tag changed from `http://namespaces.repoze.org/bfg` to `http://pylonshq.com/pyramid`. Every ZCML file beneath the top-level path (files ending with `.zcml`) will be visited and converted recursively, except ZCML files which live in directories which start with a `.` (dot).

-
- ZCML files which contain directives that have attributes which name a `repoze.bfg` API module or attribute of an API module (e.g. `context="repoze.bfg.exceptions.NotFound"`) will be converted to Pyramid compatible ZCML attributes (e.g. `context="pyramid.exceptions.NotFound"`). Every ZCML file beneath the top-level path (files ending with `.zcml`) will be visited and converted recursively, except ZCML files which live in directories which start with a `.` (dot).
5. Edit the `setup.py` file of the application you've just converted (if you've been using the example paths, this will be `/tmp/bfgapp/setup.py`) to depend on the `pyramid` distribution instead of `repoze.bfg` distribution in its `install_requires` list. If you used a paster template to create the `repoze.bfg` application, you can do so by changing the `requires` line near the top of the `setup.py` file. The original may look like this:

```
requires = ['repoze.bfg', ... other dependencies ...]
```

Edit the `setup.py` so it has:

```
requires = ['pyramid', ... other dependencies ...]
```

All other `install_requires` and `tests_requires` dependencies save for the one on `repoze.bfg` can remain the same.

6. Convert any `install_requires` dependencies your application has on other add-on packages which have `repoze.bfg` in their names to Pyramid compatible analogues (e.g. `repoze.bfg.jinja2` should be replaced with `pyramid_jinja2`). You may need to adjust configuration options and/or imports in your `repoze.bfg` application after replacing these add-ons. Read the documentation of the Pyramid add-on package for information.
7. *Only if you use ZCML and add-ons which use ZCML:* The default `xmlns` of the `configure` tag in ZCML has changed. The `bfg2pyramid` script effects the default namespace change (it changes the `configure` tag default `xmlns` from `http://namespaces.repoze.org/bfg` to `http://pylonshq.com/pyramid`).

This means that uses of add-ons which define ZCML directives in the `http://namespaces.repoze.org/bfg` namespace will begin to “fail” (they’re actually not really failing, but your ZCML assumes that they will always be used within a `configure` tag which names the `http://namespaces.repoze.org/bfg` namespace as its default `xmlns`). Symptom: when you attempt to start the application, an error such as `ConfigurationError: ('Unknown directive', u'http://namespaces.repoze.org/bfg', u'workflow')` is printed to the console and the application fails to start. In such a case, either add an `xmlns="http://namespaces.repoze.org/bfg"` attribute to each tag which causes a failure, or define a namespace alias in the `configure` tag and prefix each failing tag. For example, change this “failing” tag instance:

32. CONVERTING A REPOZE.BFG APPLICATION TO PYRAMID

```
<configure xmlns="http://pylonshq.com/pyramid">
  <failingtag attr="foo"/>
</configure>
```

To this, which will begin to succeed:

```
<configure xmlns="http://pylonshq.com/pyramid"
           xmlns:bfq="http://namespaces.repoze.org/bfq">
  <bfq:failingtag attr="foo"/>
</configure>
```

You will also need to add the `pyramid_zcml` package to your `setup.py` `install_requires` list. In Pyramid, ZCML configuration became an optional add-on supported by the `pyramid_zcml` package.

8. Retest your application using Pyramid. This might be as easy as:

```
$ cd /tmp/bfgapp
$ ~/pyramidenv/bin/python setup.py test
```

9. Fix any test failures.

10. Fix any code which generates deprecation warnings.

11. Start using the converted version of your application. Celebrate.

Two terminological changes have been made to Pyramid which make its documentation and newer APIs different than those of `repoze.bfg`. The concept that BFG called `model` is called `resource` in Pyramid and the concept that BFG called `resource` is called `asset` in Pyramid. Various APIs have changed as a result (although all have backwards compatible shims). Additionally, the environment variables that influenced server behavior which used to be prefixed with `BFG_` (such as `BFG_DEBUG_NOTFOUND`) must now be prefixed with `PYRAMID_`.

RUNNING PYRAMID ON GOOGLE'S APP ENGINE

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

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

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

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

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

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

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

4. Install Pyramid into the *virtualenv*

33. RUNNING PYRAMID ON GOOGLE'S APP ENGINE

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

This will install Pyramid in the environment.

5. Create your application

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

```
$ cd app  
$ rm -rf pyramidapp  
$ bin/paster create -t pyramid_starter pyramidapp  
$ mv pyramidapp aside  
$ mv aside/pyramidapp .  
$ rm -rf aside
```

6. Edit `config.py`

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

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

7. Edit `runner.py`

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

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

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

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

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

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

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

```
1 $ cd ../../
2 $ python2.5 /usr/local/bin/dev_appserver.py pyramidapp/app/
```

Startup success looks something like this:

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

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

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

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

10. Sign up for a GAE account and create an application. You’ll need a mobile phone to accept an SMS in order to receive authorization.
11. Edit the application’s ID in `app.yaml` to match the application name you created during GAE account setup.

33. RUNNING PYRAMID ON GOOGLE'S APP ENGINE

```
application: mycoolpyramidapp
```

12. Upload the application

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

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

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

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

A successful upload looks like so:

```
[chrism@vitaminf pyramidapp]$ python2.5 /usr/local/bin/appcfg.py \
                             update ../pyramidapp/app/

Scanning files on local disk.
Scanned 500 files.
# ... more output ...
Will check again in 16 seconds.
Checking if new version is ready to serve.
Closing update: new version is ready to start serving.
Uploading index definitions.
```

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

33.1 Zippping Files Via Pip

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

First, see which packages are available for zippping:

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

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

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

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

- Chameleon
- zope.i18n

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

RUNNING A PYRAMID APPLICATION UNDER MOD_WSGI

mod_wsgi is an Apache module developed by Graham Dumpleton. It allows *WSGI* programs to be served using the Apache web server.

This guide will outline broad steps that can be used to get a Pyramid application running under Apache via *mod_wsgi*. This particular tutorial was developed under Apple's Mac OS X platform (Snow Leopard, on a 32-bit Mac), but the instructions should be largely the same for all systems, delta specific path information for commands and files.

i Unfortunately these instructions almost certainly won't work for deploying a Pyramid application on a Windows system using *mod_wsgi*. If you have experience with Pyramid and *mod_wsgi* on Windows systems, please help us document this experience by submitting documentation to the Pylons-devel maillist.

1. The tutorial assumes you have Apache already installed on your system. If you do not, install Apache 2.X for your platform in whatever manner makes sense.
2. Once you have Apache installed, install *mod_wsgi*. Use the (excellent) installation instructions for your platform into your system's Apache installation.
3. Install *virtualenv* into the Python which *mod_wsgi* will run using the `easy_install` program.

```
$ sudo /usr/bin/easy_install-2.6 virtualenv
```

This command may need to be performed as the root user.

4. Create a *virtualenv* which we'll use to install our application.

34. RUNNING A PYRAMID APPLICATION UNDER MOD_WSGI

```
$ cd ~
$ mkdir modwsgi
$ cd modwsgi
$ /usr/local/bin/virtualenv --no-site-packages env
```

5. Install Pyramid into the newly created virtualenv:

```
$ cd ~/modwsgi/env
$ bin/easy_install pyramid
```

6. Create and install your Pyramid application. For the purposes of this tutorial, we'll just be using the `pyramid_starter` application as a baseline application. Substitute your existing Pyramid application as necessary if you already have one.

```
$ cd ~/modwsgi/env
$ bin/paster create -t pyramid_starter myapp
$ cd myapp
$ ../bin/python setup.py install
```

7. Within the virtualenv directory (`~/modwsgi/env`), create a script named `pyramid.wsgi`. Give it these contents:

```
from pyramid.paster import get_app
application = get_app(
    '/Users/chris/modwsgi/env/myapp/production.ini', 'main')
```

The first argument to `get_app` is the project Paste configuration file name. It's best to use the `production.ini` file provided by your Pyramid paster template, as it contains settings appropriate for production. The second is the name of the section within the `.ini` file that should be loaded by `mod_wsgi`. The assignment to the name `application` is important: `mod_wsgi` requires finding such an assignment when it opens the file.

8. Make the `pyramid.wsgi` script executable.

```
$ cd ~/modwsgi/env
$ chmod 755 pyramid.wsgi
```

9. Edit your Apache configuration and add some stuff. I happened to create a file named `/etc/apache2/other/modwsgi.conf` on my own system while installing Apache, so this stuff went in there.

```
# Use only 1 Python sub-interpreter.  Multiple sub-interpreters
# play badly with C extensions.
WSGIApplicationGroup %{GLOBAL}
WSGIPassAuthorization On
WSGIDaemonProcess pyramid user=chrism group=staff processes=1 \
    threads=4 \
    python-path=/Users/chrism/modwsgi/env/lib/python2.6/site-packages
WSGIScriptAlias /myapp /Users/chrism/modwsgi/env/pyramid.wsgi

<Directory /Users/chrism/modwsgi/env>
    WSGIProcessGroup pyramid
    Order allow, deny
    Allow from all
</Directory>
```

10. Restart Apache

```
$ sudo /usr/sbin/apachectl restart
```

11. Visit `http://localhost/myapp` in a browser. You should see the sample application rendered in your browser.

mod_wsgi has many knobs and a great variety of deployment modes. This is just one representation of how you might use it to serve up a Pyramid application. See the *mod_wsgi* configuration documentation for more in-depth configuration information.

Part III

API Reference

PYRAMID . AUTHORIZATION

class `ACLAuthorizationPolicy`

An *authorization policy* which consults an *ACL* object attached to a *context* to determine authorization information about a *principal* or multiple principals. If the context is part of a *lineage*, the context's parents are consulted for ACL information too. The following is true about this security policy.

- When checking whether the 'current' user is permitted (via the `permits` method), the security policy consults the `context` for an ACL first. If no ACL exists on the context, or one does exist but the ACL does not explicitly allow or deny access for any of the effective principals, consult the context's parent ACL, and so on, until the lineage is exhausted or we determine that the policy permits or denies.

During this processing, if any `pyramid.security.Deny` ACE is found matching any principal in `principals`, stop processing by returning an `pyramid.security.ACLDenied` instance (equals `False`) immediately. If any `pyramid.security.Allow` ACE is found matching any principal, stop processing by returning an `pyramid.security.ACLAllowed` instance (equals `True`) immediately. If we exhaust the context's *lineage*, and no ACE has explicitly permitted or denied access, return an instance of `pyramid.security.ACLDenied` (equals `False`).

- When computing principals allowed by a permission via the `pyramid.security.principals_allowed_by_permission()` method, we compute the set of principals that are explicitly granted the permission in the provided context. We do this by walking 'up' the object graph *from the root* to the context. During this walking process, if we find an explicit `pyramid.security.Allow` ACE for a principal that matches the `permission`, the principal is included in the allow list. However, if later in the walking process that principal is mentioned

in any `pyramid.security.Deny` ACE for the permission, the principal is removed from the allow list. If a `pyramid.security.Deny` to the principal `pyramid.security.Everyone` is encountered during the walking process that matches the permission, the allow list is cleared for all principals encountered in previous ACLs. The walking process ends after we've processed the any ACL directly attached to context; a set of principals is returned.

PYRAMID . AUTHENTICATION

36.1 Authentication Policies

```
class AuthTktAuthenticationPolicy (secret, callback=None, cookie_name='auth_tkt',  
secure=False, include_ip=False, timeout=None,  
reissue_time=None, max_age=None, path='/',  
http_only=False, wild_domain=True)
```

A Pyramid *authentication policy* which obtains data from an `paste.auth.auth_tkt` cookie.

Constructor Arguments

`secret`

The secret (a string) used for `auth_tkt` cookie signing. Required.

`callback`

Default: `None`. A callback passed the `userid` and the request, expected to return `None` if the `userid` doesn't exist or a sequence of group identifiers (possibly empty) if the user does exist. If `callback` is `None`, the `userid` will be assumed to exist with no groups. Optional.

`cookie_name`

Default: `auth_tkt`. The cookie name used (string). Optional.

`secure`

Default: `False`. Only send the cookie back over a secure conn. Optional.

`include_ip`

Default: `False`. Make the requesting IP address part of the authentication data in the cookie. Optional.

`timeout`

Default: `None`. Maximum number of seconds which a newly issued ticket will be considered valid. After this amount of time, the ticket will expire (effectively logging the user out). If this value is `None`, the ticket never expires. Optional.

`reissue_time`

Default: `None`. If this parameter is set, it represents the number of seconds that must pass before an authentication token cookie is reissued. The duration is measured as the number of seconds since the last `auth_tkt` cookie was issued and 'now'. If the `timeout` value is `None`, this parameter has no effect. If this parameter is provided, and the value of `timeout` is not `None`, the value of `reissue_time` must be smaller than value of `timeout`. A good rule of thumb: if you want auto-reissued cookies: set this to the `timeout` value divided by ten. If this value is 0, a new ticket cookie will be reissued on every request which needs authentication. Optional.

`max_age`

Default: `None`. The max age of the `auth_tkt` cookie, in seconds. This differs from `timeout` inasmuch as `timeout` represents the lifetime of the ticket contained in the cookie, while this value represents the lifetime of the cookie itself. When this value is set, the cookie's `Max-Age` and `Expires` settings will be set, allowing the `auth_tkt` cookie to last between browser sessions. It is typically nonsensical to set this to a value that is lower than `timeout` or `reissue_time`, although it is not explicitly prevented. Optional.

`path`

Default: `/`. The path for which the `auth_tkt` cookie is valid. May be desirable if the application only serves part of a domain. Optional.

`http_only`

Default: `False`. Hide cookie from JavaScript by setting the `HttpOnly` flag. Not honored by all browsers. Optional.

wild_domain

Default: `True`. An `auth_tkt` cookie will be generated for the wildcard domain. Optional.

class RepozeWho1AuthenticationPolicy (*identifier_name='auth_tkt', callback=None*)

A Pyramid *authentication policy* which obtains data from the `repoze.who 1.X` WSGI 'API' (the `repoze.who.identity` key in the WSGI environment).

Constructor Arguments

`identifier_name`

Default: `auth_tkt`. The `repoze.who` plugin name that performs remember/forget. Optional.

`callback`

Default: `None`. A callback passed the `repoze.who` identity and the *request*, expected to return `None` if the user represented by the identity doesn't exist or a sequence of group identifiers (possibly empty) if the user does exist. If `callback` is `None`, the `userid` will be assumed to exist with no groups.

class RemoteUserAuthenticationPolicy (*environ_key='REMOTE_USER', callback=None*)

A Pyramid *authentication policy* which obtains data from the `REMOTE_USER` WSGI environment variable.

Constructor Arguments

`environ_key`

Default: `REMOTE_USER`. The key in the WSGI `environ` which provides the `userid`.

`callback`

Default: `None`. A callback passed the `userid` and the *request*, expected to return `None` if the `userid` doesn't exist or a sequence of group identifiers (possibly empty) if the user does exist. If `callback` is `None`, the `userid` will be assumed to exist with no groups.

36.2 Helper Classes

class AuthTktCookieHelper (*secret, cookie_name='auth_tkt', secure=False, include_ip=False, timeout=None, reissue_time=None, max_age=None, http_only=False, path='/', wild_domain=True*)

A helper class for use in third-party authentication policy implementations. See `pyramid.authentication.AuthTktAuthenticationPolicy` for the meanings of the constructor arguments.

PYRAMID.CHAMELEON_TEXT

`get_template(path)`

Return the underlying object representing a *Chameleon* text template using the template implied by the `path` argument. The `path` argument may be a package-relative path, an absolute path, or a *asset specification*.



This API is deprecated in Pyramid 1.0. Use the `implementation()` method of a template renderer retrieved via `pyramid.renderers.get_renderer()` instead.

`render_template(path, **kw)`

Render a *Chameleon* text template using the template implied by the `path` argument. The `path` argument may be a package-relative path, an absolute path, or a *asset specification*. The arguments in `*kw` are passed as top-level names to the template, and so may be used within the template itself. Returns a string.



This API is deprecated in Pyramid 1.0. Use `pyramid.renderers.render()` instead.

`render_template_to_response(path, **kw)`

Render a *Chameleon* text template using the template implied by the `path` argument. The `path` argument may be a package-relative path, an absolute path, or a *asset specification*. The arguments in `*kw` are passed as top-level names to the template, and so may be used within the template itself. Returns a *Response* object with the body as the template result.



This API is deprecated in Pyramid 1.0. Use `pyramid.renderers.render_to_response()` instead.

These APIs will work against template files which contain simple `{Genshi}` - style replacement markers.

The API of `pyramid.chameleon_text` is identical to that of `pyramid.chameleon_zpt`; only its import location is different. If you need to import an API functions from this module as well as the `pyramid.chameleon_zpt` module within the same view file, use the `as` feature of the Python import statement, e.g.:

```
1 from pyramid.chameleon_zpt import render_template as zpt_render
2 from pyramid.chameleon_text import render_template as text_render
```

PYRAMID.CHAMELEON_ZPT

`get_template(path)`

Return the underlying object representing a *Chameleon* ZPT template using the template implied by the `path` argument. The `path` argument may be a package-relative path, an absolute path, or a *asset specification*.



This API is deprecated in Pyramid 1.0. Use the `implementation()` method of a template renderer retrieved via `pyramid.renderers.get_renderer()` instead.

`render_template(path, **kw)`

Render a *Chameleon* ZPT template using the template implied by the `path` argument. The `path` argument may be a package-relative path, an absolute path, or a *asset specification*. The arguments in `*kw` are passed as top-level names to the template, and so may be used within the template itself. Returns a string.



This API is deprecated in Pyramid 1.0. Use `pyramid.renderers.render()` instead.

`render_template_to_response(path, **kw)`

Render a *Chameleon* ZPT template using the template implied by the `path` argument. The `path` argument may be a package-relative path, an absolute path, or a *asset specification*. The arguments in `*kw` are passed as top-level names to the template, and so may be used within the template itself. Returns a *Response* object with the body as the template result.



This API is deprecated in Pyramid 1.0. Use `pyramid.renderers.render_to_response()` instead.

These APIs will work against files which supply template text which matches the *ZPT* specification.

The API of `pyramid.chameleon_zpt` is identical to that of `pyramid.chameleon_text`; only its import location is different. If you need to import an API functions from this module as well as the `pyramid.chameleon_text` module within the same view file, use the `as` feature of the Python import statement, e.g.:

```
1 from pyramid.chameleon_zpt import render_template as zpt_render
2 from pyramid.chameleon_text import render_template as text_render
```

PYRAMID.CONFIG

```
class Configurator (registry=None, package=None, settings=None, root_factory=None,  
                    authentication_policy=None, authorization_policy=None,  
                    renderers=DEFAULT_RENDERERS, debug_logger=None,  
                    locale_negotiator=None, request_factory=None, ren-  
                    derer_globals_factory=None, default_permission=None, ses-  
                    sion_factory=None, autocommit=False)
```

A Configurator is used to configure a Pyramid *application registry*.

The Configurator accepts a number of arguments: `registry`, `package`, `settings`, `root_factory`, `authentication_policy`, `authorization_policy`, `renderers`, `debug_logger`, `locale_negotiator`, `request_factory`, `renderer_globals_factory`, `default_permission`, `session_factory`, and `autocommit`.

If the `registry` argument is passed as a non-None value, it must be an instance of the `pyramid.registry.Registry` class representing the registry to configure. If `registry` is None, the configurator will create a `pyramid.registry.Registry` instance itself; it will also perform some default configuration that would not otherwise be done. After construction, the configurator may be used to add configuration to the registry. The overall state of a registry is called the ‘configuration state’.



If a `registry` is passed to the Configurator constructor, all other constructor arguments except `package` are ignored.

If the `package` argument is passed, it must be a reference to a Python *package* (e.g. `sys.modules['thepackage']`) or a *dotted Python name* to same. This value is used as a


basis to convert relative paths passed to various configuration methods, such as methods which accept a `renderer` argument, into absolute paths. If `None` is passed (the default), the package is assumed to be the Python package in which the *caller* of the `Configurator` constructor lives.

If the `settings` argument is passed, it should be a Python dictionary representing the deployment settings for this application. These are later retrievable using the `pyramid.registry.Registry.settings` attribute (aka `request.registry.settings`).

If the `root_factory` argument is passed, it should be an object representing the default *root factory* for your application or a *dotted Python name* to same. If it is `None`, a default root factory will be used.

If `authentication_policy` is passed, it should be an instance of an *authentication policy* or a *dotted Python name* to same.

If `authorization_policy` is passed, it should be an instance of an *authorization policy* or a *dotted Python name* to same.

 A `ConfigurationError` will be raised when an authorization policy is supplied without also supplying an authentication policy (authorization requires authentication).

If `renderers` is passed, it should be a list of tuples representing a set of *renderer* factories which should be configured into this application (each tuple representing a set of positional values that should be passed to `pyramid.config.Configurator.add_renderer()`). If it is not passed, a default set of renderer factories is used.

If `debug_logger` is not passed, a default debug logger that logs to `stderr` will be used. If it is passed, it should be an instance of the `logging.Logger` (PEP 282) standard library class or a *dotted Python name* to same. The debug logger is used by Pyramid itself to log warnings and authorization debugging information.

If `locale_negotiator` is passed, it should be a *locale negotiator* implementation or a *dotted Python name* to same. See *Using a Custom Locale Negotiator*.

If `request_factory` is passed, it should be a *request factory* implementation or a *dotted Python name* to same. See *Changing the Request Factory*. By default it is `None`, which means use the default request factory.

If `renderer_globals_factory` is passed, it should be a *renderer globals* factory implementation or a *dotted Python name* to same. See *Adding Renderer Globals*. By default, it is `None`, which means use no renderer globals factory.

If `default_permission` is passed, it should be a *permission* string to be used as the default permission for all view configuration registrations performed against this Configurator. An example of a permission string: `'view'`. Adding a default permission makes it unnecessary to protect each view configuration with an explicit permission, unless your application policy requires some exception for a particular view. By default, `default_permission` is `None`, meaning that view configurations which do not explicitly declare a permission will always be executable by entirely anonymous users (any authorization policy in effect is ignored). See also *Setting a Default Permission*.

If `session_factory` is passed, it should be an object which implements the *session factory* interface. If a nondefault value is passed, the `session_factory` will be used to create a session object when `request.session` is accessed. Note that the same outcome can be achieved by calling `pyramid.config.Configurator.set_session_factory()`. By default, this argument is `None`, indicating that no session factory will be configured (and thus accessing `request.session` will throw an error) unless `set_session_factory` is called later during configuration.

If `autocommit` is `True`, every method called on the configurator will cause an immediate action, and no configuration conflict detection will be used. If `autocommit` is `False`, most methods of the configurator will defer their action until `pyramid.config.Configurator.commit()` is called. When `pyramid.config.Configurator.commit()` is called, the actions implied by the called methods will be checked for configuration conflicts unless `autocommit` is `True`. If a conflict is detected a `ConfigurationConflictError` will be raised. Calling `pyramid.config.Configurator.make_wsgi_app()` always implies a final commit.

If `default_view_mapper` is passed, it will be used as the default *view mapper* factory for view configurations that don't otherwise specify one (see `pyramid.interfaces.IViewMapperFactory`). If a `default_view_mapper` is not passed, a superdefault view mapper will be used.

registry

The *application registry* which holds the configuration associated with this configurator.

begin (*request=None*)

Indicate that application or test configuration has begun. This pushes a dictionary containing the *application registry* implied by `registry` attribute of this configurator and the *request* implied by the `request` argument on to the *thread local* stack consulted by various `pyramid.threadlocal` API functions.

end ()

Indicate that application or test configuration has ended. This pops the last value pushed on to the *thread local* stack (usually by the `begin` method) and returns that value.

hook_zca()


Call `zope.component.getSiteManager.sethook()` with the argument `pyramid.threadlocal.get_current_registry`, causing the *Zope Component Architecture* 'global' APIs such as `zope.component.getSiteManager()`, `zope.component.getAdapter()` and others to use the Pyramid *application registry* rather than the Zope 'global' registry. If `zope.component` cannot be imported, this method will raise an `ImportError`.

unhook_zca()

Call `zope.component.getSiteManager.reset()` to undo the action of `pyramid.config.Configurator.hook_zca()`. If `zope.component` cannot be imported, this method will raise an `ImportError`.

get_settings()

Return a *deployment settings* object for the current application. A deployment settings object is a dictionary-like object that contains key/value pairs based on the dictionary passed as the `settings` argument to the `pyramid.config.Configurator` constructor or the `pyramid.router.make_app()` API.

 For backwards compatibility, dictionary keys can also be looked up as attributes of the settings object.


 the `pyramid.registry.Registry.settings` API performs the same duty.

commit()

Commit any pending configuration actions. If a configuration conflict is detected in the pending configuration acts, this method will raise a `ConfigurationConflictError`; within the traceback of this error will be information about the source of the conflict, usually including file names and line numbers of the cause of the configuration conflicts.

action (*discriminator*, *callable=None*, *args=()*, *kw=None*, *order=0*)

Register an action which will be executed when `pyramid.config.Configuration.commit()` is called (or executed immediately if `autocommit` is `True`).

 This method is typically only used by Pyramid framework extension authors, not by Pyramid application developers.

The `discriminator` uniquely identifies the action. It must be given, but it can be `None`, to indicate that the action never conflicts. It must be a hashable value.

The `callable` is a callable object which performs the action. It is optional. `args` and `kw` are tuple and dict objects respectively, which are passed to `callable` when this action is executed.

`order` is a crude order control mechanism, only rarely used (has no effect when `autocommit` is `True`).

include (**callables*)

Include one or more configuration callables, to support imperative application extensibility.

A configuration callable should be a callable that accepts a single argument named `config`, which will be an instance of a *Configurator* (be warned that it will not be the same configurator instance on which you call this method, however). The code which runs as the result of calling the callable should invoke methods on the configurator passed to it which add configuration state. The return value of a callable will be ignored.

Values allowed to be presented via the **callables* argument to this method: any callable Python object or any *dotted Python name* which resolves to a callable Python object. It may also be a Python *module*, in which case, the module will be searched for a callable named `includeme`, which will be treated as the configuration callable.

For example, if the `includeme` function below lives in a module named `myapp.myconfig`:

```
1 # myapp.myconfig module
2
3 def my_view(request):
4     from pyramid.response import Response
5     return Response('OK')
6
7 def includeme(config):
8     config.add_view(my_view)
```

You might cause it be included within your Pyramid application like so:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     config = Configurator()
5     config.include('myapp.myconfig.includeme')
```

Because the function is named `includeme`, the function name can also be omitted from the dotted name reference:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     config = Configurator()
5     config.include('myapp.myconfig')
```

Included configuration statements will be overridden by local configuration statements if an included callable causes a configuration conflict by registering something with the same configuration parameters.

add_directive (*name*, *directive*, *action_wrap=True*)

Add a directive method to the configurator.

Framework extenders can add directive methods to a configurator by instructing their users to call `config.add_directive('somename', 'some.callable')`. This will make `some.callable` accessible as `config.somename.some.callable`. `some.callable` should be a function which accepts `config` as a first argument, and arbitrary positional and keyword arguments following. It should use `config.action` as necessary to perform actions. Directive methods can then be invoked like 'built-in' directives such as `add_view`, `add_route`, etc.

The `action_wrap` argument should be `True` for directives which perform `config.action` with potentially conflicting discriminators. `action_wrap` will cause the directive to be wrapped in a decorator which provides more accurate conflict cause information.

`add_directive` does not participate in conflict detection, and later calls to `add_directive` will override earlier calls.

with_package (*package*)

Return a new `Configurator` instance with the same registry as this configurator using the package supplied as the `package` argument to the new configurator. `package` may be an actual Python package object or a Python dotted name representing a package.

maybe_dotted (*dotted*)

Resolve the *dotted Python name* `dotted` to a global Python object. If `dotted` is not a string, return it without attempting to do any name resolution. If `dotted` is a relative dotted name (e.g. `.foo.bar`), consider it relative to the `package` argument supplied to this `Configurator`'s constructor.

absolute_asset_spec (*relative_spec*)

Resolve the potentially relative *asset specification* string passed as `relative_spec` into an absolute asset specification string and return the string. Use the `package` of this configurator as the package to which the asset specification will be considered relative when generating an absolute asset specification. If the provided `relative_spec` argument is already absolute, or if the `relative_spec` is not a string, it is simply returned.

setup_registry (*settings=None, root_factory=None, authentication_policy=None, renderers=DEFAULT_RENDERERS, debug_logger=None, locale_negotiator=None, request_factory=None, render_globals_factory=None*)

When you pass a non-None `registry` argument to the `Configurator` constructor, no initial ‘setup’ is performed against the registry. This is because the registry you pass in may have already been initialized for use under Pyramid via a different configurator. However, in some circumstances (such as when you want to use the Zope ‘global’ registry instead of a registry created as a result of the `Configurator` constructor), or when you want to reset the initial setup of a registry, you *do* want to explicitly initialize the registry associated with a `Configurator` for use under Pyramid. Use `setup_registry` to do this initialization.

`setup_registry` configures settings, a root factory, security policies, renderers, a debug logger, a locale negotiator, and various other settings using the configurator’s current registry, as per the descriptions in the `Configurator` constructor.

add_renderer (*name, factory*)

Add a Pyramid *renderer* factory to the current configuration state.

The *name* argument is the renderer name. Use `None` to represent the default renderer (a renderer which will be used for all views unless they name another renderer specifically).

The *factory* argument is Python reference to an implementation of a *renderer* factory or a *dotted Python name* to same.

Note that this function must be called *before* any `add_view` invocation that names the renderer name as an argument. As a result, it’s usually a better idea to pass globally used renderers into the `Configurator` constructor in the sequence of renderers passed as *renderer* than it is to use this method.

add_route (*name, pattern=None, view=None, view_for=None, permission=None, factory=None, for_=None, header=None, xhr=False, accept=None, path_info=None, request_method=None, request_param=None, traverse=None, custom_predicates=(), view_permission=None, renderer=None, view_renderer=None, view_context=None, view_attr=None, use_global_views=False, path=None, pregenerator=None*)

Add a *route configuration* to the current configuration state, as well as possibly a *view configuration* to be used to specify a *view callable* that will be invoked when this route matches. The arguments to this method are divided into *predicate*, *non-predicate*, and *view-related* types. *Route predicate* arguments narrow the circumstances in which a route will be match a request; non-predicate arguments are informational.

Non-Predicate Arguments

name

The name of the route, e.g. `myroute`. This attribute is required. It must be unique among all defined routes in a given application.

factory

A Python object (often a function or a class) or a *dotted Python name* which refers to the same object that will generate a Pyramid root resource object when this route matches. For example, `mypackage.resources.MyFactory`. If this argument is not specified, a default root factory will be used.

traverse

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

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

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

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

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


pregenerator

This option should be a callable object that implements the `pyramid.interfaces.IRoutePregenerator` interface. A *pregenerator* is a callable called by the `pyramid.url.route_url` function to augment or replace the arguments it is passed when generating a URL for the route. This is a feature not often used directly by applications, it is meant to be hooked by frameworks that use Pyramid as a base.

Predicate Arguments

pattern

The pattern of the route e.g. `ideas/{idea}`. This argument is required. See *Route Pattern Syntax* for information about the syntax of route patterns. If the pattern doesn't match the current URL, route matching continues.

 For backwards compatibility purposes (as of Pyramid 1.0), a `path` keyword argument passed to this function will be used to represent the pattern value if the `pattern` argument is `None`. If both `path` and `pattern` are passed, `pattern` wins.

xhr

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

request_method

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

path_info

This value represents a regular expression pattern that will be tested against the `PATH_INFO` WSGI environment variable. If the regex matches, this predicate will return `True`. If this predicate returns `False`, route matching continues.

request_param

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

header

This argument represents an HTTP header name or a header name/value pair. If the argument contains a `:` (colon), it will be considered a name/value pair (e.g. `User-Agent:Mozilla/*` or `Host:localhost`). If the value contains a colon, the value portion should be a regular expression. If the value does not contain a colon, the entire value will be considered to be the header name (e.g. `If-Modified-Since`). If the value evaluates to a header name only without a value, the header specified by the name must be present in the request for this predicate to be true. If the value evaluates to a header name/value pair, the header specified by the name must be present in the request *and* the regular expression specified as the value must match the header value. Whether or not the value represents a header name or a header name/value pair, the case of the header name is not significant. If this predicate returns `False`, route matching continues.

accept

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

custom_predicates

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

View-Related Arguments

view

A Python object or *dotted Python name* to the same object that will be used as a view callable when this route matches. e.g. `mypackage.views.my_view`.

view_context

A class or an *interface* or *dotted Python name* to the same object which the *context* of the view should match for the view named by the route to be used. This argument is only useful if the `view` attribute is used. If this attribute is not specified, the default (`None`) will be used.

If the `view` argument is not provided, this argument has no effect.

This attribute can also be spelled as `for_` or `view_for`.

view_permission

The permission name required to invoke the view associated with this route. e.g. `edit`. (see *Using Pyramid Security With URL Dispatch* for more information about permissions).

If the `view` attribute is not provided, this argument has no effect.

This argument can also be spelled as `permission`.

view_renderer

This is either a single string term (e.g. `json`) or a string implying a path or *asset specification* (e.g. `templates/views.pt`). If the `renderer` value is a single term (does not contain a dot `.`), the specified term will be used to look up a renderer implementation, and that renderer implementation will be used to construct a response from the view return value. If the `renderer` term contains a dot (`.`), the specified term will be treated as a path, and the filename extension of the last element in the path will be used to look up the renderer implementation, which will be passed the full path. The renderer implementation will be used to construct a response from the view return value. See *Writing View Callables Which Use a Renderer* for more information.

If the `view` argument is not provided, this argument has no effect.

This argument can also be spelled as `renderer`.

view_attr

The view machinery defaults to using the `__call__` method of the view callable (or the function itself, if the view callable is a function) to obtain a response dictionary. The `attr` value allows you to vary the method attribute used to obtain the response. For example, if your view was a class, and the class has a method named `index` and you wanted to use this method instead of the class' `__call__` method to return the response, you'd say `attr="index"` in the view configuration for the view. This is most useful when the view definition is a class.

If the `view` argument is not provided, this argument has no effect.

use_global_views

When a request matches this route, and view lookup cannot find a view which has a `route_name` predicate argument that matches the route, try to fall back to using a view that otherwise matches the context, request, and view name (but which does not match the `route_name` predicate).

add_static_view (*name*, *path*, ***kw*)

Add a view used to render static assets such as images and CSS files.

The `name` argument is a string representing an application-relative local URL prefix. It may alternately be a full URL.

The `path` argument is the path on disk where the static files reside. This can be an absolute path, a package-relative path, or a *asset specification*.

The `cache_max_age` keyword argument is input to set the `Expires` and `Cache-Control` headers for static assets served. Note that this argument has no effect when the `name` is a *url prefix*. By default, this argument is `None`, meaning that no particular `Expires` or `Cache-Control` headers are set in the response.

The `permission` keyword argument is used to specify the *permission* required by a user to execute the static view. By default, it is the string `__no_permission_required__`. The `__no_permission_required__` string is a special sentinel which indicates that, even if a *default permission* exists for the current application, the static view should be rendered to completely anonymous users. This default value is permissive because, in most web apps, static assets seldom need protection from viewing.

Usage

The `add_static_view` function is typically used in conjunction with the `pyramid.url.static_url()` function. `add_static_view` adds a view which renders a static asset when some URL is visited; `pyramid.url.static_url()` generates a URL to that asset.

The `name` argument to `add_static_view` is usually a *view name*. When this is the case, the `pyramid.url.static_url()` API will generate a URL which points to a Pyramid view, which will serve up a set of assets that live in the package itself. For example:

```
add_static_view('images', 'mypackage:images/')
```

Code that registers such a view can generate URLs to the view via `pyramid.url.static_url()`:

```
static_url('mypackage:images/logo.png', request)
```

When `add_static_view` is called with a `name` argument that represents a URL prefix, as it is above, subsequent calls to `pyramid.url.static_url()` with paths that start with the `path` argument passed to `add_static_view` will generate a URL something like `http://<Pyramid app URL>/images/logo.png`, which will cause the `logo.png` file in the `images` subdirectory of the `mypackage` package to be served.

`add_static_view` can alternately be used with a `name` argument which is a *URL*, causing static assets to be served from an external webserver. This happens when the `name` argument is a fully qualified URL (e.g. starts with `http://` or similar). In this mode, the `name` is used as the prefix of the full URL when generating a URL using `pyramid.url.static_url()`. For example, if `add_static_view` is called like so:

```
add_static_view('http://example.com/images', 'mypackage:images/')
```

Subsequently, the URLs generated by `pyramid.url.static_url()` for that static view will be prefixed with `http://example.com/images`:

```
static_url('mypackage:images/logo.png', request)
```

When `add_static_view` is called with a `name` argument that is the URL `http://example.com/images`, subsequent calls to `pyramid.url.static_url()` with paths that start with the `path` argument passed to `add_static_view` will generate a URL something like `http://example.com/logo.png`. The external webserver listening on `example.com` must be itself configured to respond properly to such a request.

See *Serving Static Assets* for more information.

add_settings (*settings=None, **kw*)

Augment the `settings` argument passed in to the Configurator constructor with one or more 'setting' key/value pairs. A setting is a single key/value pair in the dictionary-ish object returned from the API `pyramid.registry.Registry.settings` and `pyramid.config.Configurator.get_settings()`.

You may pass a dictionary:

```
config.add_settings({'external_uri': 'http://example.com'})
```

Or a set of key/value pairs:

```
config.add_settings(external_uri='http://example.com')
```

This function is useful when you need to test code that accesses the `pyramid.registry.Registry.settings` API (or the `pyramid.config.Configurator.get_settings()` API) and which uses values from that API.

add_subscriber (*subscriber, iface=None*)

Add an event *subscriber* for the event stream implied by the supplied *iface* interface. The *subscriber* argument represents a callable object (or a *dotted Python name* which identifies a callable); it will be called with a single object *event* whenever Pyramid emits an *event* associated with the *iface*, which may be an *interface* or a class or a *dotted Python name* to a global object representing an interface or a class. Using the default *iface* value, `None` will cause the subscriber to be registered for all event types. See *Using Events* for more information about events and subscribers.

add_translation_dirs (**specs*)

Add one or more *translation directory* paths to the current configuration state. The *specs* argument is a sequence that may contain absolute directory paths (e.g. `/usr/share/locale`) or *asset specification* names naming a directory path (e.g. `some.package:locale`) or a combination of the two.

Example:

```
config.add_translation_dirs('/usr/share/locale',
                             'some.package:locale')
```

add_view(*view=None, name='', for_=None, permission=None, request_type=None, route_name=None, request_method=None, request_param=None, containment=None, attr=None, renderer=None, wrapper=None, xhr=False, accept=None, header=None, path_info=None, custom_predicates=(), context=None, decorator=None, mapper=None*)

Add a *view configuration* to the current configuration state. Arguments to `add_view` are broken down below into *predicate* arguments and *non-predicate* arguments. Predicate arguments narrow the circumstances in which the view callable will be invoked when a request is presented to Pyramid; non-predicate arguments are informational.

Non-Predicate Arguments

view

A *view callable* or a *dotted Python name* which refers to a view callable. This argument is required unless a `renderer` argument also exists. If a `renderer` argument is passed, and a `view` argument is not provided, the view callable defaults to a callable that returns an empty dictionary (see *Writing View Callables Which Use a Renderer*).

permission

The name of a *permission* that the user must possess in order to invoke the *view callable*. See *Configuring View Security* for more information about view security and permissions. If `permission` is omitted, a *default* permission may be used for this view registration if one was named as the `pyramid.config.Configurator` constructor's `default_permission` argument, or if `pyramid.config.Configurator.set_default_permission()` was used prior to this view registration. Pass the string `__no_permission_required__` as the `permission` argument to explicitly indicate that the view should always be executable by entirely anonymous users, regardless of the default permission, bypassing any *authorization policy* that may be in effect.

attr

The view machinery defaults to using the `__call__` method of the *view callable* (or the function itself, if the view callable is a function) to obtain a response. The `attr` value allows you to vary the method attribute used to obtain the response. For example, if your view was a class, and the class has a method named `index` and you wanted to use this method instead of the class' `__call__` method to return the response, you'd say `attr="index"` in the view configuration for the view. This is most useful when the view definition is a class.

renderer

This is either a single string term (e.g. `json`) or a string implying a path or *asset specification* (e.g. `templates/views.pt`) naming a *renderer* implementation. If the `renderer` value does not contain a dot `.`, the specified string will be used to look up a renderer implementation, and that renderer implementation will be used to construct a response from the view return value. If the `renderer` value contains a dot (`.`), the specified term will be treated as a path, and the filename extension of the last element in the path will be used to look up the renderer implementation, which will be passed the full path. The renderer implementation will be used to construct a *response* from the view return value.

Note that if the view itself returns a *response* (see *View Callable Responses*), the specified renderer implementation is never called.

When the `renderer` is a path, although a path is usually just a simple relative pathname (e.g. `templates/foo.pt`, implying that a template named “foo.pt” is in the “templates” directory relative to the directory of the current *package* of the Configurator), a path can be absolute, starting with a slash on UNIX or a drive letter prefix on Windows. The path can alternately be a *asset specification* in the form `some.dotted.package_name:relative/path`, making it possible to address template assets which live in a separate package.

The `renderer` attribute is optional. If it is not defined, the “null” renderer is assumed (no rendering is performed and the value is passed back to the upstream Pyramid machinery unmolested).

wrapper

The *view name* of a different *view configuration* which will receive the response body of this view as the `request.wrapped_body` attribute of its own *request*, and the *response* returned by this view as the `request.wrapped_response` attribute of its own request. Using a wrapper makes it possible to “chain” views together to form a composite response. The response of the outermost wrapper view will be returned to the user. The wrapper view will be found as any view is found: see *View Lookup and Invocation*. The “best” wrapper view will be found based on the lookup ordering: “under the hood” this wrapper view is looked up via `pyramid.view.render_view_to_response(context, request, 'wrapper_viewname')`. The context and request of a wrapper view is the same context and request of the inner view. If this attribute is unspecified, no view wrapping is done.

decorator

A *dotted Python name* to function (or the function itself) which will be used to decorate the registered *view callable*. The decorator function will be called with the view callable as a single argument. The view callable it is passed will accept (`context`, `request`). The decorator must return a replacement view callable which also accepts (`context`, `request`).

mapper

A Python object or *dotted Python name* which refers to a *view mapper*, or `None`. By default it is `None`, which indicates that the view should use the default view mapper. This plug-point is useful for Pyramid extension developers, but it's not very useful for 'civilians' who are just developing stock Pyramid applications. Pay no attention to the man behind the curtain.

Predicate Arguments

name

The *view name*. Read *Traversal* to understand the concept of a view name.

context

An object or a *dotted Python name* referring to an interface or class object that the *context* must be an instance of, or the *interface* that the *context* must provide in order for this view to be found and called. This predicate is true when the *context* is an instance of the represented class or if the *context* provides the represented interface; it is otherwise false. This argument may also be provided to `add_view` as `for_` (an older, still-supported spelling).

route_name

This value must match the *name* of a *route configuration* declaration (see *URL Dispatch*) that must match before this view will be called. Note that the *route configuration* referred to by `route_name` usually has a `*traverse` token in the value of its `path`, representing a part of the path that will be used by *traversal* against the result of the route's *root factory*.



Using this argument services an advanced feature that isn't often used unless you want to perform traversal *after* a route has matched. See *Combining Traversal and URL Dispatch* for more information on using this advanced feature.

request_type

This value should be an *interface* that the *request* must provide in order for this view to be found and called. This value exists only for backwards compatibility purposes.

request_method

This value can either be one of the strings GET, POST, PUT, DELETE, or HEAD representing an HTTP REQUEST_METHOD. A view declaration with this argument ensures that the view will only be called when the request's method attribute (aka the REQUEST_METHOD of the WSGI environment) string matches the supplied value.

request_param

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

containment

This value should be a Python class or *interface* or a *dotted Python name* to such an object that a parent object in the *lineage* must provide in order for this view to be found and called. The nodes in your object graph must be “location-aware” to use this feature. See *Location-Aware Resources* for more information about location-awareness.

xhr

This value should be either True or False. If this value is specified and is True, the *request* must possess an HTTP_X_REQUESTED_WITH (aka X-Requested-With) header that has the value XMLHttpRequest for this view to be found and called. This is useful for detecting AJAX requests issued from jQuery, Prototype and other Javascript libraries.

accept

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

header

This value represents an HTTP header name or a header name/value pair. If the value contains a `:` (colon), it will be considered a name/value pair (e.g. `User-Agent:Mozilla/.*` or `Host:localhost`). The value portion should be a regular expression. If the value does not contain a colon, the entire value will be considered to be the header name (e.g. `If-Modified-Since`). If the value evaluates to a header name only without a value, the header specified by the name must be present in the request for this predicate to be true. If the value evaluates to a header name/value pair, the header specified by the name must be present in the request *and* the regular expression specified as the value must match the header value. Whether or not the value represents a header name or a header name/value pair, the case of the header name is not significant.

path_info

This value represents a regular expression pattern that will be tested against the `PATH_INFO` WSGI environment variable. If the regex matches, this predicate will be `True`.

custom_predicates

This value should be a sequence of references to custom predicate callables. Use custom predicates when no set of predefined predicates do what you need. Custom predicates can be combined with predefined predicates as necessary. Each custom predicate callable should accept two arguments: `context` and `request` and should return either `True` or `False` after doing arbitrary evaluation of the context and/or the request. If all callables return `True`, the associated view callable will be considered viable for a given request.

derive_view (*view*, *attr=None*, *renderer=None*)

Create a *view callable* using the function, instance, or class (or *dotted Python name* referring to the same) provided as `view` object.

This API is useful to framework extenders who create pluggable systems which need to register ‘proxy’ view callables for functions, instances, or classes which meet the requirements of being a Pyramid view callable. For example, a `some_other_framework` function in another framework may want to allow a user to supply a view callable, but he may want to wrap the view callable in his own before registering the wrapper as a Pyramid view callable. Because a Pyramid view callable can be any of a number of valid objects, the framework extender will not know how to call the user-supplied object. Running it through `derive_view` normalizes it to a callable which accepts two arguments: `context` and `request`.

For example:

```
def some_other_framework(user_supplied_view):
    config = Configurator(reg)
    proxy_view = config.derive_view(user_supplied_view)
    def my_wrapper(context, request):
        do_something_that_mutates(request)
        return proxy_view(context, request)
    config.add_view(my_wrapper)
```

The `view` object provided should be one of the following:

- A function or another non-class callable object that accepts a *request* as a single positional argument and which returns a *response* object.
- A function or other non-class callable object that accepts two positional arguments, `context`, `request` and which returns a *response* object.
- A class which accepts a single positional argument in its constructor named `request`, and which has a `__call__` method that accepts no arguments that returns a *response* object.
- A class which accepts two positional arguments named `context`, `request`, and which has a `__call__` method that accepts no arguments that returns a *response* object.
- A *dotted Python name* which refers to any of the kinds of objects above.

This API returns a callable which accepts the arguments `context`, `request` and which returns the result of calling the provided `view` object.

The `attr` keyword argument is most useful when the view object is a class. It names the method that should be used as the callable. If `attr` is not provided, the attribute effectively defaults to `__call__`. See *Defining a View Callable as a Class* for more information.

The `renderer` keyword argument should be a renderer name. If supplied, it will cause the returned callable to use a *renderer* to convert the user-supplied view result to a *response* object. If a `renderer` argument is not supplied, the user-supplied view must itself return a *response* object.

make_wsgi_app()

Commits any pending configuration statements, sends a `pyramid.events.ApplicationCreated` event to all listeners, and returns a Pyramid WSGI application representing the committed configuration state.

override_asset (*to_override, override_with, _override=None*)

Add a Pyramid asset override to the current configuration state.

to_override is a *asset specification* to the asset being overridden.

override_with is a *asset specification* to the asset that is performing the override.

See *Static Assets* for more information about asset overrides.

scan (*package=None, categories=None*)

Scan a Python package and any of its subpackages for objects marked with *configuration decoration* such as `pyramid.view.view_config`. Any decorated object found will influence the current configuration state.

The *package* argument should be a Python *package* or module object (or a *dotted Python name* which refers to such a package or module). If *package* is `None`, the package of the *caller* is used.

The *categories* argument, if provided, should be the *Venusian* ‘scan categories’ to use during scanning. Providing this argument is not often necessary; specifying scan categories is an extremely advanced usage. By default, *categories* is `None` which will execute *all* *Venusian* decorator callbacks including Pyramid-related decorators such as `pyramid.view.view_config`. See the *Venusian* documentation for more information about limiting a scan by using an explicit set of categories.

set_forbidden_view (*view=None, attr=None, renderer=None, wrapper=None*)

Add a default forbidden view to the current configuration state.



This method has been deprecated in Pyramid 1.0. *Do not use it for new development; it should only be used to support older code bases which depend upon it.* See *Changing the Forbidden View* to see how a forbidden view should be registered in new projects.

The *view* argument should be a *view callable* or a *dotted Python name* which refers to a view callable.


The *attr* argument should be the attribute of the view callable used to retrieve the response (see the `add_view` method’s *attr* argument for a description).

The *renderer* argument should be the name of (or path to) a *renderer* used to generate a response for this view (see the `pyramid.config.Configurator.add_view()` method’s *renderer* argument for information about how a configurator relates to a *renderer*).

The *wrapper* argument should be the name of another view which will wrap this view when rendered (see the `add_view` method’s *wrapper* argument for a description).

set_not_found_view (*view=None, attr=None, renderer=None, wrapper=None*)

Add a default not found view to the current configuration state.

 This method has been deprecated in Pyramid 1.0. *Do not use it for new development; it should only be used to support older code bases which depend upon it. See Changing the Not Found View* to see how a not found view should be registered in new projects.

The `view` argument should be a *view callable* or a *dotted Python name* which refers to a view callable.

The `attr` argument should be the attribute of the view callable used to retrieve the response (see the `add_view` method's `attr` argument for a description).


The `renderer` argument should be the name of (or path to) a *renderer* used to generate a response for this view (see the `pyramid.config.Configurator.add_view()` method's `renderer` argument for information about how a configurator relates to a *renderer*).

The `wrapper` argument should be the name of another view which will wrap this view when rendered (see the `add_view` method's `wrapper` argument for a description).

set_locale_negotiator (*negotiator*)

Set the *locale negotiator* for this application. The *locale negotiator* is a callable which accepts a *request* object and which returns a *locale name*. The `negotiator` argument should be the locale negotiator implementation or a *dotted Python name* which refers to such an implementation.

Later calls to this method override earlier calls; there can be only one locale negotiator active at a time within an application. See *Activating Translation* for more information.


 Using the `locale_negotiator` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_default_permission (*permission*)


Set the default permission to be used by all subsequent *view configuration* registrations. `permission` should be a *permission* string to be used as the default permission. An example of a permission string: `'view'`. Adding a default permission makes it unnecessary to protect each view configuration with an explicit permission, unless your application policy requires some exception for a particular view.

If a default permission is *not* set, views represented by view configuration registrations which do not explicitly declare a permission will be executable by entirely anonymous users (any authorization policy is ignored).

Later calls to this method override will conflict with earlier calls; there can be only one default permission active at a time within an application.

 If a default permission is in effect, view configurations meant to create a truly anonymously accessible view (even *exception view* views) *must* use the explicit permission string `__no_permission_required__` as the permission. When this string is used as the `permission` for a view configuration, the default permission is ignored, and the view is registered, making it available to all callers regardless of their credentials.

See also *Setting a Default Permission*.


 Using the `default_permission` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_session_factory (*session_factory*)

Configure the application with a *session factory*. If this method is called, the `session_factory` argument must be a session factory callable.

set_request_factory (*factory*)

The object passed as `factory` should be an object (or a *dotted Python name* which refers to an object) which will be used by the Pyramid router to create all request objects. This factory object must have the same methods and attributes as the `pyramid.request.Request` class (particularly `__call__`, and `blank`).

 Using the `:meth:request_factory` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_renderer_globals_factory (*factory*)

The object passed as `factory` should be a callable (or a *dotted Python name* which refers to a callable) that will be used by the Pyramid rendering machinery as a renderers global factory (see *Adding Renderer Globals*).

The factory callable must accept a single argument named `system` (which will be a dictionary) and it must return a dictionary. When an application uses a renderer, the factory's return dictionary will be merged into the `system` dictionary, and therefore will be made available to the code which uses the renderer.



Using the `renderer_globals_factory()` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

testing_securitypolicy (*userid=None, groupids=(), permissive=True*)

Unit/integration testing helper: Registers a pair of faux Pyramid security policies: a *authentication policy* and a *authorization policy*.

The behavior of the registered *authorization policy* depends on the `permissive` argument. If `permissive` is true, a permissive *authorization policy* is registered; this policy allows all access. If `permissive` is false, a nonpermissive *authorization policy* is registered; this policy denies all access.

The behavior of the registered *authentication policy* depends on the values provided for the `userid` and `groupids` argument. The authentication policy will return the `userid` identifier implied by the `userid` argument and the group ids implied by the `groupids` argument when the `pyramid.security.authenticated_userid()` or `pyramid.security.effective_principals()` APIs are used.

This function is most useful when testing code that uses the APIs named `pyramid.security.has_permission()`, `pyramid.security.authenticated_userid()`, `pyramid.security.effective_principals()` and `pyramid.security.principals_allowed_by_permission()`.

testing_resources (*resources*)

Unit/integration testing helper: registers a dictionary of *resource* objects that can be resolved via the `pyramid.traversal.find_resource()` API.

The `pyramid.traversal.find_resource()` API is called with a path as one of its arguments. If the dictionary you register when calling this method contains that path as a string key (e.g. `/foo/bar` or `foo/bar`), the corresponding value will be returned to `find_resource` (and thus to your code) when `pyramid.traversal.find_resource()` is called with an equivalent path string or tuple.

testing_add_subscriber (*event_iface=None*)

Unit/integration testing helper: Registers a *subscriber* which listens for events of the type *event_iface*. This method returns a list object which is appended to by the subscriber whenever an event is captured.

When an event is dispatched that matches the value implied by the *event_iface* argument, that event will be appended to the list. You can then compare the values in the list to expected event notifications. This method is useful when testing code that wants to call `pyramid.registry.Registry.notify()`, or `zope.component.event.dispatch()`.

The default value of *event_iface* (`None`) implies a subscriber registered for *any* kind of event.

testing_add_renderer (*path, renderer=None*)

Unit/integration testing helper: register a renderer at *path* (usually a relative filename ala `templates/foo.pt` or an asset specification) and return the renderer object. If the *renderer* argument is `None`, a ‘dummy’ renderer will be used. This function is useful when testing code that calls the `pyramid.renderers.render()` function or `pyramid.renderers.render_to_response()` function or any other `render_*` or `get_*` API of the `pyramid.renderers` module.

Note that calling this method for with a *path* argument representing a renderer factory type (e.g. for `foo.pt` usually implies the `chameleon_zpt` renderer factory) clobbers any existing renderer factory registered for that type.



This method is also available under the alias `testing_add_template` (an older name for it).

PYRAMID . EVENTS

40.1 Functions

subscriber (*ifaces)

Decorator activated via a *scan* which treats the function being decorated as an event subscriber for the set of interfaces passed as **ifaces* to the decorator constructor.

For example:

```
from pyramid.events import NewRequest
from pyramid.events import subscriber

@subscriber(NewRequest)
def mysubscriber(event):
    event.request.foo = 1
```

More than one event type can be passed as a constructor argument:

```
from pyramid.events import NewRequest, NewResponse
from pyramid.events import subscriber

@subscriber(NewRequest, NewResponse)
def mysubscriber(event):
    print event
```

When the `subscriber` decorator is used without passing an arguments, the function it decorates is called for every event sent:

```
from pyramid.events import subscriber

@subscriber()
def mysubscriber(event):
    print event
```

This method will have no effect until a *scan* is performed against the package or module which contains it, ala:

```
from pyramid.config import Configurator
config = Configurator()
config.scan('somepackage_containing_subscribers')
```

40.2 Event Types

class ApplicationCreated (*app*)

An instance of this class is emitted as an *event* when the `pyramid.config.Configurator.make_wsgi_app()` is called. The instance has an attribute, `app`, which is an instance of the *router* that will handle WSGI requests. This class implements the `pyramid.interfaces.IApplicationCreated` interface.



For backwards compatibility purposes, this class can also be imported as `pyramid.events.WSGIApplicationCreatedEvent`. This was the name of the event class before Pyramid 1.0.

class NewRequest (*request*)

An instance of this class is emitted as an *event* whenever Pyramid begins to process a new request. The even instance has an attribute, `request`, which is a *request* object. This event class implements the `pyramid.interfaces.INewRequest` interface.

class ContextFound (*request*)

An instance of this class is emitted as an *event* after the Pyramid *router* finds a *context* object (after it performs traversal) but before any view code is executed. The instance has an attribute, `request`, which is the request object generated by Pyramid.

Notably, the request object will have an attribute named `context`, which is the context that will be provided to the view which will eventually be called, as well as other attributes attached by context-finding code.

This class implements the `pyramid.interfaces.IContextFound` interface.

i As of Pyramid 1.0, for backwards compatibility purposes, this event may also be imported as `pyramid.events.AfterTraversal`.

class NewResponse (*request, response*)

An instance of this class is emitted as an *event* whenever any Pyramid *view* or *exception view* returns a *response*.

The instance has two attributes: `request`, which is the request which caused the response, and `response`, which is the response object returned by a view or renderer.

If the response was generated by an *exception view*, the request will have an attribute named `exception`, which is the exception object which caused the exception view to be executed. If the response was generated by a ‘normal’ view, the request will not have this attribute.

This event will not be generated if a response cannot be created due to an exception that is not caught by an exception view (no response is created under this circumstance).

This class implements the `pyramid.interfaces.INewResponse` interface.

i Postprocessing a response is usually better handled in a WSGI *middleware* component than in subscriber code that is called by a `pyramid.interfaces.INewResponse` event. The `pyramid.interfaces.INewResponse` event exists almost purely for symmetry with the `pyramid.interfaces.INewRequest` event.

class BeforeRender (*system*)

get (*k, default=None*)

Return the value for key *k* from the renderer globals dictionary, or the default if no such value exists.

update (*d*)

Update the renderer globals dictionary with another dictionary *d*. If any of the key names in the source dictionary already exist in the target dictionary, a `KeyError` will be raised

See *Using Events* for more information about how to register code which subscribes to these events.

PYRAMID . EXCEPTIONS

class Forbidden (*message*='')

Raise this exception within *view* code to immediately return the *forbidden view* to the invoking user. Usually this is a basic 403 page, but the forbidden view can be customized as necessary. See *Changing the Forbidden View*.

This exception's constructor accepts a single positional argument, which should be a string. The value of this string will be placed onto the request by the router as the `exception_message` attribute, for availability to the *Forbidden View*.

class NotFound (*message*='')

Raise this exception within *view* code to immediately return the *Not Found view* to the invoking user. Usually this is a basic 404 page, but the Not Found view can be customized as necessary. See *Changing the Not Found View*.

This exception's constructor accepts a single positional argument, which should be a string. The value of this string will be placed into the WSGI environment by the router as the `exception_message` attribute, for availability to the *Not Found View*.

class ConfigurationError

Raised when inappropriate input values are supplied to an API method of a *Configurator*

class URLDecodeError

This exception is raised when Pyramid cannot successfully decode a URL or a URL path segment. This exception it behaves just like the Python builtin `UnicodeDecodeError`. It is a subclass of the builtin `UnicodeDecodeError` exception only for identity purposes, mostly so an exception view can be registered when a URL cannot be decoded.

PYRAMID . HTTPEXCEPTIONS

42.1 HTTP Exception

This module processes Python exceptions that relate to HTTP exceptions by defining a set of exceptions, all subclasses of `HTTPException`. Each exception, in addition to being a Python exception that can be raised and caught, is also a WSGI application and `webob.Response` object.

This module defines exceptions according to RFC 2068¹: codes with 100-300 are not really errors; 400's are client errors, and 500's are server errors. According to the WSGI specification², the application can call `start_response` more than once only under two conditions: (a) the response has not yet been sent, or (b) if the second and subsequent invocations of `start_response` have a valid `exc_info` argument obtained from `sys.exc_info()`. The WSGI specification then requires the server or gateway to handle the case where content has been sent and then an exception was encountered.

Exception

HTTPException

HTTPOk

- 200 - HTTPOk
- 201 - HTTPCreated
- 202 - HTTPAccepted

¹ <http://www.python.org/peps/pep-0333.html#error-handling>

² <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5>

- 203 - HTTPNonAuthoritativeInformation
- 204 - HTTPNoContent
- 205 - HTTPResetContent
- 206 - HTTPPartialContent

HTTPRedirection

- 300 - HTTPMultipleChoices
- 301 - HTTPMovedPermanently
- 302 - HTTPFound
- 303 - HTTPSeeOther
- 304 - HTTPNotModified
- 305 - HTTPUseProxy
- 306 - Unused (not implemented, obviously)
- 307 - HTTPTemporaryRedirect

HTTPError

HTTPClientError

- 400 - HTTPBadRequest
- 401 - HTTPUnauthorized
- 402 - HTTPPaymentRequired
- 403 - HTTPForbidden
- 404 - HTTPNotFound
- 405 - HTTPMethodNotAllowed
- 406 - HTTPNotAcceptable

- 407 - HTTPProxyAuthenticationRequired
- 408 - HTTPRequestTimeout
- 409 - HTTPConflict
- 410 - HTTPGone
- 411 - HTTPLengthRequired
- 412 - HTTPPreconditionFailed
- 413 - HTTPRequestEntityTooLarge
- 414 - HTTPRequestURITooLong
- 415 - HTTPUnsupportedMediaType
- 416 - HTTPRequestRangeNotSatisfiable
- 417 - HTTPExpectationFailed

HTTPServerError

- 500 - HTTPInternalServerError
- 501 - HTTPNotImplemented
- 502 - HTTPBadGateway
- 503 - HTTPServiceUnavailable
- 504 - HTTPGatewayTimeout
- 505 - HTTPVersionNotSupported

42.2 Subclass usage notes:

The HTTPException class is complicated by 4 factors:

1. The content given to the exception may either be plain-text or as html-text.
2. The template may want to have string-substitutions taken from the current `environ` or values from incoming headers. This is especially troublesome due to case sensitivity.
3. The final output may either be `text/plain` or `text/html` mime-type as requested by the client application.
4. Each exception has a default explanation, but those who raise exceptions may want to provide additional detail.

Subclass attributes and call parameters are designed to provide an easier path through the complications.

Attributes:

code the HTTP status code for the exception

title remainder of the status line (stuff after the code)

explanation a plain-text explanation of the error message that is not subject to environment or header substitutions; it is accessible in the template via `%(explanation)s`

detail a plain-text message customization that is not subject to environment or header substitutions; accessible in the template via `%(detail)s`

body_template a content fragment (in HTML) used for environment and header substitution; the default template includes both the explanation and further detail provided in the message

Parameters:

detail a plain-text override of the default `detail`

headers a list of (k,v) header pairs

comment a plain-text additional information which is usually stripped/hidden for end-users

body_template a string.Template object containing a content fragment in HTML that frames the explanation and further detail

To override the template (which is HTML content) or the plain-text explanation, one must subclass the given exception; or customize it after it has been created. This particular breakdown of a message into explanation, detail and template allows both the creation of plain-text and html messages for various clients as well as error-free substitution of environment variables and headers.

The subclasses of `_HTTPMove` (`HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPFound`, `HTTPSeeOther`, `HTTPUseProxy` and `HTTPTemporaryRedirect`) are redirections that require a `Location` field. Reflecting this, these subclasses have two additional keyword arguments: `location` and `add_slash`.

Parameters:

location to set the location immediately

add_slash set to `True` to redirect to the same URL as the request, except with a `/` appended

Relative URLs in the location will be resolved to absolute.

References:

status_map

A mapping of integer status code to exception class (eg. the integer “401” maps to `pyramid.httpexceptions.HTTPUnauthorized`).

class HTTPException (*message, wsgi_response*)

Exception used on pre-Python-2.5, where new-style classes cannot be used as an exception.

class HTTPOk (*detail=None, headers=None, comment=None, body_template=None, **kw*)

Base class for the 200’s status code (successful responses)

code: 200, title: OK

class HTTPRedirection (*detail=None, headers=None, comment=None, body_template=None, **kw*)

base class for 300’s status code (redirections)

This is an abstract base class for 3xx redirection. It indicates that further action needs to be taken by the user agent in order to fulfill the request. It does not necessarily signal an error condition.

class HTTPError (*detail=None, headers=None, comment=None, body_template=None, **kw*)

base class for status codes in the 400’s and 500’s

This is an exception which indicates that an error has occurred, and that any work in progress should not be committed. These are typically results in the 400’s and 500’s.

class HTTPClientError (*detail=None, headers=None, comment=None, body_template=None, **kw*)
base class for the 400's, where the client is in error

This is an error condition in which the client is presumed to be in-error. This is an expected problem, and thus is not considered a bug. A server-side traceback is not warranted. Unless specialized, this is a '400 Bad Request'

class HTTPServerError (*detail=None, headers=None, comment=None, body_template=None, **kw*)
base class for the 500's, where the server is in-error

This is an error condition in which the server is presumed to be in-error. This is usually unexpected, and thus requires a traceback; ideally, opening a support ticket for the customer. Unless specialized, this is a '500 Internal Server Error'

class HTTPCreated (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPOk

This indicates that request has been fulfilled and resulted in a new resource being created.

code: 201, title: Created

class HTTPAccepted (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPOk

This indicates that the request has been accepted for processing, but the processing has not been completed.

code: 202, title: Accepted

class HTTPNonAuthoritativeInformation (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPOk

This indicates that the returned meta-information in the entity-header is not the definitive set as available from the origin server, but is gathered from a local or a third-party copy.

code: 203, title: Non-Authoritative Information

class HTTPNoContent (*detail=None, headers=None, comment=None, body_template=None,*
***kw*)
subclass of HTTPOk

This indicates that the server has fulfilled the request but does not need to return an entity-body, and might want to return updated meta-information.

code: 204, title: No Content

class HTTPResetContent (*detail=None, headers=None, comment=None,*
*body_template=None, **kw*)
subclass of HTTPOk

This indicates that the the server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be sent.

code: 205, title: Reset Content

class HTTPPartialContent (*detail=None, headers=None, comment=None,*
*body_template=None, **kw*)
subclass of HTTPOk

This indicates that the server has fulfilled the partial GET request for the resource.

code: 206, title: Partial Content

class HTTPMultipleChoices (*detail=None, headers=None, comment=None,*
body_template=None, location=None, add_slash=False)
subclass of _HTTPMove

This indicates that the requested resource corresponds to any one of a set of representations, each with its own specific location, and agent-driven negotiation information is being provided so that the user can select a preferred representation and redirect its request to that location.

code: 300, title: Multiple Choices

class HTTPMovedPermanently (*detail=None, headers=None, comment=None,*
body_template=None, location=None, add_slash=False)
subclass of _HTTPMove

This indicates that the requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs.

code: 301, title: Moved Permanently

class HTTPFound (*detail=None, headers=None, comment=None, body_template=None, location=None, add_slash=False*)
subclass of `_HTTPMove`

This indicates that the requested resource resides temporarily under a different URI.

code: 302, title: Found

class HTTPSeeOther (*detail=None, headers=None, comment=None, body_template=None, location=None, add_slash=False*)
subclass of `_HTTPMove`

This indicates that the response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource.

code: 303, title: See Other

class HTTPNotModified (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of `HTTPRedirection`

This indicates that if the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server SHOULD respond with this status code.

code: 304, title: Not Modified

class HTTPUseProxy (*detail=None, headers=None, comment=None, body_template=None, location=None, add_slash=False*)
subclass of `_HTTPMove`

This indicates that the requested resource MUST be accessed through the proxy given by the Location field.

code: 305, title: Use Proxy

class HTTPTemporaryRedirect (*detail=None, headers=None, comment=None, body_template=None, location=None, add_slash=False*)
subclass of `_HTTPMove`

This indicates that the requested resource resides temporarily under a different URI.

code: 307, title: Temporary Redirect

class HTTPBadRequest (*detail=None, headers=None, comment=None, body_template=None, **kw*)

```
class HTTPUnauthorized (detail=None, headers=None, comment=None,  
                        body_template=None, **kw)  
    subclass of HTTPClientError
```

This indicates that the request requires user authentication.

code: 401, title: Unauthorized

```
class HTTPPaymentRequired (detail=None, headers=None, comment=None,  
                            body_template=None, **kw)  
    subclass of HTTPClientError
```

code: 402, title: Payment Required

```
class HTTPForbidden (detail=None, headers=None, comment=None, body_template=None,  
                     **kw)  
    subclass of HTTPClientError
```

This indicates that the server understood the request, but is refusing to fulfill it.

code: 403, title: Forbidden

```
class HTTPNotFound (detail=None, headers=None, comment=None, body_template=None,  
                    **kw)  
    subclass of HTTPClientError
```

This indicates that the server did not find anything matching the Request-URI.

code: 404, title: Not Found

```
class HTTPMethodNotAllowed (detail=None, headers=None, comment=None,  
                             body_template=None, **kw)  
    subclass of HTTPClientError
```

This indicates that the method specified in the Request-Line is not allowed for the resource identified by the Request-URI.

code: 405, title: Method Not Allowed

```
class HTTPNotAcceptable (detail=None, headers=None, comment=None,  
                          body_template=None, **kw)  
    subclass of HTTPClientError
```

This indicates the resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

code: 406, title: Not Acceptable

class HTTPProxyAuthenticationRequired (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This is similar to 401, but indicates that the client must first authenticate itself with the proxy.

code: 407, title: Proxy Authentication Required

class HTTPRequestTimeout (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the client did not produce a request within the time that the server was prepared to wait.

code: 408, title: Request Timeout

class HTTPConflict (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the request could not be completed due to a conflict with the current state of the resource.

code: 409, title: Conflict

class HTTPGone (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the requested resource is no longer available at the server and no forwarding address is known.

code: 410, title: Gone

class HTTPLengthRequired (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the the server refuses to accept the request without a defined Content-Length.

code: 411, title: Length Required

class HTTPPreconditionFailed (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the precondition given in one or more of the request-header fields evaluated to false when it was tested on the server.

code: 412, title: Precondition Failed

class HTTPRequestEntityTooLarge (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the server is refusing to process a request because the request entity is larger than the server is willing or able to process.

code: 413, title: Request Entity Too Large

class HTTPRequestURITooLong (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the server is refusing to service the request because the Request-URI is longer than the server is willing to interpret.

code: 414, title: Request-URI Too Long

class HTTPUnsupportedMediaType (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

code: 415, title: Unsupported Media Type

class HTTPRequestRangeNotSatisfiable (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

The server SHOULD return a response with this status code if a request included a Range request-header field, and none of the range-specifier values in this field overlap the current extent of the selected resource, and the request did not include an If-Range request-header field.

code: 416, title: Request Range Not Satisfiable

class HTTPExpectationFailed (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPClientError

This indicates that the expectation given in an Expect request-header field could not be met by this server.

code: 417, title: Expectation Failed

class HTTPInternalServerError (*detail=None, headers=None, comment=None, body_template=None, **kw*)

class HTTPNotImplemented (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPServerError

This indicates that the server does not support the functionality required to fulfill the request.

code: 501, title: Not Implemented

class HTTPBadGateway (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPServerError

This indicates that the server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

code: 502, title: Bad Gateway

class HTTPServiceUnavailable (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPServerError

This indicates that the server is currently unable to handle the request due to a temporary overloading or maintenance of the server.

code: 503, title: Service Unavailable

class HTTPGatewayTimeout (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPServerError

This indicates that the server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (e.g. HTTP, FTP, LDAP) or some other auxiliary server (e.g. DNS) it needed to access in attempting to complete the request.

code: 504, title: Gateway Timeout

class HTTPVersionNotSupported (*detail=None, headers=None, comment=None, body_template=None, **kw*)
subclass of HTTPServerError

This indicates that the server does not support, or refuses to support, the HTTP protocol version that was used in the request message.

code: 505, title: HTTP Version Not Supported

PYRAMID . I18N

class `TranslationString`

The constructor for a *translation string*. A translation string is a Unicode-like object that has some extra metadata.

This constructor accepts one required argument named `msgid`. `msgid` must be the *message identifier* for the translation string. It must be a `unicode` object or a `str` object encoded in the default system encoding.

Optional keyword arguments to this object's constructor include `domain`, `default`, and `mapping`.

`domain` represents the *translation domain*. By default, the translation domain is `None`, indicating that this translation string is associated with the default translation domain (usually messages).

`default` represents an explicit *default text* for this translation string. Default text appears when the translation string cannot be translated. Usually, the `msgid` of a translation string serves double duty as its default text. However, using this option you can provide a different default text for this translation string. This feature is useful when the default of a translation string is too complicated or too long to be used as a message identifier. If `default` is provided, it must be a `unicode` object or a `str` object encoded in the default system encoding (usually means ASCII). If `default` is `None` (its default value), the `msgid` value used by this translation string will be assumed to be the value of `default`.

`mapping`, if supplied, must be a dictionary-like object which represents the replacement values for any *translation string replacement marker* instances found within the `msgid` (or `default`) value of this translation string.

After a translation string is constructed, it behaves like most other unicode objects; its `msgid` value will be displayed when it is treated like a unicode object. Only when its `ugettext` method is called will it be translated.

Its default value is available as the `default` attribute of the object, its *translation domain* is available as the `domain` attribute, and the mapping is available as the `mapping` attribute. The object otherwise behaves much like a Unicode string.

class TranslationStringFactory

Create a factory which will generate translation strings without requiring that each call to the factory be passed a `domain` value. A single argument is passed to this class' constructor: `domain`. This value will be used as the `domain` values of `translationstring.TranslationString` objects generated by the `__call__` of this class. The `msgid`, `mapping`, and `default` values provided to the `__call__` method of an instance of this class have the meaning as described by the constructor of the `translationstring.TranslationString`

class Localizer (*locale_name, translations*)

An object providing translation and pluralizations related to the current request's locale name. A `pyramid.i18n.Localizer` object is created using the `pyramid.i18n.get_localizer()` function.

locale_name

The locale name for this localizer (e.g. `en` or `en_US`).

pluralize (*singular, plural, n, domain=None, mapping=None*)

Return a Unicode string translation by using two *message identifier* objects as a singular/plural pair and an `n` value representing the number that appears in the message using `gettext` plural forms support. The `singular` and `plural` objects passed may be translation strings or unicode strings. `n` represents the number of elements. `domain` is the translation domain to use to do the pluralization, and `mapping` is the interpolation mapping that should be used on the result. Note that if the objects passed are translation strings, their domains and mappings are ignored. The `domain` and `mapping` arguments must be used instead. If the `domain` is not supplied, a default domain is used (usually `messages`).

Example:

```
num = 1
translated = localizer.pluralize('Add ${num} item',
                                'Add ${num} items',
                                num,
                                mapping={'num': num})
```

translate (*tstring*, *domain=None*, *mapping=None*)

Translate a *translation string* to the current language and interpolate any *replacement markers* in the result. The `translate` method accepts three arguments: `tstring` (required), `domain` (optional) and `mapping` (optional). When called, it will translate the `tstring` translation string to a unicode object using the current locale. If the current locale could not be determined, the result of interpolation of the default value is returned. The optional `domain` argument can be used to specify or override the domain of the `tstring` (useful when `tstring` is a normal string rather than a translation string). The optional `mapping` argument can specify or override the `tstring` interpolation mapping, useful when the `tstring` argument is a simple string instead of a translation string.

Example:

```
from pyramid.i18n import TranslationString
ts = TranslationString('Add ${item}', domain='mypackage',
                       mapping={'item': 'Item'})
translated = localizer.translate(ts)
```

Example:

```
translated = localizer.translate('Add ${item}', domain='mypackage',
                                mapping={'item': 'Item'})
```

get_localizer (*request*)

Retrieve a `pyramid.i18n.Localizer` object corresponding to the current request's locale name.

negotiate_locale_name (*request*)

Negotiate and return the *locale name* associated with the current request (never cached).

get_locale_name (*request*)

Return the *locale name* associated with the current request (possibly cached).

default_locale_negotiator (*request*)

The default *locale negotiator*. Returns a locale name or `None`.

- First, the negotiator looks for the `__LOCALE__` attribute of the request object (possibly set by a view or a listener for an *event*).
- Then it looks for the `request.params['__LOCALE__']` value.
- Then it looks for the `request.cookies['__LOCALE__']` value.
- Finally, the negotiator returns `None` if the locale could not be determined via any of the previous checks (when a locale negotiator returns `None`, it signifies that the *default locale name* should be used.)

See *Internationalization and Localization* for more information about using Pyramid internationalization and localization services within an application.

PYRAMID . INTERFACES

44.1 Event-Related Interfaces

interface IApplicationCreated

Event issued when the `pyramid.config.Configurator.make_wsgi_app()` method is called. See the documentation attached to `pyramid.events.ApplicationCreated` for more information.



For backwards compatibility with Pyramid versions before 1.0, this interface can also be imported as `pyramid.interfaces.IWSGIApplicationCreatedEvent`.

app

Created application

interface INewRequest


An event type that is emitted whenever Pyramid begins to process a new request. See the documentation attached to `pyramid.events.NewRequest` for more information.

request

The request object

interface IContextFound

An event type that is emitted after Pyramid finds a *context* object but before it calls any view code. See the documentation attached to `pyramid.events.ContextFound` for more information.

 For backwards compatibility with versions of Pyramid before 1.0, this event interface can also be imported as `pyramid.interfaces.IAfterTraversal`.

request

The request object

interface INewResponse

An event type that is emitted whenever any Pyramid view returns a response. See the documentation attached to `pyramid.events.NewResponse` for more information.

request

The request object

response

The response object

interface IBeforeRender

Subscribers to this event may introspect the and modify the set of *renderer globals* before they are passed to a *renderer*. This event object itself has a dictionary-like interface that can be used for this purpose. For example:

```
from repoze.events import subscriber
from pyramid.interfaces import IBeforeRender

@subscriber(IBeforeRender)
def add_global(event):
    event['mykey'] = 'foo'
```

See also *Using The Before Render Event*.

__getitem__(k)

Return the value for key *k* from the renderer globals dictionary.

__contains__(k)

Return `True` if *k* exists in the renderer globals dictionary.

get (*k*, *default=None*)

Return the value for key *k* from the renderer globals dictionary, or the default if no such value exists.

update (*d*)

Update the renderer globals dictionary with another dictionary *d*. If any of the key names in the source dictionary already exist in the target dictionary, a `KeyError` will be raised

__setitem__ (*name*, *value*)

Set a name/value pair into the dictionary which is passed to a renderer as the renderer globals dictionary. If the *name* already exists in the target dictionary, a `KeyError` will be raised.

44.2 Other Interfaces

interface `IExceptionResponse`

Extends: `pyramid.interfaces.IException`,
`pyramid.interfaces.IResponse`

An interface representing a WSGI response which is also an exception object. Register an exception view using this interface as a `context` to apply the registered view for all exception types raised by Pyramid internally (`pyramid.exceptions.NotFound` and `pyramid.exceptions.Forbidden`).

interface `IRoute`

Interface representing the type of object returned from `IRoutesMapper.get_route`

name

The route name

pattern

The route pattern

factory

The *root factory* used by the Pyramid router when this route matches (or `None`)

generate (*kw*)

Generate a URL based on filling in the dynamic segment markers in the pattern using the *kw* dictionary provided.

pregenerator

This attribute should either be `None` or a callable object implementing the `IRoutePregenerator` interface

predicates

A sequence of *route predicate* objects used to determine if a request matches this route or not or not after basic pattern matching has been completed.

match (*path*)

If the *path* passed to this function can be matched by the *pattern* of this route, return a dictionary (the 'matchdict'), which will contain keys representing the dynamic segment markers in the pattern mapped to values extracted from the provided *path*.

If the *path* passed to this function cannot be matched by the *pattern* of this route, return `None`.

interface IRoutePregenerator**__call__** (*request, elements, kw*)

A pregenerator is a function associated by a developer with a *route*. The pregenerator for a route is called by `pyramid.url.route_url()` in order to adjust the set of arguments passed to it by the user for special purposes, such as Pylons 'subdomain' support. It will influence the URL returned by `route_url`.

A pregenerator should return a two-tuple of (*elements, kw*) after examining the originals passed to this function, which are the arguments (*request, elements, kw*). The simplest pregenerator is:

```
def pregenerator(request, elements, kw):  
    return elements, kw
```

You can employ a pregenerator by passing a `pregenerator` argument to the `pyramid.config.Configurator.add_route()` function.

interface ISession

An interface representing a session (a web session object, usually accessed via `request.session`).

Keys and values of a session must be pickleable.

invalidate()

Invalidate the session. The action caused by `invalidate` is implementation-dependent, but it should have the effect of completely dissociating any data stored in the session with the current request. It might set response values (such as one which clears a cookie), or it might not.

new_csrf_token()

Create and set into the session a new, random cross-site request forgery protection token. Return the token. It will be a string.

pop(*k*, **args*)

remove specified key and return the corresponding value **args* may contain a single default value, or may not be supplied. If key is not found, default is returned if given, otherwise `KeyError` is raised

get_csrf_token()

Return a random cross-site request forgery protection token. It will be a string. If a token was previously added to the session via `new_csrf_token`, that token will be returned. If no CSRF token was previously set into the session, `new_csrf_token` will be called, which will create and set a token, and this token will be returned.

__contains__(*key*)

Return true if a key exists in the mapping.

flash(*msg*, *queue*='', *allow_duplicate*=True)

Push a flash message onto the end of the flash queue represented by `queue`. An alternate flash message queue can be used by passing an optional `queue`, which must be a string. If `allow_duplicate` is false, if the `msg` already exists in the queue, it will not be readded.

peek_flash(*queue*='')

Peek at a queue in the flash storage. The queue remains in flash storage after this message is called. The queue is returned; it is a list of flash messages added by `pyramid.interfaces.ISession.flash()`

itervalues()

iterate over values

new

Boolean attribute. If `True`, the session is new.

__len__()

Return the number of items in the session.

__getitem__ (*key*)

Get a value for a key

A `KeyError` is raised if there is no value for the key.

get (*key, default=None*)

Get a value for a key

The default is returned if there is no value for the key.

keys ()

Return the keys of the mapping object.

update (*d*)

Update D from E: for k in E.keys(): D[k] = E[k]

__setitem__ (*key, value*)

Set a new item in the mapping.

iteritems ()

iterate over items

popitem ()

remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if mapping is empty

iterkeys ()

iterate over keys; equivalent to `__iter__`

pop_flash (*queue=''*)

Pop a queue from the flash storage. The queue is removed from flash storage after this message is called. The queue is returned; it is a list of flash messages added by `pyramid.interfaces.ISession.flash()`

__delitem__ (*key*)

Delete a value from the mapping using the key.

A `KeyError` is raised if there is no value for the key.

setdefault (*key, default=None*)

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

created

Integer representing Epoch time when created.

items()

Return the items of the mapping object.

clear()

delete all items

changed()

Mark the session as changed. A user of a session should call this method after he or she mutates a mutable object that is *a value of the session* (it should not be required after mutating the session itself). For example, if the user has stored a dictionary in the session under the key `foo`, and he or she does `session['foo'] = {}`, `changed()` needn't be called. However, if subsequently he or she does `session['foo']['a'] = 1`, `changed()` must be called for the sessioning machinery to notice the mutation of the internal dictionary.

__iter__()

Return an iterator for the keys of the mapping object.

values()

Return the values of the mapping object.

interface ISessionFactory

An interface representing a factory which accepts a request object and returns an ISession object

__call__(request)

Return an ISession object

interface IRendererInfo

An object implementing this interface is passed to every *renderer factory* constructor as its only argument (conventionally named `info`)

name

The value passed by the user as the renderer name

package

The “current package” when the renderer configuration statement was found

settings

The deployment settings dictionary related to the current application

registry

The “current” application registry when the renderer was created

type

The renderer type name

interface ITemplateRenderer

Extends: `pyramid.interfaces.IRenderer`

implementation()

Return the object that the underlying templating system uses to render the template; it is typically a callable that accepts arbitrary keyword arguments and returns a string or unicode object

interface IViewMapperFactory**__call__(self, **kw)**

Return an object which implements `pyramid.interfaces.IViewMapper`. `kw` will be a dictionary containing view-specific arguments, such as `permission`, `predicates`, `attr`, `renderer`, and other items. An `IViewMapperFactory` is used by `pyramid.config.Configurator.add_view()` to provide a plug-point to extension developers who want to modify potential view callable invocation signatures and response values.

interface IViewMapper**__call__(self, object)**

Provided with an arbitrary object (a function, class, or instance), returns a callable with the call signature `(context, request)`. The callable returned should itself return a `Response` object. An `IViewMapper` is returned by `pyramid.interfaces.IViewMapperFactory`.

PYRAMID . LOCATION

lineage (*resource*)

Return a generator representing the *lineage* of the *resource* object implied by the *resource* argument. The generator first returns *resource* unconditionally. Then, if *resource* supplies a `__parent__` attribute, return the resource represented by `resource.__parent__`. If *that* resource has a `__parent__` attribute, return that resource's parent, and so on, until the resource being inspected either has no `__parent__` attribute or which has a `__parent__` attribute of `None`. For example, if the resource tree is:

```
thing1 = Thing()
thing2 = Thing()
thing2.__parent__ = thing1
```

Calling `lineage(thing2)` will return a generator. When we turn it into a list, we will get:

```
list(lineage(thing2))
[ <Thing object at thing2>, <Thing object at thing1> ]
```

inside (*resource1*, *resource2*)

Is *resource1* 'inside' *resource2*? Return `True` if so, else `False`.

resource1 is 'inside' *resource2* if *resource2* is a *lineage* ancestor of *resource1*. It is a lineage ancestor if its parent (or one of its parent's parents, etc.) is an ancestor.

PYRAMID . PASTER

get_app (*config_file*, *name*)

Return the WSGI application named *name* in the PasteDeploy config file *config_file*.

PYRAMID . REGISTRY

class Registry (*name='', bases=()*)

A registry object is an *application registry*. The existence of a registry implementation detail of pyramid. It is used by the framework itself to perform mappings of URLs to view callables, as well as servicing other various duties. Despite being an implementation detail of the framework, it has a number of attributes that may be useful within application code.

For information about the purpose and usage of the application registry, see *Using the Zope Component Architecture in Pyramid*.

The application registry is usually accessed as `request.registry` in application code.

settings

The dictionary-like *deployment settings* object. See *Deployment Settings* for information. This object is often accessed as `request.registry.settings` or `config.registry.settings` in a typical Pyramid application.

PYRAMID . RENDERERS

get_renderer (*renderer_name*, *package=None*)

Return the renderer object for the renderer named as *renderer_name*.

You may supply a relative asset spec as *renderer_name*. If the *package* argument is supplied, a relative renderer name will be converted to an absolute asset specification by combining the package supplied as *package* with the relative asset specification supplied as *renderer_name*. If you do not supply a package (or *package* is *None*) the package name of the *caller* of this function will be used as the package.

render (*renderer_name*, *value*, *request=None*, *package=None*)

Using the renderer specified as *renderer_name* (a template or a static renderer) render the value (or set of values) present in *value*. Return the result of the renderer's `__call__` method (usually a string or Unicode).

If the renderer name refers to a file on disk (such as when the renderer is a template), it's usually best to supply the name as a *asset specification* (e.g. `packagename:path/to/template.pt`).

You may supply a relative asset spec as *renderer_name*. If the *package* argument is supplied, a relative renderer path will be converted to an absolute asset specification by combining the package supplied as *package* with the relative asset specification supplied as *renderer_name*. If you do not supply a package (or *package* is *None*) the package name of the *caller* of this function will be used as the package.

The *value* provided will be supplied as the input to the renderer. Usually, for template renderings, this should be a dictionary. For other renderers, this will need to be whatever sort of value the renderer expects.

The 'system' values supplied to the renderer will include a basic set of top-level system names, such as `request`, `context`, and `renderer_name`. If *renderer globals* have been specified, these will also be used to augment the value.

Supply a `request` parameter in order to provide the renderer with the most correct 'system' values (`request` and `context` in particular).

render_to_response (*renderer_name, value, request=None, package=None*)

Using the renderer specified as `renderer_name` (a template or a static renderer) render the value (or set of values) using the result of the renderer's `__call__` method (usually a string or Unicode) as the response body.

If the renderer name refers to a file on disk (such as when the renderer is a template), it's usually best to supply the name as a *asset specification*.

You may supply a relative asset spec as `renderer_name`. If the `package` argument is supplied, a relative renderer name will be converted to an absolute asset specification by combining the package supplied as `package` with the relative asset specification supplied as `renderer_name`. If you do not supply a `package` (or `package` is `None`) the package name of the *caller* of this function will be used as the package.

The `value` provided will be supplied as the input to the renderer. Usually, for template renderings, this should be a dictionary. For other renderers, this will need to be whatever sort of value the renderer expects.

The 'system' values supplied to the renderer will include a basic set of top-level system names, such as `request`, `context`, and `renderer_name`. If *renderer globals* have been specified, these will also be used to agument the value.

Supply a `request` parameter in order to provide the renderer with the most correct 'system' values (`request` and `context` in particular).

PYRAMID . REQUEST

class Request (*environ=None, environ_getter=None, charset=(No Default), unicode_errors=(No Default), decode_param_names=(No Default), **kw*)

A subclass of the *WebOb* Request class. An instance of this class is created by the *router* and is provided to a view callable (and to other subsystems) as the *request* argument.

The documentation below (save for the `add_response_callback` and `add_finished_callback` methods, which are defined in this subclass itself, and the attributes `context`, `registry`, `root`, `subpath`, `traversed`, `view_name`, `virtual_root`, and `virtual_root_path`, each of which is added to the request by the *router* at request ingress time) are autogenerated from the *WebOb* source code used when this documentation was generated.

Due to technical constraints, we can't yet display the *WebOb* version number from which this documentation is autogenerated, but it will be the 'prevailing *WebOb* version' at the time of the release of this Pyramid version. See <http://pythonpaste.org/webob/> for further information.

context

The *context* will be available as the `context` attribute of the *request* object. It will be the context object implied by the current request. See *Traversal* for information about context objects.

registry

The *application registry* will be available as the `registry` attribute of the *request* object. See *Using the Zope Component Architecture in Pyramid* for more information about the application registry.

root

The *root* object will be available as the `root` attribute of the *request* object. It will be the resource object at which traversal started (the root). See *Traversal* for information about root objects.

subpath

The traversal *subpath* will be available as the `subpath` attribute of the *request* object. It will be a sequence containing zero or more elements (which will be Unicode objects). See *Traversal* for information about the subpath.

traversed

The “traversal path” will be available as the `traversed` attribute of the *request* object. It will be a sequence representing the ordered set of names that were used to traverse to the *context*, not including the view name or subpath. If there is a virtual root associated with the request, the virtual root path is included within the traversal path. See *Traversal* for more information.

view_name

The *view name* will be available as the `view_name` attribute of the *request* object. It will be a single string (possibly the empty string if we’re rendering a default view). See *Traversal* for information about view names.

virtual_root

The *virtual root* will be available as the `virtual_root` attribute of the *request* object. It will be the virtual root object implied by the current request. See *Virtual Hosting* for more information about virtual roots.

virtual_root_path

The *virtual root path* will be available as the `virtual_root_path` attribute of the *request* object. It will be a sequence representing the ordered set of names that were used to traverse to the virtual root object. See *Virtual Hosting* for more information about virtual roots.

exception

If an exception was raised by a *root factory* or a *view callable*, or at various other points where Pyramid executes user-defined code during the processing of a request, the exception object which was caught will be available as the `exception` attribute of the request within a *exception view*, a *response callback* or a *finished callback*. If no exception occurred, the value of `request.exception` will be `None` within response and finished callbacks.

session

If a *session factory* has been configured, this attribute will represent the current user’s *session* object. If a session factory *has not* been configured, requesting the `request.session` attribute will cause a `pyramid.exceptions.ConfigurationError` to be raised.

tmpl_context

The template context for Pylons-style applications.

matchdict

If a *route* has matched during this request, this attribute will be a dictionary containing the values matched by the URL pattern associated with the route. If a route has not matched during this request, the value of this attribute will be `None`. See *The Matchdict*.

matched_route

If a *route* has matched during this request, this attribute will be an object representing the route matched by the URL pattern associated with the route. If a route has not matched during this request, the value of this attribute will be `None`. See *The Matched Route*.

add_response_callback (*callback*)

Add a callback to the set of callbacks to be called by the *router* at a point after a *response* object is successfully created. Pyramid does not have a global response object: this functionality allows an application to register an action to be performed against the response once one is created.

A ‘callback’ is a callable which accepts two positional parameters: `request` and `response`. For example:

```
1 def cache_callback(request, response):
2     'Set the cache_control max_age for the response'
3     response.cache_control.max_age = 360
4     request.add_response_callback(cache_callback)
```

Response callbacks are called in the order they’re added (first-to-most-recently-added). No response callback is called if an exception happens in application code, or if the response object returned by *view* code is invalid.

All response callbacks are called *after* the `pyramid.events.NewResponse` event is sent.

Errors raised by callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

See also: *Using Response Callbacks*.

add_finished_callback (*callback*)

Add a callback to the set of callbacks to be called unconditionally by the *router* at the very end of request processing.

`callback` is a callable which accepts a single positional parameter: `request`. For example:

```
1 import transaction
2
3 def commit_callback(request):
4     '''commit or abort the transaction associated with request'''
5     if request.exception is not None:
6         transaction.abort()
7     else:
8         transaction.commit()
9     request.add_finished_callback(commit_callback)
```

Finished callbacks are called in the order they're added (first- to most-recently- added). Finished callbacks (unlike response callbacks) are *always* called, even if an exception happens in application code that prevents a response from being generated.

The set of finished callbacks associated with a request are called *very late* in the processing of that request; they are essentially the last thing called by the *router*. They are called after response processing has already occurred in a top-level `finally:` block within the router request processing code. As a result, mutations performed to the `request` provided to a finished callback will have no meaningful effect, because response processing will have already occurred, and the request's scope will expire almost immediately after all finished callbacks have been processed.

Errors raised by finished callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

See also: *Using Finished Callbacks*.

route_url (*route_name*, **elements*, ***kw*)

Return the URL for the route named `route_name`, using `*elements` and `**kw` as modifiers.

This is a convenience method. The result of calling `pyramid.request.Request.route_url()` is the same as calling `pyramid.url.route_url()` with an explicit `request` parameter.

The `pyramid.request.Request.route_url()` method calls the `pyramid.url.route_url()` function using the `Request` object as the `request` argument. The `route_name`, `*elements` and `*kw` arguments passed to `pyramid.request.Request.route_url()` are passed through to `pyramid.url.route_url()` unchanged and its result is returned.

This call to `pyramid.request.Request.route_url()`:

```
request.route_url('route_name')
```

Is completely equivalent to calling `pyramid.url.route_url()` like this:

```
from pyramid.url import route_url
route_url('route_name', request)
```

route_path(*route_name*, **elements*, ***kw*)

Generates a path (aka a 'relative URL', a URL minus the host, scheme, and port) for a named Pyramid *route configuration*.

This is a convenience method. The result of calling `pyramid.request.Request.route_path()` is the same as calling `pyramid.url.route_path()` with an explicit `request` parameter.

This method accepts the same arguments as `pyramid.request.Request.route_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the path, query parameters, and anchor data are present in the returned string.

The `pyramid.request.Request.route_path()` method calls the `pyramid.url.route_path()` function using the `Request` object as the `request` argument. The `*elements` and `*kw` arguments passed to `pyramid.request.Request.route_path()` are passed through to `pyramid.url.route_path()` unchanged and its result is returned.

This call to `pyramid.request.Request.route_path()`:

```
request.route_path('foobar')
```

Is completely equivalent to calling `pyramid.url.route_path()` like this:

```
from pyramid.url import route_path
route_path('foobar', request)
```

See `pyramid.url.route_path()` for more information

resource_url (*resource*, **elements*, ***kw*)

Return the URL for the *resource* object named *resource*, using **elements* and ***kw* as modifiers.

This is a convenience method. The result of calling `pyramid.request.Request.resource_url()` is the same as calling `pyramid.url.resource_url()` with an explicit *request* parameter.

The `pyramid.request.Request.resource_url()` method calls the `pyramid.url.resource_url()` function using the *Request* object as the *request* argument. The *resource*, **elements* and ***kw* arguments passed to `pyramid.request.Request.resource_url()` are passed through to `pyramid.url.resource_url()` unchanged and its result is returned.

This call to `pyramid.request.Request.resource_url()`:

```
request.resource_url(myresource)
```

Is completely equivalent to calling `pyramid.url.resource_url()` like this:

```
from pyramid.url import resource_url
resource_url(resource, request)
```



For backwards compatibility purposes, this method can also be called as `pyramid.request.Request.model_url()`.

static_url (*path*, ***kw*)

Generates a fully qualified URL for a static *asset*. The asset must live within a location defined via the `pyramid.config.Configurator.add_static_view()` *configuration declaration* directive (see *Serving Static Assets*).

This is a convenience method. The result of calling `pyramid.request.Request.static_url()` is the same as calling `pyramid.url.static_url()` with an explicit *request* parameter.

The `pyramid.request.Request.static_url()` method calls the `pyramid.url.static_url()` function using the *Request* object as the *request* argument. The ***kw* arguments passed to `pyramid.request.Request.static_url()` are passed through to `pyramid.url.static_url()` unchanged and its result is returned.

This call to `pyramid.request.Request.static_url()`:

```
request.static_url('mypackage:static/foo.css')
```

Is completely equivalent to calling `pyramid.url.static_url()` like this:

```
from pyramid.url import static_url
static_url('mypackage:static/foo.css', request)
```

See `pyramid.url.static_url()` for more information

GET

Like `.str_GET`, but may decode values and keys

POST

Like `.str_POST`, but may decode values and keys

ResponseClass

alias of `Response`

accept

Gets and sets the 'HTTP_ACCEPT' key in the environment. For more information on Accept see section 14.1. Converts it as a MIME Accept.

accept_charset

Gets and sets the 'HTTP_ACCEPT_CHARSET' key in the environment. For more information on Accept-Charset see section 14.2. Converts it as a accept header.

accept_encoding

Gets and sets the 'HTTP_ACCEPT_ENCODING' key in the environment. For more information on Accept-Encoding see section 14.3. Converts it as a accept header.

accept_language

Gets and sets the 'HTTP_ACCEPT_LANGUAGE' key in the environment. For more information on Accept-Language see section 14.4. Converts it as a accept header.

add_finished_callback (*callback*)

Add a callback to the set of callbacks to be called unconditionally by the *router* at the very end of request processing.

callback is a callable which accepts a single positional parameter: `request`. For example:

```
1 import transaction
2
3 def commit_callback(request):
4     '''commit or abort the transaction associated with request'''
5     if request.exception is not None:
6         transaction.abort()
7     else:
8         transaction.commit()
9 request.add_finished_callback(commit_callback)
```

Finished callbacks are called in the order they're added (first- to most-recently- added). Finished callbacks (unlike response callbacks) are *always* called, even if an exception happens in application code that prevents a response from being generated.

The set of finished callbacks associated with a request are called *very late* in the processing of that request; they are essentially the last thing called by the *router*. They are called after response processing has already occurred in a top-level `finally:` block within the router request processing code. As a result, mutations performed to the `request` provided to a finished callback will have no meaningful effect, because response processing will have already occurred, and the request's scope will expire almost immediately after all finished callbacks have been processed.

Errors raised by finished callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

See also: *Using Finished Callbacks*.

add_response_callback (*callback*)

Add a callback to the set of callbacks to be called by the *router* at a point after a *response* object is successfully created. Pyramid does not have a global response object: this functionality allows an application to register an action to be performed against the response once one is created.

A 'callback' is a callable which accepts two positional parameters: `request` and `response`. For example:

```
1 def cache_callback(request, response):
2     'Set the cache_control max_age for the response'
3     response.cache_control.max_age = 360
4 request.add_response_callback(cache_callback)
```

Response callbacks are called in the order they're added (first-to-most-recently-added). No response callback is called if an exception happens in application code, or if the response object returned by *view* code is invalid.

All response callbacks are called *after* the `pyramid.events.NewResponse` event is sent.

Errors raised by callbacks are not handled specially. They will be propagated to the caller of the Pyramid router application.

See also: *Using Response Callbacks*.

application_url

The URL including `SCRIPT_NAME` (no `PATH_INFO` or query string)

authorization

Gets and sets the 'HTTP_AUTHORIZATION' key in the environment. For more information on Authorization see section 14.8. Converts it as a `<function parse_auth at 0x2f44aa0>` and `<function serialize_auth at 0x2f44b18>`.

classmethod blank (*path*, *environ=None*, *base_url=None*, *headers=None*, *POST=None*, ***kw*)

Create a blank request environ (and Request wrapper) with the given path (path should be urlencoded), and any keys from environ.

The path will become `path_info`, with any query string split off and used.

All necessary keys will be added to the environ, but the values you pass in will take precedence. If you pass in `base_url` then `wsgi.url_scheme`, `HTTP_HOST`, and `SCRIPT_NAME` will be filled in from that value.

Any extra keyword will be passed to `__init__` (e.g., `decode_param_names`).

body

Return the content of the request body.

body_file

Access the body of the request (`wsgi.input`) as a file-like object.

If you set this value, `CONTENT_LENGTH` will also be updated (either set to -1, 0 if you delete the attribute, or if you set the attribute to a string then the length of the string).

cache_control

Get/set/modify the Cache-Control header (section 14.9)

call_application (*application, catch_exc_info=False*)

Call the given WSGI application, returning (*status_string, headerlist, app_iter*)

Be sure to call `app_iter.close()` if it's there.

If `catch_exc_info` is true, then returns (*status_string, headerlist, app_iter, exc_info*), where the fourth item may be None, but won't be if there was an exception. If you don't do this and there was an exception, the exception will be raised directly.

charset

Get the charset of the request.

If the request was sent with a charset parameter on the Content-Type, that will be used. Otherwise if there is a default charset (set during construction, or as a class attribute) that will be returned. Otherwise None.

Setting this property after request instantiation will always update Content-Type. Deleting the property updates the Content-Type to remove any charset parameter (if none exists, then deleting the property will do nothing, and there will be no error).

content_length

Gets and sets the 'CONTENT_LENGTH' key in the environment. For more information on CONTENT_LENGTH see section 14.13. Converts it as a int.

content_type

Return the content type, but leaving off any parameters (like charset, but also things like the type in `application/atom+xml; type=entry`)

If you set this property, you can include parameters, or if you don't include any parameters in the value then existing parameters will be preserved.

cookies

Like `.str_cookies`, but may decode values and keys

copy()

Copy the request and environment object.

This only does a shallow copy, except of `wsgi.input`

copy_body()

Copies the body, in cases where it might be shared with another request object and that is not desired.

This copies the body in-place, either into a StringIO object or a temporary file.

copy_get ()

Copies the request and environment object, but turning this request into a GET along the way. If this was a POST request (or any other verb) then it becomes GET, and the request body is thrown away.

date

Gets and sets the 'HTTP_DATE' key in the environment. For more information on Date see section 14.8. Converts it as a HTTP date.

classmethod from_file (*fp*)

Reads a request from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the request, not the end of the file.

This reads the request as represented by `str(req)`; it may not read every valid HTTP request properly.

get_response (*application, catch_exc_info=False*)

Like `.call_application(application)`, except returns a response object with `.status`, `.headers`, and `.body` attributes.

This will use `self.ResponseClass` to figure out the class of the response object to return.

headers

All the request headers as a case-insensitive dictionary-like object.

host

Host name provided in HTTP_HOST, with fall-back to SERVER_NAME

host_url

The URL through the host (no path)

if_match

Gets and sets the 'HTTP_IF_MATCH' key in the environment. For more information on If-Match see section 14.24. Converts it as a Etag.

if_modified_since

Gets and sets the 'HTTP_IF_MODIFIED_SINCE' key in the environment. For more information on If-Modified-Since see section 14.25. Converts it as a HTTP date.

if_none_match

Gets and sets the 'HTTP_IF_NONE_MATCH' key in the environment. For more information on If-None-Match see section 14.26. Converts it as a Etag.

if_range

Gets and sets the 'HTTP_IF_RANGE' key in the environment. For more information on If-Range see section 14.27. Converts it as a IfRange object.

if_unmodified_since

Gets and sets the 'HTTP_IF_UNMODIFIED_SINCE' key in the environment. For more information on If-Unmodified-Since see section 14.28. Converts it as a HTTP date.

is_xhr

Returns a boolean if X-Requested-With is present and XMLHttpRequest

Note: this isn't set by every XMLHttpRequest request, it is only set if you are using a Javascript library that sets it (or you set the header yourself manually). Currently Prototype and jQuery are known to set this header.

make_body_seekable()

This forces `environ['wsgi.input']` to be seekable. That is, if it doesn't have a *seek* method already, the content is copied into a StringIO or temporary file.

The choice to copy to StringIO is made from `self.request_body_tempfile_limit`

max_forwards

Gets and sets the 'HTTP_MAX_FORWARDS' key in the environment. For more information on Max-Forwards see section 14.31. Converts it as a int.

method

Gets and sets the 'REQUEST_METHOD' key in the environment.

model_url(resource, *elements, **kw)

Return the URL for the *resource* object named *resource*, using **elements* and ***kw* as modifiers.

This is a convenience method. The result of calling `pyramid.request.Request.resource_url()` is the same as calling `pyramid.url.resource_url()` with an explicit *request* parameter.


The `pyramid.request.Request.resource_url()` method calls the `pyramid.url.resource_url()` function using the Request object as the request argument. The *resource*, **elements* and **kw* arguments passed to `pyramid.request.Request.resource_url()` are passed through to `pyramid.url.resource_url()` unchanged and its result is returned.

This call to `pyramid.request.Request.resource_url()`:

```
request.resource_url(myresource)
```

Is completely equivalent to calling `pyramid.url.resource_url()` like this:

```
from pyramid.url import resource_url
resource_url(resource, request)
```

 For backwards compatibility purposes, this method can also be called as `pyramid.request.Request.model_url()`.

params

Like `.str_params`, but may decode values and keys

path

The path of the request, without host or query string

path_info

Gets and sets the 'PATH_INFO' key in the environment.

path_info_peek()

Returns the next segment on PATH_INFO, or None if there is no next segment. Doesn't modify the environment.

path_info_pop (*pattern=None*)

'Pops' off the next segment of PATH_INFO, pushing it onto SCRIPT_NAME, and returning the popped segment. Returns None if there is nothing left on PATH_INFO.

Does not return "" when there's an empty segment (like `/path//path`); these segments are just ignored.

Optional `pattern` argument is a regexp to match the return value before returning. If there is no match, no changes are made to the request and None is returned.

path_qs

The path of the request, without host but with query string

path_url

The URL including SCRIPT_NAME and PATH_INFO, but not QUERY_STRING

postvars

Wraps a descriptor, with a deprecation warning or error

pragma

Gets and sets the 'HTTP_PRAGMA' key in the environment. For more information on Pragma see section 14.32.

query_string

Gets and sets the 'QUERY_STRING' key in the environment.

queryvars

Wraps a descriptor, with a deprecation warning or error

range

Gets and sets the 'HTTP_RANGE' key in the environment. For more information on Range see section 14.35. Converts it as a Range object.

referer

Gets and sets the 'HTTP_REFERER' key in the environment. For more information on Referer see section 14.36.

referrer

Gets and sets the 'HTTP_REFERER' key in the environment. For more information on Referrer see section 14.36.

relative_url (*other_url, to_application=False*)

Resolve other_url relative to the request URL.

If *to_application* is True, then resolve it relative to the URL with only SCRIPT_NAME

remote_addr

Gets and sets the 'REMOTE_ADDR' key in the environment.

remote_user

Gets and sets the 'REMOTE_USER' key in the environment.

remove_conditional_headers (*remove_encoding=True, remove_range=True, remove_match=True, remove_modified=True*)

Remove headers that make the request conditional.

These headers can cause the response to be 304 Not Modified, which in some cases you may not want to be possible.

This does not remove headers like If-Match, which are used for conflict detection.

resource_url (*resource*, **elements*, ***kw*)

Return the URL for the *resource* object named *resource*, using **elements* and ***kw* as modifiers.

This is a convenience method. The result of calling `pyramid.request.Request.resource_url()` is the same as calling `pyramid.url.resource_url()` with an explicit *request* parameter.

The `pyramid.request.Request.resource_url()` method calls the `pyramid.url.resource_url()` function using the *Request* object as the *request* argument. The *resource*, **elements* and **kw* arguments passed to `pyramid.request.Request.resource_url()` are passed through to `pyramid.url.resource_url()` unchanged and its result is returned.

This call to `pyramid.request.Request.resource_url()`:

```
request.resource_url(myresource)
```

Is completely equivalent to calling `pyramid.url.resource_url()` like this:

```
from pyramid.url import resource_url
resource_url(resource, request)
```



For backwards compatibility purposes, this method can also be called as `pyramid.request.Request.model_url()`.

route_path (*route_name*, **elements*, ***kw*)

Generates a path (aka a 'relative URL', a URL minus the host, scheme, and port) for a named Pyramid *route configuration*.

This is a convenience method. The result of calling `pyramid.request.Request.route_path()` is the same as calling `pyramid.url.route_path()` with an explicit *request* parameter.

This method accepts the same arguments as `pyramid.request.Request.route_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the path, query parameters, and anchor data are present in the returned string.

The `pyramid.request.Request.route_path()` method calls the `pyramid.url.route_path()` function using the *Request* object as the *request* argument. The **elements* and **kw* arguments passed to `pyramid.request.Request.route_path()` are passed through to `pyramid.url.route_path()` unchanged and its result is returned.

This call to `pyramid.request.Request.route_path()`:

```
request.route_path('foobar')
```

Is completely equivalent to calling `pyramid.url.route_path()` like this:

```
from pyramid.url import route_path
route_path('foobar', request)
```

See `pyramid.url.route_path()` for more information

route_url (*route_name*, **elements*, ***kw*)

Return the URL for the route named `route_name`, using `*elements` and `**kw` as modifiers.

This is a convenience method. The result of calling `pyramid.request.Request.route_url()` is the same as calling `pyramid.url.route_url()` with an explicit `request` parameter.

The `pyramid.request.Request.route_url()` method calls the `pyramid.url.route_url()` function using the `Request` object as the `request` argument. The `route_name`, `*elements` and `*kw` arguments passed to `pyramid.request.Request.route_url()` are passed through to `pyramid.url.route_url()` unchanged and its result is returned.

This call to `pyramid.request.Request.route_url()`:

```
request.route_url('route_name')
```

Is completely equivalent to calling `pyramid.url.route_url()` like this:

```
from pyramid.url import route_url
route_url('route_name', request)
```

scheme

Gets and sets the 'wsgi.url_scheme' key in the environment.

script_name

Gets and sets the 'SCRIPT_NAME' key in the environment.

server_name

Gets and sets the 'SERVER_NAME' key in the environment.

server_port

Gets and sets the 'SERVER_PORT' key in the environment. Converts it as a int.

session

Obtain the *session* object associated with this request. If a *session factory* has not been registered during application configuration, a `pyramid.exceptions.ConfigurationError` will be raised

static_url (*path*, ***kw*)

Generates a fully qualified URL for a static *asset*. The asset must live within a location defined via the `pyramid.config.Configurator.add_static_view()` *configuration declaration* directive (see *Serving Static Assets*).

This is a convenience method. The result of calling `pyramid.request.Request.static_url()` is the same as calling `pyramid.url.static_url()` with an explicit *request* parameter.

The `pyramid.request.Request.static_url()` method calls the `pyramid.url.static_url()` function using the *Request* object as the *request* argument. The **kw* arguments passed to `pyramid.request.Request.static_url()` are passed through to `pyramid.url.static_url()` unchanged and its result is returned.

This call to `pyramid.request.Request.static_url()`:

```
request.static_url('mypackage:static/foo.css')
```

Is completely equivalent to calling `pyramid.url.static_url()` like this:

```
from pyramid.url import static_url
static_url('mypackage:static/foo.css', request)
```

See `pyramid.url.static_url()` for more information

str_GET

Return a *MultiDict* containing all the variables from the *QUERY_STRING*.

str_POST

Return a *MultiDict* containing all the variables from a form request. Returns an empty dict-like object for non-form requests.

Form requests are typically POST requests, however PUT requests with an appropriate Content-Type are also supported.

str_cookies

Return a *plain* dictionary of cookies as found in the request.

str_params

A dictionary-like object containing both the parameters from the query string and request body.

str_postvars

Wraps a descriptor, with a deprecation warning or error

str_queryvars

Wraps a descriptor, with a deprecation warning or error

tmpl_context

Template context (for Pylons apps)

upath_info

upath_property('PATH_INFO')

url

The full request URL, including QUERY_STRING

urlargs

Return any *positional* variables matched in the URL.

Takes values from `environ['wsgiorg.routing_args']`. Systems like routes set this value.

urlvars

Return any *named* variables matched in the URL.

Takes values from `environ['wsgiorg.routing_args']`. Systems like routes set this value.

uscript_name

upath_property('SCRIPT_NAME')

user_agent

Gets and sets the 'HTTP_USER_AGENT' key in the environment. For more information on User-Agent see section 14.43.

PYRAMID . RESPONSE

class Response (*body=None, status=None, headerlist=None, app_iter=None, request=None, content_type=None, conditional_response=None, **kw*)
Represents a WSGI response

RequestClass

alias of Request

accept_ranges

Gets and sets and deletes the Accept-Ranges header. For more information on Accept-Ranges see section 14.5.

age

Gets and sets and deletes the Age header. For more information on Age see section 14.6. Converts it as a int.

allow

Gets and sets and deletes the Allow header. For more information on Allow see section 14.7. Converts it as a list.

app_iter

Returns the app_iter of the response.

If body was set, this will create an app_iter from that body (a single-item list)

app_iter_range (*start, stop*)

Return a new app_iter built from the response app_iter, that serves up only the given start:stop range.

body

The body of the response, as a `str`. This will read in the entire `app_iter` if necessary.

body_file

A file-like object that can be used to write to the body. If you passed in a list `app_iter`, that `app_iter` will be modified by writes.

cache_control

Get/set/modify the Cache-Control header (section 14.9)

charset

Get/set the charset (in the Content-Type)

conditional_response_app (*environ, start_response*)

Like the normal `__call__` interface, but checks conditional headers:

- If-Modified-Since (304 Not Modified; only on GET, HEAD)
- If-None-Match (304 Not Modified; only on GET, HEAD)
- Range (406 Partial Content; only on GET, HEAD)

content_disposition

Gets and sets and deletes the Content-Disposition header. For more information on Content-Disposition see section 19.5.1.

content_encoding

Gets and sets and deletes the Content-Encoding header. For more information on Content-Encoding see section 14.11.

content_language

Gets and sets and deletes the Content-Language header. For more information on Content-Language see section 14.12. Converts it as a list.

content_length

Gets and sets and deletes the Content-Length header. For more information on Content-Length see section 14.17. Converts it as a int.

content_location

Gets and sets and deletes the Content-Location header. For more information on Content-Location see section 14.14.

content_md5

Gets and sets and deletes the Content-MD5 header. For more information on Content-MD5 see section 14.14.

content_range

Gets and sets and deletes the Content-Range header. For more information on Content-Range see section 14.16. Converts it as a ContentRange object.

content_type

Get/set the Content-Type header (or None), *without* the charset or any parameters.

If you include parameters (or ; at all) when setting the content_type, any existing parameters will be deleted; otherwise they will be preserved.

content_type_params

A dictionary of all the parameters in the content type.

(This is not a view, set to change, modifications of the dict would not be applied otherwise)

copy ()

Makes a copy of the response

date

Gets and sets and deletes the Date header. For more information on Date see section 14.18. Converts it as a HTTP date.

delete_cookie (*key, path='/', domain=None*)

Delete a cookie from the client. Note that path and domain must match how the cookie was originally set.

This sets the cookie to the empty string, and max_age=0 so that it should expire immediately.

encode_content (*encoding='gzip', lazy=False*)

Encode the content with the given encoding (only gzip and identity are supported).

environ

Get/set the request environ associated with this response, if any.

etag

Gets and sets and deletes the ETag header. For more information on ETag see section 14.19. Converts it as a Entity tag.

expires

Gets and sets and deletes the Expires header. For more information on Expires see section 14.21. Converts it as a HTTP date.

classmethod from_file (*fp*)

Reads a response from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the response, not the end of the file.

This reads the response as represented by `str(resp)`; it may not read every valid HTTP response properly. Responses must have a `Content-Length`

headerlist

The list of response headers

headers

The headers in a dictionary-like object

last_modified

Gets and sets and deletes the Last-Modified header. For more information on Last-Modified see section 14.29. Converts it as a HTTP date.

location

Gets and sets and deletes the Location header. For more information on Location see section 14.30.

md5_etag (*body=None, set_content_md5=False*)

Generate an etag for the response object using an MD5 hash of the body (the `body` parameter, or `self.body` if not given)

Sets `self.etag` If `set_content_md5` is `True` sets `self.content_md5` as well

merge_cookies (*resp*)

Merge the cookies that were set on this response with the given `resp` object (which can be any WSGI application).

If the `resp` is a `webob.Response` object, then the other object will be modified in-place.

pragma

Gets and sets and deletes the Pragma header. For more information on Pragma see section 14.32.

request

Return the request associated with this response if any.

retry_after

Gets and sets and deletes the Retry-After header. For more information on Retry-After see section 14.37. Converts it as a HTTP date or delta seconds.

server

Gets and sets and deletes the Server header. For more information on Server see section 14.38.

set_cookie (*key*, *value*='', *max_age*=None, *path*='/', *domain*=None, *secure*=False, *httponly*=False, *comment*=None, *expires*=None, *overwrite*=False)

Set (add) a cookie for the response

status

The status string

status_code

Wraps a descriptor, with a deprecation warning or error

status_int

The status as an integer

ubody

Alias for unicode_body

unicode_body

Get/set the unicode value of the body (using the charset of the Content-Type)

unset_cookie (*key*, *strict*=True)

Unset a cookie with the given name (remove it from the response).

vary

Gets and sets and deletes the Vary header. For more information on Vary see section 14.44.
Converts it as a list.

www_authenticate

Gets and sets and deletes the WWW-Authenticate header. For more information on WWW-Authenticate see section 14.47. Converts it as a <function parse_auth at 0x2f44aa0> and <function serialize_auth at 0x2f44b18>.

PYRAMID . SCRIPTING

get_root (*app*, *request=None*)

Return a tuple composed of (*root*, *closer*) when provided a *router* instance as the *app* argument. The *root* returned is the application root object. The *closer* returned is a callable (accepting no arguments) that should be called when your scripting application is finished using the root. If *request* is not *None*, it is used as the request passed to the Pyramid application root factory. A request is constructed and passed to the root factory if *request* is *None*.

PYRAMID . SECURITY

52.1 Authentication API Functions

authenticated_userid (*request*)

Return the userid of the currently authenticated user or `None` if there is no *authentication policy* in effect or there is no currently authenticated user.

unauthenticated_userid (*request*)

Return an object which represents the *claimed* (not verified) user id of the credentials present in the request. `None` if there is no *authentication policy* in effect or there is no user data associated with the current request. This differs from `authenticated_userid()`, because the effective authentication policy will not ensure that a record associated with the userid exists in persistent storage.

effective_principals (*request*)

Return the list of ‘effective’ *principal* identifiers for the *request*. This will include the userid of the currently authenticated user if a user is currently authenticated. If no *authentication policy* is in effect, this will return an empty sequence.

forget (*request*)

Return a sequence of header tuples (e.g. `[('Set-Cookie', 'foo=abc')]`) suitable for ‘forgetting’ the set of credentials possessed by the currently authenticated user. A common usage might look like so within the body of a view function (*response* is assumed to be an *WebOb*-style *response* object computed previously by the view code):

```
from pyramid.security import forget
headers = forget(request)
response.headerlist.extend(headers)
return response
```

If no *authentication policy* is in use, this function will always return an empty sequence.

remember (*request*, *principal*, ***kw*)

Return a sequence of header tuples (e.g. [('Set-Cookie', 'foo=abc')]) suitable for 'remembering' a set of credentials implied by the data passed as *principal* and **kw* using the current *authentication policy*. Common usage might look like so within the body of a view function (*response* is assumed to be a *WebOb* -style *response* object computed previously by the view code):

```
from pyramid.security import remember
headers = remember(request, 'chrism', password='123', max_age='86400')
response.headerlist.extend(headers)
return response
```

If no *authentication policy* is in use, this function will always return an empty sequence. If used, the composition and meaning of ***kw* must be agreed upon by the calling code and the effective *authentication policy*.


52.2 Authorization API Functions

has_permission (*permission*, *context*, *request*)

Provided a *permission* (a string or unicode object), a *context* (a *resource* instance) and a *request* object, return an instance of `pyramid.security.Allowed` if the *permission* is granted in this *context* to the user implied by the *request*. Return an instance of `pyramid.security.Denied` if this *permission* is not granted in this *context* to this user. This function delegates to the current *authentication* and *authorization policies*. Return `pyramid.security.Allowed` unconditionally if no *authentication policy* has been configured in this application.

principals_allowed_by_permission (*context*, *permission*)

Provided a *context* (a *resource* object), and a *permission* (a string or unicode object), if a *authorization policy* is in effect, return a sequence of *principal* ids that possess the *permission* in the *context*. If no *authorization policy* is in effect, this will return a sequence with the single value `pyramid.security.Everyone` (the special *principal* identifier representing all *principals*).

 even if an *authorization policy* is in effect, some (exotic) authorization policies may not implement the required machinery for this function; those will cause a `NotImplementedError` exception to be raised when this function is invoked.

view_execution_permitted(*context, request, name=''*)

If the view specified by `context` and `name` is protected by a *permission*, check the permission associated with the view using the effective authentication/authorization policies and the `request`. Return a boolean result. If no *authorization policy* is in effect, or if the view is not protected by a permission, return `True`.

52.3 Constants

Everyone

The special principal id named 'Everyone'. This principal id is granted to all requests. Its actual value is the string 'system.Everyone'.

Authenticated

The special principal id named 'Authenticated'. This principal id is granted to all requests which contain any other non-Everyone principal id (according to the *authentication policy*). Its actual value is the string 'system.Authenticated'.

ALL_PERMISSIONS

An object that can be used as the `permission` member of an ACE which matches all permissions unconditionally. For example, an ACE that uses `ALL_PERMISSIONS` might be composed like so: ('Deny', 'system.Everyone', `ALL_PERMISSIONS`).

DENY_ALL

A convenience shorthand ACE that defines ('Deny', 'system.Everyone', `ALL_PERMISSIONS`). This is often used as the last ACE in an ACL in systems that use an "inheriting" security policy, representing the concept "don't inherit any other ACEs".

52.4 Return Values

Allow

The ACE "action" (the first element in an ACE e.g. (Allow, Everyone, 'read') that means allow access. A sequence of ACEs makes up an ACL. It is a string, and its actual value is "Allow".

Deny

The ACE “action” (the first element in an ACE e.g. (`Deny`, `'george'`, `'read'`) that means deny access. A sequence of ACEs makes up an ACL. It is a string, and its actual value is “Deny”.

class ACLDenied

An instance of `ACLDenied` represents that a security check made explicitly against ACL was denied. It evaluates equal to all boolean false types. It also has attributes which indicate which `acl`, `ace`, `permission`, `principals`, and `context` were involved in the request. Its `__str__` method prints a summary of these attributes for debugging purposes. The same summary is available as the `msg` attribute.

class ACLAllowed

An instance of `ACLAllowed` represents that a security check made explicitly against ACL was allowed. It evaluates equal to all boolean true types. It also has attributes which indicate which `acl`, `ace`, `permission`, `principals`, and `context` were involved in the request. Its `__str__` method prints a summary of these attributes for debugging purposes. The same summary is available as the `msg` attribute.

class Denied

An instance of `Denied` is returned when a security-related API or other Pyramid code denies an action unrelated to an ACL check. It evaluates equal to all boolean false types. It has an attribute named `msg` describing the circumstances for the deny.

class Allowed

An instance of `Allowed` is returned when a security-related API or other Pyramid code allows an action unrelated to an ACL check. It evaluates equal to all boolean true types. It has an attribute named `msg` describing the circumstances for the allow.

PYRAMID . SETTINGS

`get_settings ()`

Return a *deployment settings* object for the current application. The object is a dictionary-like object that contains key/value pairs based on the dictionary passed as the `settings` argument to the `pyramid.config.Configurator` constructor or the `pyramid.router.make_app ()` API.



This method is deprecated as of Pyramid 1.0. Use `pyramid.threadlocals.get_current_registry().settings` instead or use the `settings` attribute of the registry available from the request (`request.registry.settings`).

`asbool (s)`

Return the boolean value `True` if the case-lowered value of string input `s` is any of `t`, `true`, `y`, `on`, or `1`, otherwise return the boolean value `False`. If `s` is the value `None`, return `False`. If `s` is already one of the boolean values `True` or `False`, return it.

PYRAMID . TESTING

setUp (*registry=None, request=None, hook_zca=True, autocommit=True, settings=None*)

Set Pyramid registry and request thread locals for the duration of a single unit test.

Use this function in the `setUp` method of a unittest test case which directly or indirectly uses:

- any method of the `pyramid.config.Configurator` object returned by this function.
- the `pyramid.threadlocal.get_current_registry()` or `pyramid.threadlocal.get_current_request()` functions.

If you use the `get_current_*` functions (or call Pyramid code that uses these functions) without calling `setUp`, `pyramid.threadlocal.get_current_registry()` will return a *global application registry*, which may cause unit tests to not be isolated with respect to registrations they perform.

If the `registry` argument is `None`, a new empty *application registry* will be created (an instance of the `pyramid.registry.Registry` class). If the `registry` argument is not `None`, the value passed in should be an instance of the `pyramid.registry.Registry` class or a suitable testing analogue.

After `setUp` is finished, the registry returned by the `pyramid.threadlocal.get_current_request()` function will be the passed (or constructed) registry until `pyramid.testing.tearDown()` is called (or `pyramid.testing.setUp()` is called again).

If the `hook_zca` argument is `True`, `setUp` will attempt to perform the operation `zope.component.getSiteManager().sethook()`

`pyramid.threadlocal.get_current_registry()`, which will cause the *Zope Component Architecture* global API (e.g. `zope.component.getSiteManager()`, `zope.component.getAdapter()`, and so on) to use the registry constructed by `setUp` as the value it returns from `zope.component.getSiteManager()`. If the `zope.component` package cannot be imported, or if `hook_zca` is `False`, the hook will not be set.

If `settings` is not `None`, it must be a dictionary representing the values passed to a Configurator as its `settings=` argument.

This function returns an instance of the `pyramid.config.Configurator` class, which can be used for further configuration to set up an environment suitable for a unit or integration test. The `registry` attribute attached to the Configurator instance represents the ‘current’ *application registry*; the same registry will be returned by `pyramid.threadlocal.get_current_registry()` during the execution of the test.

tearDown (*unhook_zca=True*)

Undo the effects `pyramid.testing.setUp()`. Use this function in the `tearDown` method of a unit test that uses `pyramid.testing.setUp()` in its `setUp` method.

If the `unhook_zca` argument is `True` (the default), call `zope.component.getSiteManager.reset()`. This undoes the action of `pyramid.testing.setUp()` called with the argument `hook_zca=True`. If `zope.component` cannot be imported, ignore the argument.

cleanUp (**arg, **kw*)

`pyramid.testing.cleanUp()` is an alias for `pyramid.testing.setUp()`.

class DummyResource (*__name__=None, __parent__=None, __provides__=None, **kw*)

A dummy Pyramid *resource* object.

clone (*__name__=<object object at 0x4d99140>, __parent__=<object object at 0x4d99140>, **kw*)

Create a clone of the resource object. If `__name__` or `__parent__` arguments are passed, use these values to override the existing `__name__` or `__parent__` of the resource. If any extra keyword args are passed in via the `kw` argument, use these keywords to add to or override existing resource keywords (attributes).

items ()

Return the items set by `__setitem__`

keys ()

Return the keys set by `__setitem__`

values ()

Return the values set by `__setitem__`

class DummyRequest (*params=None, environ=None, headers=None, path='/', cookies=None, post=None, **kw*)

A dummy request object (imitates a *request* object).

The `params`, `environ`, `headers`, `path`, and `cookies` arguments correspond to their `:term` 'WebOb' equivalents.

The `post` argument, if passed, populates the request's `POST` attribute, but *not* `params`, in order to allow testing that the app accepts data for a given view only from `POST` requests. This argument also sets `self.method` to "POST".

Extra keyword arguments are assigned as attributes of the request itself.

class DummyTemplateRenderer (*string_response=''*)

An instance of this class is returned from `pyramid.config.Configurator.testing_add_renderer`. It has a helper function (`assert_`) that makes it possible to make an assertion which compares data passed to the renderer by the view function against expected key/value pairs.

assert_ (***kw*)

Accept an arbitrary set of assertion key/value pairs. For each assertion key/value pair assert that the renderer (eg. `pyramid.renderer.render_to_response()`) received the key with a value that equals the asserted value. If the renderer did not receive the key at all, or the value received by the renderer doesn't match the assertion value, raise an `AssertionError`.

PYRAMID . THREADLOCAL

`get_current_request ()`

Return the currently active request or `None` if no request is currently active.

This function should be used *extremely sparingly*, usually only in unit testing code. it's almost always usually a mistake to use `get_current_request` outside a testing context because its usage makes it possible to write code that can be neither easily tested nor scripted.

`get_current_registry ()`

Return the currently active *application registry* or the global application registry if no request is currently active.

This function should be used *extremely sparingly*, usually only in unit testing code. it's almost always usually a mistake to use `get_current_registry` outside a testing context because its usage makes it possible to write code that can be neither easily tested nor scripted.

PYRAMID . TRAVERSAL

find_interface (*resource*, *class_or_interface*)

Return the first resource found in the *lineage* of *resource* which, a) if *class_or_interface* is a Python class object, is an instance of the class or any subclass of that class or b) if *class_or_interface* is a *interface*, provides the specified interface. Return `None` if no resource providing *interface_or_class* can be found in the lineage. The *resource* passed in *must* be *location-aware*.

find_resource (*resource*, *path*)

Given a resource object and a string or tuple representing a path (such as the return value of `pyramid.traversal.resource_path()` or `pyramid.traversal.resource_path_tuple()`), return a resource in this application's resource tree at the specified path. The resource passed in *must* be *location-aware*. If the path cannot be resolved (if the respective node in the resource tree does not exist), a `KeyError` will be raised.

This function is the logical inverse of `pyramid.traversal.resource_path()` and `pyramid.traversal.resource_path_tuple()`; it can resolve any path string or tuple generated by either of those functions.

Rules for passing a *string* as the *path* argument: if the first character in the path string is the with the `/` character, the path will considered absolute and the resource tree traversal will start at the root resource. If the first character of the path string is *not* the `/` character, the path is considered relative and resource tree traversal will begin at the resource object supplied to the function as the *resource* argument. If an empty string is passed as *path*, the *resource* passed in will be returned. Resource path strings must be escaped in the following manner: each Unicode path segment must be encoded as UTF-8 and as each path segment must escaped via Python's `urllib.quote`. For example,

`/path/to%20the/La%20Pe%C3%B1a` (absolute) or `to%20the/La%20Pe%C3%B1a` (relative). The `pyramid.traversal.resource_path()` function generates strings which follow these rules (albeit only absolute ones).

Rules for passing a *tuple* as the `path` argument: if the first element in the path tuple is the empty string (for example `('', 'a', 'b', 'c')`), the path is considered absolute and the resource tree traversal will start at the resource tree root object. If the first element in the path tuple is not the empty string (for example `('a', 'b', 'c')`), the path is considered relative and resource tree traversal will begin at the resource object supplied to the function as the `resource` argument. If an empty sequence is passed as `path`, the `resource` passed in itself will be returned. No URL-quoting or UTF-8-encoding of individual path segments within the tuple is required (each segment may be any string or unicode object representing a resource name). Resource path tuples generated by `pyramid.traversal.resource_path_tuple()` can always be resolved by `find_resource`.



For backwards compatibility purposes, this function can also be imported as `pyramid.traversal.find_model()`, although doing so will emit a deprecation warning.

find_root (*resource*)

Find the root node in the resource tree to which `resource` belongs. Note that `resource` should be *location-aware*. Note that the root resource is available in the request object by accessing the `request.root` attribute.

resource_path (*resource*, **elements*)

Return a string object representing the absolute physical path of the resource object based on its position in the resource tree, e.g. `/foo/bar`. Any positional arguments passed in as `elements` will be appended as path segments to the end of the resource path. For instance, if the resource's path is `/foo/bar` and `elements` equals `('a', 'b')`, the returned string will be `/foo/bar/a/b`. The first character in the string will always be the `/` character (a leading `/` character in a path string represents that the path is absolute).

Resource path strings returned will be escaped in the following manner: each unicode path segment will be encoded as UTF-8 and each path segment will be escaped via Python's `urllib.quote`. For example, `/path/to%20the/La%20Pe%C3%B1a`.

This function is a logical inverse of `pyramid.traversal.find_resource`: it can be used to generate path references that can later be resolved via that function.

The `resource` passed in *must* be *location-aware*.

i Each segment in the path string returned will use the `__name__` attribute of the resource it represents within the resource tree. Each of these segments *should* be a unicode or string object (as per the contract of *location-awareness*). However, no conversion or safety checking of resource names is performed. For instance, if one of the resources in your tree has a `__name__` which (by error) is a dictionary, the `pyramid.traversal.resource_path()` function will attempt to append it to a string and it will cause a `pyramid.exceptions.URLDecodeError`.

i The *root* resource *must* have a `__name__` attribute with a value of either `None` or the empty string for paths to be generated properly. If the root resource has a non-null `__name__` attribute, its name will be prepended to the generated path rather than a single leading `'/'` character.

i For backwards compatibility purposes, this function can also be imported as `model_path`, although doing so will cause a deprecation warning to be emitted.

resource_path_tuple (*resource*, **elements*)

Return a tuple representing the absolute physical path of the `resource` object based on its position in a resource tree, e.g. `("", 'foo', 'bar')`. Any positional arguments passed in as `elements` will be appended as elements in the tuple representing the resource path. For instance, if the resource's path is `("", 'foo', 'bar')` and `elements` equals `('a', 'b')`, the returned tuple will be `("", 'foo', 'bar', 'a', 'b')`. The first element of this tuple will always be the empty string (a leading empty string element in a path tuple represents that the path is absolute).

This function is a logical inverse of `pyramid.traversal.find_resource()`: it can be used to generate path references that can later be resolved that function.

The `resource` passed in *must* be *location-aware*.

i Each segment in the path tuple returned will equal the `__name__` attribute of the resource it represents within the resource tree. Each of these segments *should* be a unicode or string object (as per the contract of *location-awareness*). However, no conversion or safety checking of resource names is performed. For instance, if one of the resources in your tree has a `__name__` which (by error) is a dictionary, that dictionary will be placed in the path tuple; no warning or error will be given.

i The *root* resource *must* have a `__name__` attribute with a value of either `None` or the empty string for path tuples to be generated properly. If the root resource has a non-null `__name__` attribute, its name will be the first element in the generated path tuple rather than the empty string.

i For backwards compatibility purposes, this function can also be imported as `model_path_tuple`, although doing so will cause a deprecation warning to be emitted.

quote_path_segment (*segment*)

Return a quoted representation of a ‘path segment’ (such as the string `__name__` attribute of a resource) as a string. If the *segment* passed in is a unicode object, it is converted to a UTF-8 string, then it is URL-quoted using Python’s `urllib.quote`. If the *segment* passed in is a string, it is URL-quoted using Python’s `urllib.quote`. If the *segment* passed in is not a string or unicode object, an error will be raised. The return value of `quote_path_segment` is always a string, never Unicode.

i The return value for each segment passed to this function is cached in a module-scope dictionary for speed: the cached version is returned when possible rather than recomputing the quoted version. No cache emptying is ever done for the lifetime of an application, however. If you pass arbitrary user-supplied strings to this function (as opposed to some bounded set of values from a ‘working set’ known to your application), it may become a memory leak.

virtual_root (*resource*, *request*)

Provided any *resource* and a *request* object, return the resource object representing the *virtual root* of the current *request*. Using a virtual root in a *traversal*-based Pyramid application permits rooting, for example, the resource at the traversal path `/cms` at `http://example.com/` instead of rooting it at `http://example.com/cms/`.

If the *resource* passed in is a context obtained via *traversal*, and if the `HTTP_X_VHM_ROOT` key is in the WSGI environment, the value of this key will be treated as a ‘virtual root path’: the `pyramid.traversal.find_resource()` API will be used to find the virtual root resource using this path; if the resource is found, it will be returned. If the `HTTP_X_VHM_ROOT` key is not present in the WSGI environment, the physical *root* of the resource tree will be returned instead.

Virtual roots are not useful at all in applications that use *URL dispatch*. Contexts obtained via URL dispatch don’t really support being virtually rooted (each URL dispatch context is both its own physical and virtual root). However if this API is called with a *resource* argument which is a context obtained via URL dispatch, the resource passed in will be returned unconditionally.

traverse (*resource, path*)

Given a resource object as `resource` and a string or tuple representing a path as `path` (such as the return value of `pyramid.traversal.resource_path()` or `pyramid.traversal.resource_path_tuple()` or the value of `request.environ['PATH_INFO']`), return a dictionary with the keys `context`, `root`, `view_name`, `subpath`, `traversed`, `virtual_root`, and `virtual_root_path`.

A definition of each value in the returned dictionary:

- `context`: The *context* (a *resource* object) found via traversal or url dispatch. If the `path` passed in is the empty string, the value of the `resource` argument passed to this function is returned.
- `root`: The resource object at which *traversal* begins. If the `resource` passed in was found via url dispatch or if the `path` passed in was relative (non-absolute), the value of the `resource` argument passed to this function is returned.
- `view_name`: The *view name* found during *traversal* or *url dispatch*; if the `resource` was found via traversal, this is usually a representation of the path segment which directly follows the path to the `context` in the path. The `view_name` will be a Unicode object or the empty string. The `view_name` will be the empty string if there is no element which follows the `context` path. An example: if the path passed is `/foo/bar`, and a resource object is found at `/foo` (but not at `/foo/bar`), the ‘view name’ will be `u'bar'`. If the resource was found via *urldispatch*, the `view_name` will be the name the route found was registered with.
- `subpath`: For a resource found via *traversal*, this is a sequence of path segments found in the path that follow the `view_name` (if any). Each of these items is a Unicode object. If no path segments follow the `view_name`, the `subpath` will be the empty sequence. An example: if the path passed is `/foo/bar/baz/buz`, and a resource object is found at `/foo` (but not `/foo/bar`), the ‘view name’ will be `u'bar'` and the `subpath` will be `[u'baz', u'buz']`. For a resource found via url dispatch, the `subpath` will be a sequence of values discerned from `*subpath` in the route pattern matched or the empty sequence.
- `traversed`: The sequence of path elements traversed from the `root` to find the `context` object during *traversal*. Each of these items is a Unicode object. If no path segments were traversed to find the `context` object (e.g. if the `path` provided is the empty string), the `traversed` value will be the empty sequence. If the `resource` is a resource found via *url dispatch*, `traversed` will be `None`.
- `virtual_root`: A resource object representing the ‘virtual’ root of the resource tree being traversed during *traversal*. See *Virtual Hosting* for a definition of the virtual root object. If no virtual hosting is in effect, and the `path` passed in was absolute, the `virtual_root` will be the *physical* root resource object (the object at which *traversal* begins). If the `resource` passed in was found via *URL dispatch* or if the `path` passed in was relative, the `virtual_root` will always equal the `root` object (the resource passed in).

- `virtual_root_path` – If *traversal* was used to find the resource, this will be the sequence of path elements traversed to find the `virtual_root` resource. Each of these items is a Unicode object. If no path segments were traversed to find the `virtual_root` resource (e.g. if virtual hosting is not in effect), the `traversed` value will be the empty list. If url dispatch was used to find the resource, this will be `None`.

If the path cannot be resolved, a `KeyError` will be raised.

Rules for passing a *string* as the `path` argument: if the first character in the path string is the with the `/` character, the path will considered absolute and the resource tree traversal will start at the root resource. If the first character of the path string is *not* the `/` character, the path is considered relative and resource tree traversal will begin at the resource object supplied to the function as the `resource` argument. If an empty string is passed as `path`, the `resource` passed in will be returned. Resource path strings must be escaped in the following manner: each Unicode path segment must be encoded as UTF-8 and each path segment must escaped via Python's `urllib.quote`. For example, `/path/to%20the/La%20Pe%C3%B1a` (absolute) or `to%20the/La%20Pe%C3%B1a` (relative). The `pyramid.traversal.resource_path()` function generates strings which follow these rules (albeit only absolute ones).

Rules for passing a *tuple* as the `path` argument: if the first element in the path tuple is the empty string (for example `('', 'a', 'b', 'c')`), the path is considered absolute and the resource tree traversal will start at the resource tree root object. If the first element in the path tuple is not the empty string (for example `('a', 'b', 'c')`), the path is considered relative and resource tree traversal will begin at the resource object supplied to the function as the `resource` argument. If an empty sequence is passed as `path`, the `resource` passed in itself will be returned. No URL-quoting or UTF-8-encoding of individual path segments within the tuple is required (each segment may be any string or unicode object representing a resource name).

Explanation of the conversion of `path` segment values to Unicode during traversal: Each segment is URL-unquoted, and decoded into Unicode. Each segment is assumed to be encoded using the UTF-8 encoding (or a subset, such as ASCII); a `pyramid.exceptions.URLDecodeError` is raised if a segment cannot be decoded. If a segment name is empty or if it is `.`, it is ignored. If a segment name is `..`, the previous segment is deleted, and the `..` is ignored. As a result of this process, the return values `view_name`, each element in the `subpath`, each element in `traversed`, and each element in the `virtual_root_path` will be Unicode as opposed to a string, and will be URL-decoded.

traversal_path (*path*)

Given a `PATH_INFO` string (slash-separated path segments), return a tuple representing that path which can be used to traverse a resource tree.

The `PATH_INFO` is split on slashes, creating a list of segments. Each segment is URL-unquoted, and subsequently decoded into Unicode. Each segment is assumed to be encoded using the UTF-8

encoding (or a subset, such as ASCII); a `pyramid.exceptions.URLDecodeError` is raised if a segment cannot be decoded. If a segment name is empty or if it is `.`, it is ignored. If a segment name is `..`, the previous segment is deleted, and the `..` is ignored.

If this function is passed a Unicode object instead of a string, that Unicode object *must* directly encodeable to ASCII. For example, `u'foo'` will work but `u'<unprintable unicode>'` (a Unicode object with characters that cannot be encoded to ascii) will not.

Examples:

```
/
()
/foo/bar/baz
(u'foo', u'bar', u'baz')
foo/bar/baz
(u'foo', u'bar', u'baz')
/foo/bar/baz/
(u'foo', u'bar', u'baz')
/foo//bar//baz/
(u'foo', u'bar', u'baz')
/foo/bar/baz/..
(u'foo', u'bar')
/my%20archives/hello
(u'my archives', u'hello')
/archives/La%20Pe%C3%B1a
(u'archives', u'<unprintable unicode>')
```




This function does not generate the same type of tuples that `pyramid.traversal.resource_path_tuple()` does. In particular, the leading empty string is not present in the tuple it returns, unlike tuples returned by `pyramid.traversal.resource_path_tuple()`. As a result, tuples generated by `traversal_path` are not resolveable by the `pyramid.traversal.find_resource()` API. `traversal_path` is a function mostly used by the internals of Pyramid and by people writing their own traversal machinery, as opposed to users writing applications in Pyramid.

PYRAMID . URL

Utility functions for dealing with URLs in pyramid

resource_url (*context, request, *elements, query=None, anchor=None*)


Generate a string representing the absolute URL of the *resource* object based on the `wsgi.url_scheme`, `HTTP_HOST` or `SERVER_NAME` in the *request*, plus any `SCRIPT_NAME`. The overall result of this function is always a UTF-8 encoded string (never Unicode).

 Calling `pyramid.Request.resource_url()` can be used to achieve the same result as `pyramid.url.resource_url()`.


Examples:

```
resource_url(context, request) =>
    http://example.com/
resource_url(context, request, 'a.html') =>
    http://example.com/a.html
resource_url(context, request, 'a.html', query={'q':'1'}) =>
    http://example.com/a.html?q=1
resource_url(context, request, 'a.html', anchor='abc') =>
    http://example.com/a.html#abc
```


Any positional arguments passed in as `elements` must be strings or Unicode objects. These will be joined by slashes and appended to the generated resource URL. Each of the elements passed in is URL-quoted before being appended; if any element is Unicode, it will be converted to a UTF-8 bytestring before being URL-quoted.

 if no `elements` arguments are specified, the resource URL will end with a trailing slash. If any `elements` are used, the generated URL will *not* end in trailing slash.

If a keyword argument `query` is present, it will be used to compose a query string that will be tacked on to the end of the URL. The value of `query` must be a sequence of two-tuples or a data structure with an `.items()` method that returns a sequence of two-tuples (presumably a dictionary). This data structure will be turned into a query string per the documentation of `pyramid.url.urlencode` function. After the query data is turned into a query string, a leading `?` is prepended, and the resulting string is appended to the generated URL.


 Python data structures that are passed as `query` which are sequences or dictionaries are turned into a string under the same rules as when run through `urllib.urlencode()` with the `doseq` argument equal to `True`. This means that sequences can be passed as values, and a `k=v` pair will be placed into the query string for each value.

If a keyword argument `anchor` is present, its string representation will be used as a named anchor in the generated URL (e.g. if `anchor` is passed as `foo` and the resource URL is `http://example.com/resource/url`, the resulting generated URL will be `http://example.com/resource/url#foo`).

 If `anchor` is passed as a string, it should be UTF-8 encoded. If `anchor` is passed as a Unicode object, it will be converted to UTF-8 before being appended to the URL. The anchor value is not quoted in any way before being appended to the generated URL.

If both `anchor` and `query` are specified, the anchor element will always follow the query element, e.g. `http://example.com?foo=1#bar`.

If the `resource` passed in has a `__resource_url__` method, it will be used to generate the URL (scheme, host, port, path) that for the base resource which is operated upon by this function. See also *Overriding Resource URL Generation*.

 If the `resource` used is the result of a *traversal*, it must be *location-aware*. The resource can also be the context of a *URL dispatch*; contexts found this way do not need to be location-aware.

i If a ‘virtual root path’ is present in the request environment (the value of the WSGI environ key `HTTP_X_VHM_ROOT`), and the resource was obtained via *traversal*, the URL path will not include the virtual root prefix (it will be stripped off the left hand side of the generated URL).

i For backwards compatibility purposes, this function can also be imported as `model_url`, although doing so will emit a deprecation warning.

route_url (*route_name*, *request*, **elements*, ***kw*)

Generates a fully qualified URL for a named Pyramid *route configuration*.

i Calling `pyramid.Request.route_url()` can be used to achieve the same result as `pyramid.url.route_url()`.

Use the route’s name as the first positional argument. Use a request object as the second positional argument. Additional positional arguments are appended to the URL as path segments after it is generated.

Use keyword arguments to supply values which match any dynamic path elements in the route definition. Raises a `KeyError` exception if the URL cannot be generated for any reason (not enough arguments, for example).

For example, if you’ve defined a route named “foobar” with the path `{foo}/{bar}/*traverse`:

```
route_url('foobar', request, foo='1')           => <KeyError exception>
route_url('foobar', request, foo='1', bar='2') => <KeyError exception>
route_url('foobar', request, foo='1', bar='2',
          traverse=('a','b'))                   => http://e.com/1/2/a/b
route_url('foobar', request, foo='1', bar='2',
          traverse='/a/b')                       => http://e.com/1/2/a/b
```

Values replacing `:segment` arguments can be passed as strings or Unicode objects. They will be encoded to UTF-8 and URL-quoted before being placed into the generated URL.

Values replacing `*remainder` arguments can be passed as strings *or* tuples of Unicode/string values. If a tuple is passed as a `*remainder` replacement value, its values are URL-quoted and encoded to UTF-8. The resulting strings are joined with slashes and rendered into the URL. If a string is passed as a `*remainder` replacement value, it is tacked on to the URL untouched.

If a keyword argument `_query` is present, it will be used to compose a query string that will be tacked on to the end of the URL. The value of `_query` must be a sequence of two-tuples or a data structure with an `.items()` method that returns a sequence of two-tuples (presumably a dictionary). This data structure will be turned into a query string per the documentation of `pyramid.encode.urlencode()` function. After the query data is turned into a query string, a leading `?` is prepended, and the resulting string is appended to the generated URL.

i Python data structures that are passed as `_query` which are sequences or dictionaries are turned into a string under the same rules as when run through `urllib.urlencode()` with the `doseq` argument equal to `True`. This means that sequences can be passed as values, and a `k=v` pair will be placed into the query string for each value.

If a keyword argument `_anchor` is present, its string representation will be used as a named anchor in the generated URL (e.g. if `_anchor` is passed as `foo` and the route URL is `http://example.com/route/url`, the resulting generated URL will be `http://example.com/route/url#foo`).

i If `_anchor` is passed as a string, it should be UTF-8 encoded. If `_anchor` is passed as a Unicode object, it will be converted to UTF-8 before being appended to the URL. The anchor value is not quoted in any way before being appended to the generated URL.

If both `_anchor` and `_query` are specified, the anchor element will always follow the query element, e.g. `http://example.com?foo=1#bar`.

If a keyword `_app_url` is present, it will be used as the protocol/hostname/port/leading path prefix of the generated URL. For example, using an `_app_url` of `http://example.com:8080/foo` would cause the URL `http://example.com:8080/foo/fleeb/flub` to be returned from this function if the expansion of the route pattern associated with the `route_name` expanded to `/fleeb/flub`. If `_app_url` is not specified, the result of `request.application_url` will be used as the prefix (the default).

This function raises a `KeyError` if the URL cannot be generated due to missing replacement names. Extra replacement names are ignored.

If the route object which matches the `route_name` argument has a *pregenerator*, the `*elements` and `**kw` arguments arguments passed to this function might be augmented or changed.

current_route_url (*request*, **elements*, ***kw*)

Generates a fully qualified URL for a named Pyramid *route configuration* based on the ‘current route’.

This function supplements `pyramid.url.route_url()`. It presents an easy way to generate a URL for the ‘current route’ (defined as the route which matched when the request was generated).

The arguments to this function have the same meaning as those with the same names passed to `pyramid.url.route_url()`. It also understands an extra argument which `route_url` does not named `_route_name`.

The route name used to generate a URL is taken from either the `_route_name` keyword argument or the name of the route which is currently associated with the request if `_route_name` was not passed. Keys and values from the current request *matchdict* are combined with the *kw* arguments to form a set of defaults named *newkw*. Then `route_url(route_name, request, *elements, **newkw)` is called, returning a URL.

Examples follow.


If the ‘current route’ has the route pattern `/foo/{page}` and the current url path is `/foo/1`, the *matchdict* will be `{‘page’:‘1’}`. The result of `current_route_url(request)` in this situation will be `/foo/1`.

If the ‘current route’ has the route pattern `/foo/{page}` and the current current url path is `/foo/1`, the *matchdict* will be `{‘page’:‘1’}`. The result of `current_route_url(request, page=‘2’)` in this situation will be `/foo/2`.

Usage of the `_route_name` keyword argument: if our routing table defines routes `/foo/{action}` named ‘foo’ and `/foo/{action}/{page}` named `fooaction`, and the current url pattern is `/foo/view` (which has matched the `/foo/{action}` route), we may want to use the *matchdict* args to generate a URL to the `fooaction` route. In this scenario, `current_url(request, _route_name=‘fooaction’, page=‘5’)` Will return string like: `/foo/view/5`.

route_path (*route_name*, *request*, **elements*, ***kw*)

Generates a path (aka a ‘relative URL’, a URL minus the host, scheme, and port) for a named Pyramid *route configuration*.

 Calling `pyramid.Request.route_path()` can be used to achieve the same result as `pyramid.url.route_path()`.

This function accepts the same argument as `pyramid.url.route_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the path, query parameters, and anchor data are present in the returned string.

For example, if you’ve defined a route named ‘fooabar’ with the path `/ {foo} / {bar}`, this call to `route_path`:

```
route_path('foobar', request, foo='1', bar='2')
```

Will return the string `/1/2`.

i Calling `route_path('route', request)` is the same as calling `route_url('route', request, _app_url="")`. `route_path` is, in fact, implemented in terms of `route_url` in just this way. As a result, any `_app_url` pass within the `**kw` values to `route_path` will be ignored.

static_url (*path*, *request*, ***kw*)

Generates a fully qualified URL for a static *asset*. The asset must live within a location defined via the `pyramid.config.Configurator.add_static_view()` *configuration declaration* (see *Serving Static Assets*).

i Calling `pyramid.Request.static_url()` can be used to achieve the same result as `pyramid.url.static_url()`.

Example:

```
static_url('mypackage:static/foo.css', request) =>
  
http://example.com/static/foo.css
```

The `path` argument points at a file or directory on disk which a URL should be generated for. The `path` may be either a relative path (e.g. `static/foo.css`) or a *asset specification* (e.g. `mypackage:static/foo.css`). A path may not be an absolute filesystem path (a `ValueError` will be raised if this function is supplied with an absolute path).

The `request` argument should be a *request* object.

The purpose of the `**kw` argument is the same as the purpose of the `pyramid.url.route_url()` `**kw` argument. See the documentation for that function to understand the arguments which you can provide to it. However, typically, you don't need to pass anything as `*kw` when generating a static asset URL.

This function raises a `ValueError` if a static view definition cannot be found which matches the path specification.

urlencode (*query*, *doseq=True*)

An alternate implementation of Python's `stdlib urllib.urlencode` function which accepts unicode keys and values within the `query` dict/sequence; all Unicode keys and values are first converted to UTF-8 before being used to compose the query string.

The value of `query` must be a sequence of two-tuples representing key/value pairs *or* an object (often a dictionary) with an `.items()` method that returns a sequence of two-tuples representing key/value pairs.

For minimal calling convention backwards compatibility, this version of `urlencode` accepts *but ignores* a second argument conventionally named `doseq`. The Python `stdlib` version behaves differently when `doseq` is `False` and when a sequence is presented as one of the values. This version always behaves in the `doseq=True` mode, no matter what the value of the second argument.

See the Python `stdlib` documentation for `urllib.urlencode` for more information.

PYRAMID.VIEW

render_view_to_response (*context, request, name='', secure=True*)

Call the *view callable* configured with a *view configuration* that matches the *view name* name registered against the specified *context* and *request* and return a *response* object. This function will return `None` if a corresponding *view callable* cannot be found (when no *view configuration* matches the combination of *name / context / and request*).

If *secure* is `True`, and the *view callable* found is protected by a permission, the permission will be checked before calling the view function. If the permission check disallows view execution (based on the current *authorization policy*), a `pyramid.exceptions.Forbidden` exception will be raised. The exception's `args` attribute explains why the view access was disallowed.

If *secure* is `False`, no permission checking is done.

render_view_to_iterable (*context, request, name='', secure=True*)

Call the *view callable* configured with a *view configuration* that matches the *view name* name registered against the specified *context* and *request* and return an iterable object which represents the body of a response. This function will return `None` if a corresponding *view callable* cannot be found (when no *view configuration* matches the combination of *name / context / and request*). Additionally, this function will raise a `ValueError` if a view function is found and called but the view function's result does not have an `app_iter` attribute.

You can usually get the string representation of the return value of this function by calling `".join(iterable)`, or just use `pyramid.view.render_view()` instead.

If *secure* is `True`, and the view is protected by a permission, the permission will be checked before the view function is invoked. If the permission check disallows view execution (based on the current *authentication policy*), a `pyramid.exceptions.Forbidden` exception will be raised; its `args` attribute explains why the view access was disallowed.

If *secure* is `False`, no permission checking is done.

render_view (*context*, *request*, *name*='', *secure*=True)


Call the *view callable* configured with a *view configuration* that matches the *view name* name registered against the specified *context* and *request* and unwind the view response's *app_iter* (see *View Callable Responses*) into a single string. This function will return *None* if a corresponding *view callable* cannot be found (when no *view configuration* matches the combination of *name* / *context* / and *request*). Additionally, this function will raise a *ValueError* if a view function is found and called but the view function's result does not have an *app_iter* attribute. This function will return *None* if a corresponding view cannot be found.

If *secure* is *True*, and the view is protected by a permission, the permission will be checked before the view is invoked. If the permission check disallows view execution (based on the current *authorization policy*), a *pyramid.exceptions.Forbidden* exception will be raised; its *args* attribute explains why the view access was disallowed.

If *secure* is *False*, no permission checking is done.

is_response (*ob*)

Return *True* if *ob* implements the interface implied by *View Callable Responses*. *False* if not.

 This isn't a true interface or subclass check. Instead, it's a duck-typing check, as response objects are not obligated to be of a particular class or provide any particular Zope interface.

class view_config (*name*='', *request_type*=None, *for_*=None, *permission*=None, *route_name*=None, *request_method*=None, *request_param*=None, *containment*=None, *attr*=None, *renderer*=None, *wrapper*=None, *xhr*=False, *accept*=None, *header*=None, *path_info*=None, *custom_predicates*=(), *context*=None, *decorator*=None, *mapper*=None)

A function, class or method *decorator* which allows a developer to create view registrations nearer to a *view callable* definition than use *imperative configuration* to do the same.

For example, this code in a module *views.py*:

```
from resources import MyResource

@view_config(name='my_view', context=MyResource, permission='read',
             route_name='site1')
def my_view(context, request):
    return 'OK'
```

Might replace the following call to the *pyramid.config.Configurator.add_view()* method:

```
import views
from resources import MyResource
config.add_view(views.my_view, context=MyResource, name='my_view',
                permission='read', 'route_name='sitel')
```

The following arguments are supported as arguments to `pyramid.view.view_config`: `context`, `permission`, `name`, `request_type`, `route_name`, `request_method`, `request_param`, `containment`, `xhr`, `accept`, `header`, `path_info`, `custom_predicates`, `decorator`, and `mapper`.

The meanings of these arguments are the same as the arguments passed to `pyramid.config.Configurator.add_view()`.

See *View Configuration Using the @view_config Decorator* for details about using `view_config`.

class static (*root_dir*, *cache_max_age=3600*, *package_name=None*)

An instance of this class is a callable which can act as a Pyramid *view callable*; this view will serve static files from a directory on disk based on the `root_dir` you provide to its constructor.

The directory may contain subdirectories (recursively); the static view implementation will descend into these directories as necessary based on the components of the URL in order to resolve a path into a response.

You may pass an absolute or relative filesystem path or a *asset specification* representing the directory containing static files as the `root_dir` argument to this class' constructor.

If the `root_dir` path is relative, and the `package_name` argument is `None`, `root_dir` will be considered relative to the directory in which the Python file which *calls* `static` resides. If the `package_name` name argument is provided, and a relative `root_dir` is provided, the `root_dir` will be considered relative to the Python *package* specified by `package_name` (a dotted path to a Python package).

`cache_max_age` influences the `Expires` and `Max-Age` response headers returned by the view (default is 3600 seconds or five minutes).



If the `root_dir` is relative to a *package*, or is a *asset specification* the Pyramid `pyramid.config.Configurator` method can be used to override assets within the named `root_dir` package-relative directory. However, if the `root_dir` is absolute, configuration will not be able to override the assets it contains.

append_slash_notfound_view (*context, request*)

For behavior like Django's `APPEND_SLASH=True`, use this view as the *Not Found* view in your application.

When this view is the Not Found view (indicating that no view was found), and any routes have been defined in the configuration of your application, if the value of the `PATH_INFO` WSGI environment variable does not already end in a slash, and if the value of `PATH_INFO` *plus* a slash matches any route's path, do an HTTP redirect to the slash-appended `PATH_INFO`. Note that this will *lose* POST data information (turning it into a GET), so you shouldn't rely on this to redirect POST requests.

Use the `pyramid.config.Configurator.add_view()` method to configure this view as the Not Found view:

```
from pyramid.exceptions import NotFound
from pyramid.view import append_slash_notfound_view
config.add_view(append_slash_notfound_view, context=NotFound)
```

See also *Changing the Not Found View*.

class AppendSlashNotFoundViewFactory (*notfound_view=None*)

There can only be one *Not Found* view in any Pyramid application. Even if you use `pyramid.view.append_slash_notfound_view()` as the Not Found view, Pyramid still must generate a 404 Not Found response when it cannot redirect to a slash-appended URL; this not found response will be visible to site users.

If you don't care what this 404 response looks like, and you only need redirections to slash-appended route URLs, you may use the `pyramid.view.append_slash_notfound_view()` object as the Not Found view. However, if you wish to use a *custom* notfound view callable when a URL cannot be redirected to a slash-appended URL, you may wish to use an instance of this class as the Not Found view, supplying a *view callable* to be used as the custom notfound view as the first argument to its constructor. For instance:

```
from pyramid.exceptions import NotFound
from pyramid.view import AppendSlashNotFoundViewFactory

def notfound_view(context, request):
    return HTTPNotFound('It aint there, stop trying!')

custom_append_slash = AppendSlashNotFoundViewFactory(notfound_view)
config.add_view(custom_append_slash, context=NotFound)
```

The `notfound_view` supplied must adhere to the two-argument view callable calling convention of (`context, request`) (`context` will be the exception object).

PYRAMID.WSGI

`wsgiapp` (*wrapped*)

Decorator to turn a WSGI application into a Pyramid *view callable*. This decorator differs from the `pyramid.wsgi.wsgiapp2()` decorator inasmuch as fixups of `PATH_INFO` and `SCRIPT_NAME` within the WSGI environment *are not* performed before the application is invoked.

E.g., the following in a `views.py` module:

```
@wsgiapp
def hello_world(environ, start_response):
    body = 'Hello world'
    start_response('200 OK', [ ('Content-Type', 'text/plain'),
                               ('Content-Length', len(body)) ] )
    return [body]
```

Allows the following call to `pyramid.config.Configurator.add_view()`:

```
from views import hello_world
config.add_view(hello_world, name='hello_world.txt')
```

The `wsgiapp` decorator will convert the result of the WSGI application to a *Response* and return it to Pyramid as if the WSGI app were a pyramid view.

`wsgiapp2` (*wrapped*)

Decorator to turn a WSGI application into a Pyramid view callable. This decorator differs from the `pyramid.wsgi.wsgiapp()` decorator inasmuch as fixups of `PATH_INFO` and `SCRIPT_NAME` within the WSGI environment *are* performed before the application is invoked.

E.g. the following in a `views.py` module:

```
@wsgiapp2
def hello_world(environ, start_response):
    body = 'Hello world'
    start_response('200 OK', [ ('Content-Type', 'text/plain'),
                               ('Content-Length', len(body)) ] )
    return [body]
```

Allows the following call to `pyramid.config.Configurator.add_view()`:

```
from views import hello_world
config.add_view(hello_world, name='hello_world.txt')
```

The `wsgiapp2` decorator will convert the result of the WSGI application to a `Response` and return it to Pyramid as if the WSGI app were a Pyramid view. The `SCRIPT_NAME` and `PATH_INFO` values present in the WSGI environment are fixed up before the application is invoked.

Part IV

Glossary and Index

GLOSSARY

ACE An *access control entry*. An access control entry is one element in an *ACL*. An access control entry is a three-tuple that describes three things: an *action* (one of either `Allow` or `Deny`), a *principal* (a string describing a user or group), and a *permission*. For example the ACE, `(Allow, 'bob', 'read')` is a member of an *ACL* that indicates that the principal `bob` is allowed the permission `read` against the resource the *ACL* is attached to.

ACL An *access control list*. An *ACL* is a sequence of *ACE* tuples. An *ACL* is attached to a resource instance. An example of an *ACL* is `[(Allow, 'bob', 'read'), (Deny, 'fred', 'write')]`. If an *ACL* is attached to a resource instance, and that resource is findable via the context resource, it will be consulted any active security policy to determine wither a particular request can be fulfilled given the *authentication* information in the request.

Agendaless Consulting A consulting organization formed by Paul Everitt, Tres Seaver, and Chris McDonough. See also <http://agendaless.com>.

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

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

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

authentication The act of determining that the credentials a user presents during a particular request are “good”. Authentication in Pyramid is performed via an *authentication policy*.

authentication policy An authentication policy in Pyramid terms is a bit of code which has an API which determines the current *principal* (or principals) associated with a request.

authorization The act of determining whether a user can perform a specific action. In pyramid terms, this means determining whether, for a given resource, any *principal* (or principals) associated with the request have the requisite *permission* to allow the request to continue. Authorization in Pyramid is performed via its *authorization policy*.

authorization policy An authorization policy in Pyramid terms is a bit of code which has an API which determines whether or not the principals associated with the request can perform an action associated with a permission, based on the information found on the *context* resource.

Babel A collection of tools for internationalizing Python applications. Pyramid does not depend on Babel to operate, but if Babel is installed, additional locale functionality becomes available to your application.

Chameleon chameleon is an attribute language template compiler which supports both the *ZPT* and *Genshi* templating specifications. It is written and maintained by Malthe Borch. It has several extensions, such as the ability to use bracketed (Genshi-style) `{name}` syntax, even within *ZPT*. It is also much faster than the reference implementations of both *ZPT* and *Genshi*. Pyramid offers Chameleon templating out of the box in *ZPT* and text flavors.

configuration declaration An individual method call made to an instance of a Pyramid *Configurator* object which performs an arbitrary action, such as registering a *view configuration* (via the `add_view` method of the configurator) or *route configuration* (via the `add_route` method of the configurator). A set of configuration declarations is also implied by the *configuration decoration* detected by a *scan* of code in a package.

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

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

context A resource in the resource tree that is found during *traversal* or *URL dispatch* based on URL data; if it's found via traversal, it's usually a *resource* object that is part of a resource tree; if it's found via *URL dispatch*, it's an object manufactured on behalf of the route's "factory". A context resource becomes the subject of a *view*, and often has security information attached to it. See the *Traversal* chapter and the *URL Dispatch* chapter for more information about how a URL is resolved to a context resource.

CPython The C implementation of the Python language. This is the reference implementation that most people refer to as simply "Python"; *Jython*, Google's App Engine, and PyPy are examples of non-C based Python implementations.

declarative configuration The configuration mode in which you use *ZCML* to make a set of *configuration declaration* statements. See *pyramid_zcml*.

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

Default Locale Name The *locale name* used by an application when no explicit locale name is set. See *Localization-Related Deployment Settings*.

default permission A *permission* which is registered as the default for an entire application. When a default permission is in effect, every *view configuration* registered with the system will be effectively amended with a `permission` argument that will require that the executing user possess the default permission in order to successfully execute the associated *view callable*. See also *Setting a Default Permission*.

Default view The default view of a *resource* is the view invoked when the *view name* is the empty string (`""`). This is the case when *traversal* exhausts the path elements in the `PATH_INFO` of a request before it returns a *context* resource.

Deployment settings Deployment settings are settings passed to the *Configurator* as a `settings` argument. These are later accessible via a `request.registry.settings` dictionary. Deployment settings can be used as global application values.

distribution (Setuptools/distutils terminology). A file representing an installable library or application. Distributions are usually files that have the suffix of `.egg`, `.tar.gz`, or `.zip`. Distributions are the target of Setuptools commands such as `easy_install`.

distutils The standard system for packaging and distributing Python packages. See <http://docs.python.org/distutils/index.html> for more information. *setuptools* is actually an *extension* of the Distutils.

Django A full-featured Python web framework.

domain model Persistent data related to your application. For example, data stored in a relational database. In some applications, the *resource tree* acts as the domain model.

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

entry point A *setuptools* indirection, defined within a setuptools *distribution* `setup.py`. It is usually a name which refers to a function somewhere in a package which is held by the distribution.

event An object broadcast to zero or more *subscriber* callables during normal Pyramid system operations during the lifetime of an application. Application code can subscribe to these events by using the subscriber functionality described in *Using Events*.

Exception view An exception view is a *view callable* which may be invoked by Pyramid when an exception is raised during request processing. See *Exception Views* for more information.

finished callback A user-defined callback executed by the *router* unconditionally at the very end of request processing . See *Using Finished Callbacks*.

Forbidden view An *exception view* invoked by Pyramid when the developer explicitly raises a `pyramid.exceptions.Forbidden` exception from within *view* code or *root factory* code, or when the *view configuration* and *authorization policy* found for a request disallows a particular view invocation. Pyramid provides a default implementation of a forbidden view; it can be overridden. See *Changing the Forbidden View*.

Genshi An XML templating language by Christopher Lenz.

Gettext The GNU gettext library, used by the Pyramid translation machinery.

Google App Engine Google App Engine (aka “GAE”) is a Python application hosting service offered by Google. Pyramid runs on GAE.

Grok A web framework based on Zope 3.

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

interface A Zope interface object. In Pyramid, an interface may be attached to a *resource* object or a *request* object in order to identify that the object is “of a type”. Interfaces are used internally by Pyramid to perform view lookups and other policy lookups. The ability to make use of an interface is exposed to an application programmers during *view configuration* via the `context` argument, the `request_type` argument and the `containment` argument. Interfaces are also exposed to application developers when they make use of the *event* system. Fundamentally, Pyramid programmers can think of an interface as something that they can attach to an object that stamps it with a “type” unrelated to its underlying Python type. Interfaces can also be used to describe the behavior of an object (its methods and attributes), but unless they choose to, Pyramid programmers do not need to understand or use this feature of interfaces.

Internationalization The act of creating software with a user interface that can potentially be displayed in more than one language or cultural context. Often shortened to “i18n” (because the word “internationalization” is I, 18 letters, then N). See also: *Localization*.

Jinja2 A text templating language by Armin Ronacher.

JSON JavaScript Object Notation is a data serialization format.

Jython A Python implementation written for the Java Virtual Machine.

lineage An ordered sequence of objects based on a “*location* -aware” resource. The lineage of any given *resource* is composed of itself, its parent, its parent’s parent, and so on. The order of the sequence is resource-first, then the parent of the resource, then its parent’s parent, and so on. The parent of a resource in a lineage is available as its `__parent__` attribute.

Locale Name A string like `en`, `en_US`, `de`, or `de_AT` which uniquely identifies a particular locale.

Locale Negotiator An object supplying a policy determining which *locale name* best represents a given *request*. It is used by the `pyramid.i18n.get_locale_name()`, and `pyramid.i18n.negotiate_locale_name()` functions, and indirectly by `pyramid.i18n.get_localizer()`. The `pyramid.i18n.default_locale_negotiator()` function is an example of a locale negotiator.

Localization The process of displaying the user interface of an internationalized application in a particular language or cultural context. Often shortened to “L10” (because the word “localization” is L, 10 letters, then N). See also: *Internationalization*.

Localizer An instance of the class `pyramid.i18n.Localizer` which provides translation and pluralization services to an application. It is retrieved via the `pyramid.i18n.get_localizer()` function.

location The path to an object in a *resource tree*. See *Location-Aware Resources* for more information about how to make a resource object *location-aware*.

Mako Mako is a template language language which refines the familiar ideas of componentized layout and inheritance using Python with Python scoping and calling semantics.

matchdict The dictionary attached to the *request* object as `request.matchdict` when a *URL dispatch* route has been matched. Its keys are names as identified within the route pattern; its values are the values matched by each pattern name.

Message Catalog A *gettext* `.mo` file containing translations.

Message Identifier A string used as a translation lookup key during localization. The `msgid` argument to a *translation string* is a message identifier. Message identifiers are also present in a *message catalog*.

METAL Macro Expansion for TAL, a part of *ZPT* which makes it possible to share common look and feel between templates.

middleware *Middleware* is a *WSGI* concept. It is a *WSGI* component that acts both as a server and an application. Interesting uses for middleware exist, such as caching, content-transport encoding, and other functions. See *WSGI.org* or *PyPI* to find middleware for your application.

mod_wsgi `mod_wsgi` is an Apache module developed by Graham Dumpleton. It allows *WSGI* applications (such as applications developed using Pyramid) to be served using the Apache web server.

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

multidict An ordered dictionary that can have multiple values for each key. Adds the methods `getall`, `getone`, `mixed`, and `add` to the normal dictionary interface. See <http://pythonpaste.org/webob/class-webob.multidict.MultiDict.html>

Not Found view An *exception view* invoked by Pyramid when the developer explicitly raises a `pyramid.exceptions.NotFound` exception from within *view* code or *root factory* code, or when the current request doesn't match any *view configuration*. Pyramid provides a default implementation of a not found view; it can be overridden. See *Changing the Not Found View*.

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

Paste Paste is a WSGI development and deployment system developed by Ian Bicking.

PasteDeploy PasteDeploy is a library used by Pyramid which makes it possible to configure *WSGI* components together declaratively within an `.ini` file. It was developed by Ian Bicking as part of *Paste*.

permission A string or unicode object that represents an action being taken against a *context* resource. A permission is associated with a view name and a resource type by the developer. Resources are decorated with security declarations (e.g. an *ACL*), which reference these tokens also. Permissions are used by the active to security policy to match the view permission against the resources's statements about which permissions are granted to which principal in a context in order to to answer the question "is this user allowed to do this". Examples of permissions: `read`, or `view_blog_entries`.

pipeline The *Paste* term for a single configuration of a WSGI server, a WSGI application, with a set of middleware in-between.

pkg_resources A module which ships with *setuptools* that provides an API for addressing "asset files" within a Python *package*. Asset files are static files, template files, etc; basically anything non-Python-source that lives in a Python package can be considered a asset file. See also `PkgResources`

predicate A test which returns `True` or `False`. Two different types of predicates exist in Pyramid: a *view predicate* and a *route predicate*. View predicates are attached to *view configuration* and route predicates are attached to *route configuration*.

pregenerator A pregenerator is a function associated by a developer with a *route*. It is called by `pyramid.url.route_url()` in order to adjust the set of arguments passed to it by the user for special purposes. It will influence the URL returned by `route_url`. See `pyramid.interfaces.IRoutePregenerator` for more information.

principal A *principal* is a string or unicode object representing a userid or a group id. It is provided by an *authentication policy*. For example, if a user had the user id “bob”, and Bob was part of two groups named “group foo” and “group bar”, the request might have information attached to it that would indicate that Bob was represented by three principals: “bob”, “group foo” and “group bar”.

project (Setuptools/distutils terminology). A directory on disk which contains a `setup.py` file and one or more Python packages. The `setup.py` file contains code that allows the package(s) to be installed, distributed, and tested.

Pylons A lightweight Python web framework and a predecessor of Pyramid.

PyPI The Python Package Index, a collection of software available for Python.

Pyramid Cookbook An additional documentation resource for Pyramid which presents topical, practical usages of Pyramid available via <http://docs.pylonsproject.org/>.

pyramid_handlers An add-on package which allows Pyramid users to create classes that are analogues of Pylons 1 “controllers”. See http://docs.pylonsproject.org/projects/pyramid_handlers/dev/.

pyramid_jinja2 *Jinja2* templating system bindings for Pyramid, documented at http://docs.pylonsproject.org/projects/pyramid_jinja2/dev/. This package also includes a paster template named `pyramid_jinja2_starter`, which creates an application package based on the Jinja2 templating system.

pyramid_sqla A package which provides a Pylons-esque paster template which sports support for *view handler* application development, *SQLAlchemy* support, and other Pylons-like features. See https://bytebucket.org/sluggo/pyramid_sqla/wiki/html/index.html for more information.

pyramid_zcml An add-on package to Pyramid which allows applications to be configured via ZCML. It is available on *PyPI*. If you use `pyramid_zcml`, you can use ZCML as an alternative to *imperative configuration*.

Python The programming language in which Pyramid is written.

renderer A serializer that can be referred to via *view configuration* which converts a non-*Response* return values from a *view* into a string (and ultimately a response). Using a renderer can make writing views that require templating or other serialization less tedious. See *Writing View Callables Which Use a Renderer* for more information.

renderer factory A factory which creates a *renderer*. See *Adding and Changing Renderers* for more information.

renderer globals Values injected as names into a renderer based on application policy. See *Adding Renderer Globals* for more information.

Repoze “Repoze” is essentially a “brand” of software developed by Agendaless Consulting and a set of contributors. The term has no special intrinsic meaning. The project’s website has more information. The software developed “under the brand” is available in a Subversion repository. Pyramid was originally known as `repoze.bfg`.

repoze.catalog An indexing and search facility (fielded and full-text) based on `zope.index`. See the documentation for more information.

repoze.lemonade Zope2 CMF-like data structures and helper facilities for CA-and-ZODB-based applications useful within Pyramid applications.

repoze.who Authentication middleware for *WSGI* applications. It can be used by Pyramid to provide authentication information.

repoze.workflow Barebones workflow for Python apps . It can be used by Pyramid to form a workflow system.

request A `WebOb` request object. See *Request and Response Objects* (narrative) and `pyramid.request` (API documentation) for information about request objects.

request factory An object which, provided a *WSGI* environment as a single positional argument, returns a `WebOb` compatible request.

request type An attribute of a *request* that allows for specialization of view invocation based on arbitrary categorization. The every *request* object that Pyramid generates and manipulates has one or more *interface* objects attached to it. The default interface attached to a request object is `pyramid.interfaces.IRequest`.

resource An object representing a node in the *resource tree* of an application. If `traversal` is used, a resource is an element in the resource tree traversed by the system. When `traversal` is used, a resource becomes the *context* of a *view*. If `url_dispatch` is used, a single resource is generated for each request and is used as the context resource of a view.

Resource Location The act of locating a *context* resource given a *request*. *Traversal* and *URL dispatch* are the resource location subsystems used by Pyramid.

resource tree A nested set of dictionary-like objects, each of which is a *resource*. The act of *traversal* uses the resource tree to find a *context* resource.

-
- response** An object that has three attributes: `app_iter` (representing an iterable body), `headerlist` (representing the http headers sent to the user agent), and `status` (representing the http status string sent to the user agent). This is the interface defined for `WebOb` response objects. See *Request and Response Objects* for information about response objects.
- response callback** A user-defined callback executed by the *router* at a point after a *response* object is successfully created. See *Using Response Callbacks*.
- reStructuredText** A plain text format that is the defacto standard for descriptive text shipped in *distribution* files, and Python docstrings. This documentation is authored in ReStructuredText format.
- root** The object at which *traversal* begins when Pyramid searches for a *context* resource (for *URL Dispatch*, the root is *always* the context resource unless the `traverse=` argument is used in route configuration).
- root factory** The “root factory” of a Pyramid application is called on every request sent to the application. The root factory returns the traversal root of an application. It is conventionally named `get_root`. An application may supply a root factory to Pyramid during the construction of a *Configurator*. If a root factory is not supplied, the application uses a default root object. Use of the default root object is useful in application which use *URL dispatch* for all URL-to-view code mappings.
- route** A single pattern matched by the *url dispatch* subsystem, which generally resolves to a *root factory* (and then ultimately a *view*). See also *url dispatch*.
- route configuration** Route configuration is the act of associating request parameters with a particular *route* using pattern matching and *route predicate* statements. See *URL Dispatch* for more information about route configuration.
- route predicate** An argument to a *route configuration* which implies a value that evaluates to `True` or `False` for a given *request*. All predicates attached to a *route configuration* must evaluate to `True` for the associated route to “match” the current request. If a route does not match the current request, the next route (in definition order) is attempted.
- router** The *WSGI* application created when you start a Pyramid application. The router intercepts requests, invokes traversal and/or URL dispatch, calls view functions, and returns responses to the WSGI server on behalf of your Pyramid application.
- Routes** A system by Ben Bangert which parses URLs and compares them against a number of user defined mappings. The URL pattern matching syntax in Pyramid is inspired by the Routes syntax (which was inspired by Ruby On Rails pattern syntax).
- routes mapper** An object which compares path information from a request to an ordered set of route patterns. See *URL Dispatch*.

- scan** The term used by Pyramid to define the process of importing and examining all code in a Python package or module for *configuration decoration*.
- session** A namespace that is valid for some period of continual activity that can be used to represent a user's interaction with a web application.
- session factory** A callable, which, when called with a single argument named `request` (a *request* object), returns a *session* object.
- setuptools** Setuptools builds on Python's `distutils` to provide easier building, distribution, and installation of libraries and applications.
- SQLAlchemy** SQLAlchemy is an object relational mapper used in tutorials within this documentation.
- subpath** A list of element "left over" after the *router* has performed a successful traversal to a view. The subpath is a sequence of strings, e.g. `['left', 'over', 'names']`. Within Pyramid applications that use URL dispatch rather than traversal, you can use `*subpath` in the route pattern to influence the subpath. See *Using *subpath in a Route Pattern* for more information.
- subscriber** A callable which receives an *event*. A callable becomes a subscriber via *imperative configuration* or via *configuration decoration*. See *Using Events* for more information.
- template** A file with replaceable parts that is capable of representing some text, XML, or HTML when rendered.
- thread local** A thread-local variable is one which is essentially a global variable in terms of how it is accessed and treated, however, each thread used by the application may have a different value for this same "global" variable. Pyramid uses a small number of thread local variables, as described in *Thread Locals*. See also the `threading.local` documentation for more information.
- Translation Directory** A translation directory is a *gettext* translation directory. It contains language folders, which themselves contain `LC_MESSAGES` folders, which contain `.mo` files. Each `.mo` file represents a set of translations for a language in a *translation domain*. The name of the `.mo` file (minus the `.mo` extension) is the translation domain name.
- Translation Domain** A string representing the "context" in which a translation was made. For example the word "java" might be translated differently if the translation domain is "programming-languages" than would be if the translation domain was "coffee". A translation domain is represented by a collection of `.mo` files within one or more *translation directory* directories.
- Translation String** An instance of `pyramid.i18n.TranslationString`, which is a class that behaves like a Unicode string, but has several extra attributes such as `domain`, `msgid`, and `mapping` for use during translation. Translation strings are usually created by hand within software, but are sometimes created on the behalf of the system for automatic template translation. For more information, see *Internationalization and Localization*.

Translator A callable which receives a *translation string* and returns a translated Unicode object for the purposes of internationalization. A *localizer* supplies a translator to a Pyramid application accessible via its `translate` method.

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

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

Venusian Venusian is a library which allows framework authors to defer decorator actions. Instead of taking actions when a function (or class) decorator is executed at import time, the action usually taken by the decorator is deferred until a separate “scan” phase. Pyramid relies on Venusian to provide a basis for its *scan* feature.

view Common vernacular for a *view callable*.

view callable A “view callable” is a callable Python object which is associated with a *view configuration*; it returns a *response* object. A view callable accepts a single argument: `request`, which will be an instance of a *request* object. An alternate calling convention allows a view to be defined as a callable which accepts a pair of arguments: `context` and `request`: this calling convention is useful for traversal-based applications in which a *context* is always very important. A view callable is the primary mechanism by which a developer writes user interface code within Pyramid. See *Views* for more information about Pyramid view callables.

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

View handler A view handler ties together `pyramid.config.Configurator.add_route()` and `pyramid.config.Configurator.add_view()` to make it more convenient to register a collection of views as a single class when using *url dispatch*. View handlers ship as part of the *pyramid_handlers* add-on package.

View Lookup The act of finding and invoking the “best” *view callable* given a *request* and a *context* resource.

- view mapper** A view mapper is a class which implements the `pyramid.interfaces.IViewMapperFactory` interface, which performs view argument and return value mapping. This is a plug point for extension builders, not normally used by “civilians”.
- view name** The “URL name” of a view, e.g `index.html`. If a view is configured without a name, its name is considered to be the empty string (which implies the *default view*).
- view predicate** An argument to a *view configuration* which evaluates to `True` or `False` for a given *request*. All predicates attached to a view configuration must evaluate to true for the associated view to be considered as a possible callable for a given request.
- virtual root** A resource object representing the “virtual” root of a request; this is typically the physical root object (the object returned by the application root factory) unless *Virtual Hosting* is in use.
- virtualenv** An isolated Python environment. Allows you to control which packages are used on a particular project by cloning your main Python. `virtualenv` was created by Ian Bicking.
- WebError** WSGI middleware which can display debuggable traceback information in the browser when an exception is raised by a Pyramid application. See <http://pypi.python.org/pypi/WebError> .
- WebOb** `WebOb` is a WSGI request/response library created by Ian Bicking.
- WebTest** `WebTest` is a package which can help you write functional tests for your WSGI application.
- WSGI** Web Server Gateway Interface. This is a Python standard for connecting web applications to web servers, similar to the concept of Java Servlets. Pyramid requires that your application be served as a WSGI application.
- ZCML** Zope Configuration Markup Language, an XML dialect used by Zope and `pyramid_zcml` for configuration tasks.
- ZCML declaration** The concrete use of a *ZCML directive* within a ZCML file.
- ZCML directive** A ZCML “tag” such as `<view>` or `<route>`.
- ZEO** Zope Enterprise Objects allows multiple simultaneous processes to access a single *ZODB* database.
- ZODB** Zope Object Database, a persistent Python object store.
- Zope** The Z Object Publishing Framework, a full-featured Python web framework.
- Zope Component Architecture** The Zope Component Architecture (aka *ZCA*) is a system which allows for application pluggability and complex dispatching based on objects which implement an *interface*. Pyramid uses the *ZCA* “under the hood” to perform view dispatching and other application configuration tasks.
- ZPT** The Zope Page Template templating language.

INDEX

Symbols

`__call__()` (IRoutePregenerator method), 448
`__call__()` (ISessionFactory method), 451
`__call__()` (IViewMapper method), 452
`__call__()` (IViewMapperFactory method), 452
`__contains__()` (IBeforeRender method), 446
`__contains__()` (ISession method), 449
`__delitem__()` (ISession method), 450
`__getitem__()` (IBeforeRender method), 446
`__getitem__()` (ISession method), 449
`__init__.py`, 42
`__iter__()` (ISession method), 451
`__len__()` (ISession method), 449
`__setitem__()` (IBeforeRender method), 447
`__setitem__()` (ISession method), 450

A

`absolute_asset_spec()` (Configurator method), 402
`accept` (Request attribute), 467
`accept_charset` (Request attribute), 467
`accept_encoding` (Request attribute), 467
`accept_language` (Request attribute), 467
`accept_ranges` (Response attribute), 479
access control entry, 179
access control list, 178
ACE, 179, 523
ACE (special), 181
ACL, 178, 523
ACL inheritance, 182
ACLAllowed (class in `pyramid.security`), 490
ACLAuthorizationPolicy (class in `pyramid.authorization`), 387
ACLDenied (class in `pyramid.security`), 490
`action()` (Configurator method), 400
activating
 translation, 210
`add_directive()` (Configurator method), 402
`add_finished_callback()` (Request method), 463, 467
`add_renderer()` (Configurator method), 403
`add_response_callback()` (Request method), 463, 468
`add_route`, 50
`add_route()` (Configurator method), 403
`add_settings()` (Configurator method), 410
`add_static_view`, 150
`add_static_view()` (Configurator method), 408
`add_subscriber()` (Configurator method), 410
`add_translation_dirs()` (Configurator method), 410
`add_view`, 132
`add_view()` (Configurator method), 410
advanced
 configuration, 249
`age` (Response attribute), 479
Agendaless Consulting, 3, 4, 523
Akkerman, Wichert, ix
ALL_PERMISSIONS (in module `pyramid.security`), 489
Allow (in module `pyramid.security`), 489
`allow` (Response attribute), 479
Allowed (class in `pyramid.security`), 490

app (IAApplicationCreated attribute), 445
app_iter (Response attribute), 479
app_iter_range() (Response method), 479
append_slash_notfound_view() (in module pyramid.view), 517
AppendSlashNotFoundViewFactory (class in pyramid.view), 518
application configuration, 12
application registry, 275, 523
application_url (Request attribute), 469
ApplicationCreated (class in pyramid.events), 424
asbool() (in module pyramid.settings), 491
assert_() (DummyTemplateRenderer method), 495
asset, 523
asset specification, 523
assets, 148
 overriding, 155, 266
Authenticated (in module pyramid.security), 489
authenticated_userid() (in module pyramid.security), 487
authentication, 523
authentication policy, 523
authentication policy (creating), 183
authorization, 523
authorization (Request attribute), 469
authorization policy, 175, 523
authorization policy (creating), 184
AuthTktAuthenticationPolicy (class in pyramid.authentication), 389
AuthTktCookieHelper (class in pyramid.authentication), 391
automatic reloading of templates, 121

B

Babel, 201, 207, 524
 message extractors, 201
Bangert, Ben, ix
Beaker, 169
before render event, 241
BeforeRender (class in pyramid.events), 425
begin() (Configurator method), 399
bfg2pyramid, 370
Bicking, Ian, ix, 157

blank() (pyramid.request.Request class method), 469
body (Request attribute), 469
body (Response attribute), 479
body_file (Request attribute), 469
body_file (Response attribute), 480
book audience, vii
book content overview, vii
Borch, Malthe, ix
Brandl, Georg, ix
built-in renderers, 98

C

cache_control (Request attribute), 469
cache_control (Response attribute), 480
call_application() (Request method), 469
Chameleon, 524
 translation strings, 208
chameleon
 renderer, 100
Chameleon text templates, 117
Chameleon ZPT macros, 116
Chameleon ZPT templates, 115
changed() (ISession method), 451
charset (Request attribute), 470
charset (Response attribute), 480
cleanUp() (in module pyramid.testing), 494
clear() (ISession method), 451
clone() (DummyResource method), 494
code scanning, 14
commit() (Configurator method), 400
compiling
 message catalog, 205
conditional_response_app() (Response method), 480
configuration
 advanced, 249
configuration declaration, 524
configuration decoration, 14, 524
configuration decorator, 248
ConfigurationError (class in pyramid.exceptions), 427
Configurator, 19

-
- configurator, 524
 - Configurator (class in `pyramid.config`), 397
 - Configurator testing API, 232
 - container resources, 137
 - `content_disposition` (Response attribute), 480
 - `content_encoding` (Response attribute), 480
 - `content_language` (Response attribute), 480
 - `content_length` (Request attribute), 470
 - `content_length` (Response attribute), 480
 - `content_location` (Response attribute), 480
 - `content_md5` (Response attribute), 480
 - `content_range` (Response attribute), 480
 - `content_type` (Request attribute), 470
 - `content_type` (Response attribute), 481
 - `content_type_params` (Response attribute), 481
 - context, 80, 524
 - context (Request attribute), 461
 - ContextFound (class in `pyramid.events`), 424
 - converting a BFG app, 370
 - cookies (Request attribute), 470
 - `copy()` (Request method), 470
 - `copy()` (Response method), 481
 - `copy_body()` (Request method), 470
 - `copy_get()` (Request method), 471
 - CPython, 524
 - created (ISession attribute), 450
 - creating a project, 26
 - cross-site request forgery attacks, prevention, 171
 - `current_route_url()` (in module `pyramid.url`), 510
- ## D
- date (Request attribute), 471
 - date (Response attribute), 481
 - date and currency formatting (i18n), 207
 - de la Guardia, Carlos, ix
 - debug settings, 221
 - `debug_all`, 221
 - `debug_authorization`, 221
 - `debug_notfound`, 221
 - debugging authorization failures, 183
 - debugging not found errors, 135
 - declarative configuration, 524
 - decorator, 524
 - Default Locale Name, 524
 - default permission, 525
 - Default view, 525
 - default view, 80
 - `default_locale_name`, 209, 221
 - `default_locale_negotiator()` (in module `pyramid.i18n`), 443
 - `delete_cookie()` (Response method), 481
 - Denied (class in `pyramid.security`), 490
 - Deny (in module `pyramid.security`), 489
 - DENY_ALL (in module `pyramid.security`), 489
 - Deployment settings, 525
 - `derive_view()` (Configurator method), 415
 - development install, 28
 - distribution, 525
 - distutils, 525
 - Django, 3, 4, 525
 - domain
 - translation, 197
 - domain model, 525
 - dotted Python name, 525
 - DummyRequest (class in `pyramid.testing`), 495
 - DummyResource (class in `pyramid.testing`), 494
 - DummyTemplateRenderer (class in `pyramid.testing`), 495
 - Duncan, Casey, ix
- ## E
- `effective_principals()` (in module `pyramid.security`), 487
 - `encode_content()` (Response method), 481
 - `end()` (Configurator method), 399
 - entry point, 525
 - `environ` (Response attribute), 481
 - environment variables, 221
 - etag (Response attribute), 481
 - event, 217, 525
 - Everitt, Paul, ix
 - Everyone (in module `pyramid.security`), 489
 - exception (Request attribute), 462
 - Exception view, 525
 - exception views, 92
 - expires (Response attribute), 481

extending an existing application, 263

extensible application, 262

extracting

 messages, 202

F

factory (IRoute attribute), 447

find_interface() (in module pyramid.traversal), 499

find_resource() (in module pyramid.traversal), 499

find_root() (in module pyramid.traversal), 500

finished callback, 242, 526

flash messages, 169

flash(), 170

flash() (ISession method), 449

Forbidden (class in pyramid.exceptions), 427

Forbidden view, 526

forbidden view, 182, 238

forget() (in module pyramid.security), 487

forms, views, and unicode, 94

framework, 3

frameworks vs. libraries, 3

from_file() (pyramid.request.Request class method), 471

from_file() (pyramid.response.Response class method), 481

Fulton, Jim, ix

functional testing, 228

functional tests, 235

G

generate() (IRoute method), 447

generating

 resource url, 140

generating route URLs, 62

generating static asset urls, 152

Genshi, 526

GET (Request attribute), 467

get() (BeforeRender method), 425

get() (IBeforeRender method), 446

get() (ISession method), 450

get_app() (in module pyramid.paster), 455

get_csrf_token() (ISession method), 449

get_current_registry, 271, 277, 280

get_current_registry() (in module pyramid.threadlocal), 497

get_current_request, 271

get_current_request() (in module pyramid.threadlocal), 497

get_locale_name, 206

get_locale_name() (in module pyramid.i18n), 443

get_localizer, 205

get_localizer() (in module pyramid.i18n), 443

get_renderer() (in module pyramid.renderers), 459

get_response() (Request method), 471

get_root() (in module pyramid.scripting), 485

get_settings() (Configurator method), 400

get_settings() (in module pyramid.settings), 491

get_template() (in module pyramid.chameleon_text), 393

get_template() (in module pyramid.chameleon_zpt), 395

getGlobalSiteManager, 280

getSiteManager, 275, 277

Gettext, 526

gettext, 200

getUtility, 275, 277

Google App Engine, 526

Grok, 526

H

Hardwick, Nat, ix

has_permission() (in module pyramid.security), 488

Hathaway, Shane, ix

headerlist (Response attribute), 482

headers (Request attribute), 471

headers (Response attribute), 482

helloworld (imperative), 19

Holth, Daniel, ix

hook_zca (configurator method), 279

hook_zca() (Configurator method), 399

host (Request attribute), 471

host_url (Request attribute), 471

http redirect (from a view), 91

HTTPAccepted (class in pyramid.httpexceptions), 434

-
- HTTPBadGateway (class in pyramid.httpexceptions), 440
 HTTPBadRequest (class in pyramid.httpexceptions), 436
 HTTPClientError (class in pyramid.httpexceptions), 433
 HTTPConflict (class in pyramid.httpexceptions), 438
 HTTPCreated (class in pyramid.httpexceptions), 434
 HTTPError (class in pyramid.httpexceptions), 433
 HTTPException (class in pyramid.httpexceptions), 433
 HTTPExpectationFailed (class in pyramid.httpexceptions), 439
 HTTPForbidden (class in pyramid.httpexceptions), 437
 HTTPFound (class in pyramid.httpexceptions), 435
 HTTPGatewayTimeout (class in pyramid.httpexceptions), 440
 HTTPGone (class in pyramid.httpexceptions), 438
 HTTPInternalServerError (class in pyramid.httpexceptions), 440
 HTTPLengthRequired (class in pyramid.httpexceptions), 438
 HTTPMethodNotAllowed (class in pyramid.httpexceptions), 437
 HTTPMovedPermanently (class in pyramid.httpexceptions), 435
 HTTPMultipleChoices (class in pyramid.httpexceptions), 435
 HTTPNoContent (class in pyramid.httpexceptions), 434
 HTTPNonAuthoritativeInformation (class in pyramid.httpexceptions), 434
 HTTPNotAcceptable (class in pyramid.httpexceptions), 437
 HTTPNotFound (class in pyramid.httpexceptions), 437
 HTTPNotImplemented (class in pyramid.httpexceptions), 440
 HTTPNotModified (class in pyramid.httpexceptions), 436
 HTTPOk (class in pyramid.httpexceptions), 433
 HTTPPartialContent (class in pyramid.httpexceptions), 435
 HTTPPaymentRequired (class in pyramid.httpexceptions), 437
 HTTPPreconditionFailed (class in pyramid.httpexceptions), 438
 HTTPProxyAuthenticationRequired (class in pyramid.httpexceptions), 437
 HTTPRedirection (class in pyramid.httpexceptions), 433
 HTTPRequestEntityTooLarge (class in pyramid.httpexceptions), 439
 HTTPRequestRangeNotSatisfiable (class in pyramid.httpexceptions), 439
 HTTPRequestTimeout (class in pyramid.httpexceptions), 438
 HTTPRequestURITooLong (class in pyramid.httpexceptions), 439
 HTTPResetContent (class in pyramid.httpexceptions), 435
 HTTPSeeOther (class in pyramid.httpexceptions), 436
 HTTPServerError (class in pyramid.httpexceptions), 434
 HTTPServiceUnavailable (class in pyramid.httpexceptions), 440
 HTTPTemporaryRedirect (class in pyramid.httpexceptions), 436
 HTTPUnauthorized (class in pyramid.httpexceptions), 436
 HTTPUnsupportedMediaType (class in pyramid.httpexceptions), 439
 HTTPUseProxy (class in pyramid.httpexceptions), 436
 HTTPVersionNotSupported (class in pyramid.httpexceptions), 440
 |
 i18n, 195
 IApplicationCreated (interface in pyramid.interfaces), 445

- IBeforeRender (interface in pyramid.interfaces), 446
 - IContextFound (interface in pyramid.interfaces), 445
 - IExceptionResponse (interface in pyramid.interfaces), 447
 - if_match (Request attribute), 471
 - if_modified_since (Request attribute), 471
 - if_none_match (Request attribute), 471
 - if_range (Request attribute), 471
 - if_unmodified_since (Request attribute), 472
 - imperative configuration, 13, 19, 251, 526
 - implementation() (ITemplateRenderer method), 452
 - include() (Configurator method), 401
 - INewRequest, 217
 - INewRequest (interface in pyramid.interfaces), 445
 - INewResponse, 217
 - INewResponse (interface in pyramid.interfaces), 446
 - ini file, 35
 - ini file settings, 221
 - initializing
 - message catalog, 204
 - inside() (in module pyramid.location), 453
 - install preparation, 7
 - installing on Google App Engine, 12
 - installing on UNIX, 9
 - installing on Windows, 11
 - integration testing, 228
 - integration tests, 234
 - interactive shell, 29
 - interface, 526
 - Internationalization, 526
 - internationalization, 195
 - internationalization (of templates), 120
 - invalidate() (ISession method), 448
 - IPython, 29
 - IRendererInfo (interface in pyramid.interfaces), 451
 - IRoute (interface in pyramid.interfaces), 447
 - IRoutePregenerator (interface in pyramid.interfaces), 448
 - is_response() (in module pyramid.view), 516
 - is_xhr (Request attribute), 472
 - ISession (interface in pyramid.interfaces), 448
 - ISessionFactory (interface in pyramid.interfaces), 451
 - ITemplateRenderer (interface in pyramid.interfaces), 452
 - items() (DummyResource method), 494
 - items() (ISession method), 450
 - iteritems() (ISession method), 450
 - iterkeys() (ISession method), 450
 - itervalues() (ISession method), 449
 - IViewMapper (interface in pyramid.interfaces), 452
 - IViewMapperFactory (interface in pyramid.interfaces), 452
- ## J
- Jinja2, 122, 526
 - JSON, 526
 - renderer, 99
 - Jython, 526
- ## K
- keys() (DummyResource method), 494
 - keys() (ISession method), 450
 - Koym, Todd, ix
- ## L
- 110n, 195
 - Laflamme, Blaise, ix
 - last_modified (Response attribute), 482
 - leaf resources, 137
 - lineage, 526
 - lineage() (in module pyramid.location), 453
 - locale
 - negotiator, 210
 - Locale Name, 527
 - locale name, 206
 - Locale Negotiator, 527
 - locale_name (Localizer attribute), 442
 - Localization, 527
 - localization, 195
 - localization deployment settings, 209

Localizer, 527
 localizer, 205
 Localizer (class in pyramid.i18n), 442
 location, 527
 location (Response attribute), 482
 location-aware
 resource, 138
 security, 182

M

make_body_seekable() (Request method), 472
 make_wsgi_app, 21
 make_wsgi_app() (Configurator method), 416
 Mako, 120, 527
 mako
 renderer, 101
 match() (IRoute method), 448
 matchdict, 58, 527
 matchdict (Request attribute), 462
 matched_route, 59
 matched_route (Request attribute), 463
 matching the root URL, 62
 max_forwards (Request attribute), 472
 maybe_dotted() (Configurator method), 402
 md5_etag() (Response method), 482
 merge_cookies() (Response method), 482
 Message Catalog, 527
 message catalog
 compiling, 205
 initializing, 204
 updating, 204
 Message Identifier, 527
 message identifier, 197
 messages
 extracting, 202
 METAL, 527
 method (Request attribute), 472
 middleware, 527
 mod_wsgi, 31, 527
 model_url() (Request method), 472
 module, 528
 Moroz, Tom, ix
 msgid

 translation, 197
 multidict, 528
 multidict (WebOb), 164
 MVC, 4

N

name (IRendererInfo attribute), 451
 name (IRoute attribute), 447
 negotiate_locale_name, 206
 negotiate_locale_name() (in module pyramid.i18n), 443
 negotiator
 locale, 210
 new (ISession attribute), 449
 new_csrf_token() (ISession method), 449
 NewRequest, 217
 NewRequest (class in pyramid.events), 424
 NewResponse, 217
 NewResponse (class in pyramid.events), 425
 not found error (debugging), 135
 Not Found view, 528
 not found view, 237
 NotFound (class in pyramid.exceptions), 427

O

object tree, 78, 137
 Oram, Simon, ix
 Orr, Mike, ix
 override_asset, 156
 override_asset() (Configurator method), 416
 overriding
 assets, 155, 266
 routes, 265
 views, 265

P

package, 42, 528
 package (IRendererInfo attribute), 451
 params (Request attribute), 473
 Paste, 528
 PasteDeploy, 35, 528
 PasteDeploy settings, 221
 paster proutes, 66

- paster pshell, 29
 - paster serve, 31
 - paster templates, 25
 - path (Request attribute), 473
 - path_info (Request attribute), 473
 - path_info_peek() (Request method), 473
 - path_info_pop() (Request method), 473
 - path_qs (Request attribute), 473
 - path_url (Request attribute), 473
 - pattern (IRoute attribute), 447
 - peek_flash(), 171
 - peek_flash() (ISession method), 449
 - permission, 528
 - permission names, 181
 - permissions, 176
 - Peters, Tim, ix
 - pipeline, 528
 - pkg_resources, 528
 - pluralize() (Localizer method), 442
 - pluralizing (i18n), 206
 - pop() (ISession method), 449
 - pop_flash(), 171
 - pop_flash() (ISession method), 450
 - popitem() (ISession method), 450
 - POST (Request attribute), 467
 - postvars (Request attribute), 473
 - pragma (Request attribute), 474
 - pragma (Response attribute), 482
 - predicate, 528
 - predicates (IRoute attribute), 448
 - pregenerator, 528
 - pregenerator (IRoute attribute), 447
 - preventing cross-site request forgery attacks, 171
 - principal, 181, 529
 - principal names, 181
 - principals_allowed_by_permission() (in module pyramid.security), 488
 - printing
 - routes, 66
 - project, 26, 529
 - project structure, 34
 - protecting views, 176
 - PyLons, 3, 4, 529
 - PyPI, 529
 - pyramid and other frameworks, 4
 - Pyramid Cookbook, 529
 - pyramid genesis, viii
 - pyramid.authentication (module), 389
 - pyramid.authorization (module), 387
 - pyramid.chameleon_text (module), 393
 - pyramid.chameleon_zpt (module), 395
 - pyramid.config (module), 397
 - pyramid.events (module), 423
 - pyramid.exceptions (module), 427
 - pyramid.httpexceptions (module), 429
 - pyramid.i18n (module), 441
 - pyramid.interfaces (module), 445
 - pyramid.location (module), 453
 - pyramid.paster (module), 455
 - pyramid.registry (module), 457
 - pyramid.renderers (module), 459
 - pyramid.request (module), 461
 - pyramid.response (module), 479
 - pyramid.scripting (module), 485
 - pyramid.security (module), 487
 - pyramid.settings (module), 491
 - pyramid.testing, 232
 - pyramid.testing (module), 493
 - pyramid.threadlocal (module), 497
 - pyramid.traversal (module), 499
 - pyramid.url (module), 507
 - pyramid.view (module), 515
 - pyramid.wsgi (module), 519
 - pyramid_alchemy paster template, 25
 - pyramid_beaker, 169
 - pyramid_handlers, 529
 - pyramid_jinja2, 529
 - pyramid_routesalchemy paster template, 25
 - pyramid_sqla, 529
 - pyramid_starter paster template, 25
 - pyramid_zcml, 529
 - pyramid_zodb paster template, 25
 - Python, 529
- ## Q
- query_string (Request attribute), 474

- queryvars (Request attribute), 474
 quote_path_segment() (in module `pyramid.traversal`), 502
- ## R
- range (Request attribute), 474
 redirecting to slash-appended routes, 63
 referer (Request attribute), 474
 referrer (Request attribute), 474
 Registry (class in `pyramid.registry`), 457
 registry (Configurator attribute), 399
 registry (IRendererInfo attribute), 451
 registry (Request attribute), 461
 relative_url() (Request method), 474
 reload, 31, 221
 reload settings, 221
 reload_all, 221
 reload_assets, 221, 228
 reload_templates, 228
 remember() (in module `pyramid.security`), 488
 remote_addr (Request attribute), 474
 remote_user (Request attribute), 474
 RemoteUserAuthenticationPolicy (class in `pyramid.authentication`), 391
 remove_conditional_headers() (Request method), 474
 render() (in module `pyramid.renderers`), 459
 render_template() (in module `pyramid.chameleon_text`), 393
 render_template() (in module `pyramid.chameleon_zpt`), 395
 render_template_to_response() (in module `pyramid.chameleon_text`), 393
 render_template_to_response() (in module `pyramid.chameleon_zpt`), 395
 render_to_response() (in module `pyramid.renderers`), 460
 render_view() (in module `pyramid.view`), 516
 render_view_to_iterable() (in module `pyramid.view`), 515
 render_view_to_response() (in module `pyramid.view`), 515
 renderer, 97, 529
 chameleon, 100
 JSON, 99
 mako, 101
 string, 98
 renderer (adding), 103
 renderer (template), 113
 renderer factory, 529
 renderer globals, 240, 530
 renderer response headers, 102
 renderers (built-in), 98
 Repoze, 530
 repoze namespace package, 4
 repoze.bfg genesis, viii
 repoze.catalog, 530
 repoze.lemonade, 530
 repoze.who, 530
 repoze.workflow, 530
 repoze.zope2, viii
 RepozeWho1AuthenticationPolicy (class in `pyramid.authentication`), 391
 request, 530
 request (and unicode), 161
 Request (class in `pyramid.request`), 461
 request (IContextFound attribute), 446
 request (INewRequest attribute), 445
 request (INewResponse attribute), 446
 request (Response attribute), 482
 request attributes, 159
 request attributes (special), 160
 request factory, 239, 530
 request methods, 161
 request object, 159
 request type, 530
 request URLs, 160
 request.registry, 278
 RequestClass (Response attribute), 479
 resource, 530
 location-aware, 138
 resource API functions, 148
 resource interfaces, 133, 145
 Resource Location, 530
 resource tree, 78, 137, 530
 resource url

- generating, 140
- resource_path() (in module pyramid.traversal), 500
- resource_path_tuple() (in module pyramid.traversal), 501
- resource_url, 140
- resource_url() (in module pyramid.url), 507
- resource_url() (Request method), 465, 474
- resources.py, 44
- response, 90, 530
- Response (class in pyramid.response), 479
- response (creating), 163
- response (INewResponse attribute), 446
- response callback, 241, 531
- response exceptions, 164
- response headers, 163
- response headers (from a renderer), 102
- response object, 162
- ResponseClass (Request attribute), 467
- reStructuredText, 531
- retry_after (Response attribute), 482
- root, 531
- root (Request attribute), 461
- root factory, 80, 531
- root url (matching), 62
- Rossi, Chris, ix
- route, 531
- route configuration, 531
 - view callable, 50
- route factory, 54
- route ordering, 54
- route path pattern syntax, 51
- route predicate, 531
- route subpath, 194
- route URLs, 62
- route_path() (in module pyramid.url), 511
- route_path() (Request method), 465, 475
- route_url() (in module pyramid.url), 509
- route_url() (Request method), 464, 476
- router, 531
- Routes, 531
- routes
 - overriding, 265
 - printing, 66

- routes mapper, 531
- running an application, 31
- running tests, 28

S

- sample template, 116
- Sawyers, Andrew, ix
- scan, 531
- scan() (Configurator method), 417
- scheme (Request attribute), 476
- script_name (Request attribute), 476
- Seaver, Tres, ix
- security, 173
 - location-aware, 182
 - URL dispatch, 65
 - view, 134
- server (Response attribute), 482
- server_name (Request attribute), 476
- server_port (Request attribute), 476
- session, 165, 532
- session (Request attribute), 462, 477
- session factory, 169, 532
- set_cookie() (Response method), 483
- set_default_permission() (Configurator method), 418
- set_forbidden_view() (Configurator method), 417
- set_locale_negotiator() (Configurator method), 418
- set_notfound_view() (Configurator method), 418
- set_renderer_globals_factory() (Configurator method), 419
- set_request_factory() (Configurator method), 419
- set_session_factory() (Configurator method), 419
- setdefault() (ISession method), 450
- settings, 221
- settings (IRendererInfo attribute), 451
- settings (Registry attribute), 457
- setUp() (in module pyramid.testing), 493
- setup.py, 39
- setup.py develop, 28
- setup_registry() (Configurator method), 403
- setuptools, 532
- Shipman, John, ix
- special ACE, 181

- special permission names, 181
 - SQLAlchemy, 532
 - startup, 31
 - startup process, 267
 - static (class in `pyramid.view`), 517
 - static asset urls, 152
 - static assets view, 153
 - static assets, 148
 - `static_url()` (in module `pyramid.url`), 512
 - `static_url()` (Request method), 466, 477
 - status (Response attribute), 483
 - `status_code` (Response attribute), 483
 - `status_int` (Response attribute), 483
 - `status_map` (in module `pyramid.httpexceptions`), 433
 - `str_cookies` (Request attribute), 477
 - `str_GET` (Request attribute), 477
 - `str_params` (Request attribute), 478
 - `str_POST` (Request attribute), 477
 - `str_postvars` (Request attribute), 478
 - `str_queryvars` (Request attribute), 478
 - string
 - renderer, 98
 - subpath, 80, 532
 - subpath (Request attribute), 461
 - subpath (route), 194
 - subscriber, 217, 532
 - `subscriber()` (in module `pyramid.events`), 423
- T**
- `tearDown()` (in module `pyramid.testing`), 494
 - template, 532
 - template automatic reload, 121
 - template internationalization, 120
 - template renderer side effects, 118
 - template renderers, 113
 - template system bindings, 122
 - templates used as renderers, 113
 - templates used directly, 109
 - test setup, 230
 - test tear down, 230
 - `testing_add_renderer()` (Configurator method), 421
 - `testing_add_subscriber()` (Configurator method), 420
 - `testing_resources()` (Configurator method), 420
 - `testing_securitypolicy()` (Configurator method), 420
 - tests (running), 28
 - `tests.py`, 45
 - thread local, 532
 - thread locals, 271
 - `tmpl_context` (Request attribute), 462, 478
 - `translate()` (Localizer method), 442
 - translating (i18n), 205
 - translation
 - activating, 210
 - domain, 197
 - msgid, 197
 - translation directories, 200
 - Translation Directory, 532
 - translation directory, 210
 - Translation Domain, 532
 - Translation String, 532
 - translation string, 197
 - translation string factory, 199
 - translation strings
 - Chameleon, 208
 - TranslationString (class in `pyramid.i18n`), 441
 - TranslationStringFactory (class in `pyramid.i18n`), 442
 - Translator, 532
 - traversal, 533
 - traversal algorithm, 80
 - traversal details, 77
 - traversal examples, 83
 - traversal overview, 71
 - traversal tree, 78, 137
 - `traversal_path()` (in module `pyramid.traversal`), 504
 - `traverse()` (in module `pyramid.traversal`), 502
 - traversed (Request attribute), 462
 - traverser, 243
 - type (IRendererInfo attribute), 451
- U**
- ubody (Response attribute), 483

- unauthenticated_userid() (in module `pyramid.security`), 487
- unhook_zca() (Configurator method), 400
- unicode (and the request), 161
- unicode, views, and forms, 94
- unicode_body (Response attribute), 483
- unit testing, 228
- unittest, 230
- unset_cookie() (Response method), 483
- upath_info (Request attribute), 478
- update() (BeforeRender method), 425
- update() (IBeforeRender method), 447
- update() (ISession method), 450
- updating
 - message catalog, 204
- url (Request attribute), 478
- URL dispatch, 47, 533
 - security, 65
- url generation (traversal), 148
- url generator, 245
- urlargs (Request attribute), 478
- URLDecodeError (class in `pyramid.exceptions`), 427
- urlencode() (in module `pyramid.url`), 512
- urlvars (Request attribute), 478
- uscript_name (Request attribute), 478
- user_agent (Request attribute), 478
- V**
 - values() (DummyResource method), 494
 - values() (ISession method), 451
 - van Rossum, Guido, ix
 - vary (Response attribute), 483
 - Venusian, 533
 - view, 533
 - security, 134
 - view callable, 533
 - view calling convention, 88, 89
 - view class, 88
 - view configuration, 533
 - view exceptions, 92
 - view function, 88
 - View handler, 533
 - view http redirect, 91
 - View Lookup, 533
 - view lookup, 80, 123
 - view mapper, 246, 533
 - view name, 80, 534
 - view predicate, 534
 - view renderer, 97
 - view response, 90
 - view security, 134
 - view_config, 14
 - view_config (class in `pyramid.view`), 516
 - view_config decorator, 129
 - view_execution_permitted() (in module `pyramid.security`), 489
 - view_name (Request attribute), 462
 - views
 - overriding, 265
 - views, forms, and unicode, 94
 - virtual hosting, 214
 - virtual root, 216, 534
 - virtual_root (Request attribute), 462
 - virtual_root() (in module `pyramid.traversal`), 502
 - virtual_root_path (Request attribute), 462
 - virtualenv, 10, 534
- W**
 - WebError, 534
 - WebOb, 157, 534
 - WebTest, 534
 - with_package() (Configurator method), 402
 - WSGI, 534
 - WSGI application, 21
 - wsgiapp() (in module `pyramid.wsgi`), 519
 - wsgiapp2() (in module `pyramid.wsgi`), 519
 - www_authenticate (Response attribute), 483
- Z**
 - ZCA, 275
 - ZCA global API, 277
 - ZCA global registry, 280
 - ZCML, 534
 - ZCML declaration, 534
 - ZCML directive, 534

ZEO, 534
ZODB, 534
Zope, 3, 4, 534
Zope 2, viii
Zope 3, viii
Zope Component Architecture, 275, 534
zope.component, 275
ZPT, 534
ZPT macros, 116
ZPT templates (Chameleon), 115