

The Pyramid Web Framework

Version 1.5.1

Chris McDonough

| | |
|--|------------|
| 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 | ix |
| Audience | ix |
| Book Content | x |
| The Genesis of <code>repoze.bfg</code> | x |
| The Genesis of Pyramid | xi |
| Thanks | xi |
| I Narrative Documentation | 1 |
| 1 Pyramid Introduction | 3 |
| 1.1 What Makes Pyramid Unique | 4 |
| 1.1.1 Single-file applications | 5 |
| 1.1.2 Decorator-based configuration | 5 |
| 1.1.3 URL generation | 6 |
| 1.1.4 Static file serving | 6 |

| | | |
|----------|--|-----------|
| 1.1.5 | Fully Interactive Development | 6 |
| 1.1.6 | Debugging settings | 7 |
| 1.1.7 | Add-ons | 7 |
| 1.1.8 | Class-based and function-based views | 7 |
| 1.1.9 | Asset specifications | 8 |
| 1.1.10 | Extensible templating | 9 |
| 1.1.11 | Rendered views can return dictionaries | 9 |
| 1.1.12 | Event system | 10 |
| 1.1.13 | Built-in internationalization | 10 |
| 1.1.14 | HTTP caching | 10 |
| 1.1.15 | Sessions | 11 |
| 1.1.16 | Speed | 11 |
| 1.1.17 | Exception views | 11 |
| 1.1.18 | No singletons | 12 |
| 1.1.19 | View predicates and many views per route | 12 |
| 1.1.20 | Transaction management | 12 |
| 1.1.21 | Configuration conflict detection | 13 |
| 1.1.22 | Configuration extensibility | 13 |
| 1.1.23 | Flexible authentication and authorization | 14 |
| 1.1.24 | Traversal | 14 |
| 1.1.25 | Tweens | 14 |
| 1.1.26 | View response adapters | 15 |
| 1.1.27 | “Global” response object | 17 |
| 1.1.28 | Automating repetitive configuration | 17 |
| 1.1.29 | Programmatic Introspection | 18 |
| 1.1.30 | Python 3 Compatibility | 19 |
| 1.1.31 | Testing | 19 |
| 1.1.32 | Support | 19 |
| 1.1.33 | Documentation | 20 |
| 1.2 | What Is The Pylons Project? | 20 |
| 1.3 | Pyramid and Other Web Frameworks | 20 |
| 2 | Installing Pyramid | 23 |
| 2.1 | Before You Install | 23 |
| 2.1.1 | For Mac OS X Users | 24 |
| 2.1.2 | If You Don’t Yet Have A Python Interpreter (UNIX) | 24 |
| 2.1.3 | If You Don’t Yet Have A Python Interpreter (Windows) | 25 |
| 2.2 | Installing Pyramid on a UNIX System | 26 |
| 2.2.1 | Installing Setuptools | 26 |
| 2.2.2 | Installing the <code>virtualenv</code> Package | 27 |
| 2.2.3 | Creating the Virtual Python Environment | 27 |
| 2.2.4 | Installing Pyramid Into the Virtual Python Environment | 28 |
| 2.3 | Installing Pyramid on a Windows System | 28 |

| | | |
|----------|---|-----------|
| 2.4 | What Gets Installed | 29 |
| 3 | Creating Your First Pyramid Application | 31 |
| 3.1 | Hello World | 31 |
| 3.1.1 | Imports | 32 |
| 3.1.2 | View Callable Declarations | 33 |
| 3.1.3 | Application Configuration | 33 |
| 3.1.4 | Configurator Construction | 33 |
| 3.1.5 | Adding Configuration | 34 |
| 3.1.6 | WSGI Application Creation | 34 |
| 3.1.7 | WSGI Application Serving | 35 |
| 3.1.8 | Conclusion | 35 |
| 3.2 | References | 36 |
| 4 | Application Configuration | 37 |
| 4.1 | Imperative Configuration | 37 |
| 4.2 | Declarative Configuration | 38 |
| 4.3 | Summary | 39 |
| 5 | Creating a Pyramid Project | 41 |
| 5.1 | Scaffolds Included with Pyramid | 41 |
| 5.2 | Creating the Project | 42 |
| 5.3 | Installing your Newly Created Project for Development | 43 |
| 5.4 | Running The Tests For Your Application | 44 |
| 5.5 | Running The Project Application | 45 |
| 5.5.1 | Reloading Code | 46 |
| 5.6 | Viewing the Application | 47 |
| 5.6.1 | The Debug Toolbar | 48 |
| 5.7 | The Project Structure | 49 |
| 5.8 | The <code>MyProject</code> <i>Project</i> | 50 |
| 5.8.1 | <code>development.ini</code> | 51 |
| 5.8.2 | <code>production.ini</code> | 53 |
| 5.8.3 | <code>MANIFEST.in</code> | 54 |
| 5.8.4 | <code>setup.py</code> | 54 |
| 5.9 | The <code>myproject</code> <i>Package</i> | 56 |
| 5.9.1 | <code>__init__.py</code> | 57 |
| 5.9.2 | <code>views.py</code> | 57 |
| 5.9.3 | <code>static</code> | 59 |
| 5.9.4 | <code>templates/mytemplate.pt</code> | 59 |
| 5.9.5 | <code>tests.py</code> | 59 |
| 5.10 | Modifying Package Structure | 60 |
| 5.11 | Using the Interactive Shell | 61 |
| 5.12 | What Is This <code>pserve</code> Thing | 61 |
| 5.13 | Using an Alternate WSGI Server | 62 |

| | | |
|----------|--|-----------|
| 6 | Startup | 63 |
| 6.1 | The Startup Process | 63 |
| 6.2 | Deployment Settings | 67 |
| 7 | Request Processing | 69 |
| 8 | URL Dispatch | 75 |
| 8.1 | High-Level Operational Overview | 75 |
| 8.2 | Route Configuration | 75 |
| 8.2.1 | Configuring a Route to Match a View | 76 |
| 8.2.2 | Route Pattern Syntax | 77 |
| 8.2.3 | Route Declaration Ordering | 81 |
| 8.2.4 | Route Configuration Arguments | 81 |
| 8.3 | Route Matching | 82 |
| 8.3.1 | The Matchdict | 82 |
| 8.3.2 | The Matched Route | 83 |
| 8.4 | Routing Examples | 83 |
| 8.4.1 | Example 1 | 83 |
| 8.4.2 | Example 2 | 84 |
| 8.4.3 | Example 3 | 85 |
| 8.5 | Matching the Root URL | 85 |
| 8.6 | Generating Route URLs | 86 |
| 8.7 | Static Routes | 87 |
| 8.8 | External Routes | 88 |
| 8.9 | Redirecting to Slash-Appended Routes | 88 |
| 8.10 | Debugging Route Matching | 90 |
| 8.11 | Using a Route Prefix to Compose Applications | 90 |
| 8.12 | Custom Route Predicates | 92 |
| 8.13 | Route Factories | 95 |
| 8.14 | Using Pyramid Security With URL Dispatch | 96 |
| 8.15 | Route View Callable Registration and Lookup Details | 97 |
| 8.16 | References | 97 |
| 9 | Views | 99 |
| 9.1 | View Callables | 99 |
| 9.2 | Defining a View Callable as a Function | 100 |
| 9.3 | Defining a View Callable as a Class | 100 |
| 9.4 | View Callable Responses | 101 |
| 9.5 | Using Special Exceptions In View Callables | 101 |
| 9.5.1 | HTTP Exceptions | 102 |
| 9.5.2 | How Pyramid Uses HTTP Exceptions | 103 |
| 9.6 | Custom Exception Views | 103 |
| 9.7 | Using a View Callable to do an HTTP Redirect | 105 |
| 9.8 | Handling Form Submissions in View Callables (Unicode and Character Set Issues) | 105 |

| | | |
|-----------|--|------------|
| 9.9 | Alternate View Callable Argument/Calling Conventions | 107 |
| 9.10 | Passing Configuration Variables to a View | 109 |
| 9.11 | Pylons-1.0-Style “Controller” Dispatch | 109 |
| 10 | Renderers | 111 |
| 10.1 | Writing View Callables Which Use a Renderer | 112 |
| 10.2 | Built-In Renderers | 113 |
| 10.2.1 | string: String Renderer | 113 |
| 10.2.2 | JSON Renderer | 114 |
| 10.2.3 | JSONP Renderer | 116 |
| 10.3 | Varying Attributes of Rendered Responses | 117 |
| 10.4 | Adding and Changing Renderers | 118 |
| 10.4.1 | Adding a New Renderer | 119 |
| 10.4.2 | Adding a Default Renderer | 121 |
| 10.4.3 | Changing an Existing Renderer | 121 |
| 10.5 | Overriding A Renderer At Runtime | 122 |
| 11 | Templates | 123 |
| 11.1 | Using Templates Directly | 123 |
| 11.2 | System Values Used During Rendering | 126 |
| 11.3 | Templates Used as Renderers via Configuration | 127 |
| 11.4 | Debugging Templates | 129 |
| 11.5 | Automatically Reloading Templates | 129 |
| 11.6 | Available Add-On Template System Bindings | 130 |
| 12 | View Configuration | 131 |
| 12.1 | Mapping a Resource or URL Pattern to a View Callable | 131 |
| 12.1.1 | View Configuration Parameters | 132 |
| 12.1.2 | Adding View Configuration Using the @view_config Decorator | 140 |
| 12.1.3 | Adding View Configuration Using add_view() | 143 |
| 12.2 | @view_defaults Class Decorator | 144 |
| 12.2.1 | Configuring View Security | 147 |
| 12.2.2 | NotFound Errors | 148 |
| 12.3 | Influencing HTTP Caching | 148 |
| 12.4 | Debugging View Configuration | 149 |
| 13 | Static Assets | 151 |
| 13.1 | Understanding Asset Specifications | 151 |
| 13.2 | Serving Static Assets | 152 |
| 13.2.1 | Generating Static Asset URLs | 154 |
| 13.3 | Advanced: Serving Static Assets Using a View Callable | 155 |
| 13.3.1 | Root-Relative Custom Static View (URL Dispatch Only) | 156 |
| 13.3.2 | Registering A View Callable to Serve a “Static” Asset | 157 |
| 13.4 | Overriding Assets | 157 |

| | | |
|-----------|--|------------|
| 13.4.1 | The <code>override_asset</code> API | 158 |
| 14 | Request and Response Objects | 161 |
| 14.1 | Request | 162 |
| 14.1.1 | Special Attributes Added to the Request by Pyramid | 163 |
| 14.1.2 | URLs | 163 |
| 14.1.3 | Methods | 163 |
| 14.1.4 | Unicode | 164 |
| 14.1.5 | Multidict | 164 |
| 14.1.6 | Dealing With A JSON-Encoded Request Body | 164 |
| 14.1.7 | Cleaning Up After a Request | 165 |
| 14.1.8 | More Details | 166 |
| 14.2 | Response | 167 |
| 14.2.1 | Headers | 168 |
| 14.2.2 | Instantiating the Response | 168 |
| 14.2.3 | Exception Responses | 168 |
| 14.2.4 | More Details | 169 |
| 15 | Sessions | 171 |
| 15.1 | Using The Default Session Factory | 171 |
| 15.2 | Using a Session Object | 172 |
| 15.3 | Using Alternate Session Factories | 173 |
| 15.4 | Creating Your Own Session Factory | 174 |
| 15.5 | Flash Messages | 174 |
| 15.5.1 | Using the <code>session.flash</code> Method | 174 |
| 15.5.2 | Using the <code>session.pop_flash</code> Method | 175 |
| 15.5.3 | Using the <code>session.peek_flash</code> Method | 175 |
| 15.6 | Preventing Cross-Site Request Forgery Attacks | 176 |
| 15.6.1 | Using the <code>session.get_csrf_token</code> Method | 176 |
| 15.6.2 | Checking CSRF Tokens Manually | 177 |
| 15.6.3 | Checking CSRF Tokens With A View Predicate | 178 |
| 15.6.4 | Using the <code>session.new_csrf_token</code> Method | 178 |
| 16 | Using Events | 179 |
| 16.1 | Configuring an Event Listener Imperatively | 179 |
| 16.2 | Configuring an Event Listener Using a Decorator | 180 |
| 16.3 | An Example | 181 |
| 16.4 | Creating Your Own Events | 181 |
| 17 | Environment Variables and <code>.ini</code> File Settings | 185 |
| 17.1 | Reloading Templates | 185 |
| 17.2 | Reloading Assets | 186 |
| 17.3 | Debugging Authorization | 186 |
| 17.4 | Debugging Not Found Errors | 186 |

| | | |
|-----------|--|------------|
| 17.5 | Debugging Route Matching | 187 |
| 17.6 | Preventing HTTP Caching | 187 |
| 17.7 | Debugging All | 187 |
| 17.8 | Reloading All | 188 |
| 17.9 | Default Locale Name | 188 |
| 17.10 | Including Packages | 188 |
| 17.10.1 | pyramid.includes vs. pyramid.config.Configurator.include() | 189 |
| 17.11 | Explicit Tween Configuration | 190 |
| 17.11.1 | PasteDeploy Configuration vs. Plain-Python Configuration | 191 |
| 17.12 | Examples | 192 |
| 17.13 | Understanding the Distinction Between reload_templates and reload_assets | 193 |
| 17.14 | Adding A Custom Setting | 193 |
| 18 | Logging | 195 |
| 18.1 | Logging Configuration | 195 |
| 18.2 | Sending Logging Messages | 198 |
| 18.3 | Filtering log messages | 199 |
| 18.4 | Advanced Configuration | 200 |
| 18.5 | Logging Exceptions | 200 |
| 18.6 | Request Logging with Paste's TransLogger | 200 |
| 19 | PasteDeploy Configuration Files | 203 |
| 19.1 | PasteDeploy | 203 |
| 19.1.1 | Entry Points and PasteDeploy .ini Files | 204 |
| 19.1.2 | [DEFAULT] Section of a PasteDeploy .ini File | 207 |
| 20 | Command-Line Pyramid | 209 |
| 20.1 | Displaying Matching Views for a Given URL | 209 |
| 20.2 | The Interactive Shell | 211 |
| 20.2.1 | Extending the Shell | 212 |
| 20.2.2 | IPython or bpython | 214 |
| 20.3 | Displaying All Application Routes | 214 |
| 20.4 | Displaying "Tweens" | 215 |
| 20.5 | Invoking a Request | 216 |
| 20.6 | Using Custom Arguments to Python when Running p* Scripts | 217 |
| 20.7 | Showing All Installed Distributions and their Versions | 218 |
| 20.8 | Writing a Script | 218 |
| 20.8.1 | Changing the Request | 220 |
| 20.8.2 | Cleanup | 221 |
| 20.8.3 | Setting Up Logging | 221 |
| 20.9 | Making Your Script into a Console Script | 222 |
| 21 | Internationalization and Localization | 227 |
| 21.1 | Creating a Translation String | 227 |

| | | |
|-----------|---|------------|
| 21.1.1 | Using The <code>TranslationString</code> Class | 227 |
| 21.1.2 | Using the <code>TranslationStringFactory</code> Class | 229 |
| 21.2 | Working With <code>gettext</code> Translation Files | 230 |
| 21.2.1 | Installing <code>Lingua</code> and <code>Gettext</code> | 231 |
| 21.2.2 | Extracting Messages from Code and Templates | 232 |
| 21.2.3 | Initializing a Message Catalog File | 232 |
| 21.2.4 | Updating a Catalog File | 233 |
| 21.2.5 | Compiling a Message Catalog File | 233 |
| 21.3 | Using a Localizer | 234 |
| 21.3.1 | Performing a Translation | 234 |
| 21.3.2 | Performing a Pluralization | 235 |
| 21.4 | Obtaining the Locale Name for a Request | 236 |
| 21.5 | Performing Date Formatting and Currency Formatting | 236 |
| 21.6 | Chameleon Template Support for Translation Strings | 237 |
| 21.7 | Mako Pyramid I18N Support | 238 |
| 21.8 | Localization-Related Deployment Settings | 238 |
| 21.9 | “Detecting” Available Languages | 238 |
| 21.10 | Activating Translation | 239 |
| 21.10.1 | Adding a Translation Directory | 240 |
| 21.10.2 | Setting the Locale | 240 |
| 21.11 | Locale Negotiators | 241 |
| 21.11.1 | The Default Locale Negotiator | 241 |
| 21.11.2 | Using a Custom Locale Negotiator | 241 |
| 22 | Virtual Hosting | 243 |
| 22.1 | Hosting an Application Under a URL Prefix | 243 |
| 22.2 | Virtual Root Support | 244 |
| 22.3 | Further Documentation and Examples | 245 |
| 23 | Unit, Integration, and Functional Testing | 247 |
| 23.1 | Test Set Up and Tear Down | 248 |
| 23.1.1 | Test setup using a context manager | 250 |
| 23.1.2 | What? | 250 |
| 23.2 | Using the <code>Configurator</code> and <code>pyramid.testing</code> APIs in Unit Tests | 250 |
| 23.3 | Creating Integration Tests | 253 |
| 23.4 | Creating Functional Tests | 254 |
| 24 | Resources | 255 |
| 24.1 | Defining a Resource Tree | 256 |
| 24.2 | Location-Aware Resources | 257 |
| 24.3 | Generating The URL Of A Resource | 258 |
| 24.3.1 | Overriding Resource URL Generation | 259 |
| 24.4 | Generating the Path To a Resource | 260 |
| 24.5 | Finding a Resource by Path | 261 |

| | | |
|-----------|---|------------|
| 24.6 | Obtaining the Lineage of a Resource | 262 |
| 24.7 | Determining if a Resource is In The Lineage of Another Resource | 262 |
| 24.8 | Finding the Root Resource | 263 |
| 24.9 | Resources Which Implement Interfaces | 263 |
| 24.10 | Finding a Resource With a Class or Interface in Lineage | 265 |
| 24.11 | Pyramid API Functions That Act Against Resources | 266 |
| 25 | Hello Traversal World | 267 |
| 25.1 | Example requests | 268 |
| 26 | Much Ado About Traversal | 269 |
| 26.1 | URL Dispatch | 270 |
| 26.2 | Historical Refresher | 270 |
| 26.3 | Traversal (aka Resource Location) | 271 |
| 26.4 | What Is a “Resource”? | 272 |
| 26.5 | View Lookup | 272 |
| 26.6 | Use Cases | 273 |
| 27 | Traversal | 275 |
| 27.1 | Traversal Details | 276 |
| 27.2 | The Resource Tree | 277 |
| 27.3 | The Traversal Algorithm | 278 |
| 27.3.1 | A Description of The Traversal Algorithm | 279 |
| 27.3.2 | Traversal Algorithm Examples | 282 |
| 27.3.3 | Using Resource Interfaces In View Configuration | 284 |
| 27.4 | References | 286 |
| 28 | Security | 287 |
| 28.1 | Enabling an Authorization Policy | 288 |
| 28.1.1 | Enabling an Authorization Policy Imperatively | 288 |
| 28.2 | Protecting Views with Permissions | 289 |
| 28.2.1 | Setting a Default Permission | 289 |
| 28.3 | Assigning ACLs to your Resource Objects | 290 |
| 28.4 | Elements of an ACL | 291 |
| 28.5 | Special Principal Names | 293 |
| 28.6 | Special Permissions | 294 |
| 28.7 | Special ACEs | 294 |
| 28.8 | ACL Inheritance and Location-Awareness | 294 |
| 28.9 | Changing the Forbidden View | 295 |
| 28.10 | Debugging View Authorization Failures | 295 |
| 28.11 | Debugging Imperative Authorization Failures | 296 |
| 28.12 | Creating Your Own Authentication Policy | 296 |
| 28.13 | Creating Your Own Authorization Policy | 297 |
| 28.14 | Admonishment Against Secret-Sharing | 298 |

| | |
|---|------------|
| 29 Combining Traversal and URL Dispatch | 299 |
| 29.1 A Review of Non-Hybrid Applications | 299 |
| 29.1.1 URL Dispatch Only | 299 |
| 29.1.2 Traversal Only | 300 |
| 29.2 Hybrid Applications | 300 |
| 29.2.1 The Root Object for a Route Match | 302 |
| 29.2.2 Using <code>*traverse</code> In a Route Pattern | 302 |
| 29.2.3 Using the <code>traverse</code> Argument In a Route Definition | 305 |
| 29.2.4 Using <code>*subpath</code> in a Route Pattern | 306 |
| 29.3 Corner Cases | 307 |
| 29.3.1 Registering a Default View for a Route That Has a <code>view</code> Attribute | 307 |
| 29.3.2 Binding Extra Views Against a Route Configuration that Doesn't Have a <code>*traverse</code> Element In Its Pattern | 308 |
| 29.4 Generating Hybrid URLs | 308 |
| 30 Invoking a Subrequest | 311 |
| 31 Using Hooks | 317 |
| 31.1 Changing the Not Found View | 317 |
| 31.2 Changing the Forbidden View | 319 |
| 31.3 Changing the Request Factory | 321 |
| 31.4 Adding Methods or Properties to Request Object | 321 |
| 31.5 Using The Before Render Event | 323 |
| 31.6 Using Response Callbacks | 324 |
| 31.7 Using Finished Callbacks | 325 |
| 31.8 Changing the Traverser | 326 |
| 31.9 Changing How <code>pyramid.request.Request.resource_url()</code> Generates a URL | 327 |
| 31.10 Changing How Pyramid Treats View Responses | 329 |
| 31.11 Using a View Mapper | 331 |
| 31.12 Registering Configuration Decorators | 332 |
| 31.13 Registering Tweens | 334 |
| 31.13.1 Creating a Tween | 334 |
| 31.13.2 Registering an Implicit Tween Factory | 336 |
| 31.13.3 Suggesting Implicit Tween Ordering | 338 |
| 31.13.4 Explicit Tween Ordering | 339 |
| 31.13.5 Tween Conflicts and Ordering Cycles | 340 |
| 31.13.6 Displaying Tween Ordering | 340 |
| 31.14 Adding A Third Party View, Route, or Subscriber Predicate | 341 |
| 31.14.1 View and Route Predicates | 341 |
| 31.14.2 Subscriber Predicates | 343 |
| 32 Pyramid Configuration Introspection | 345 |
| 32.1 Using the Introspector | 345 |
| 32.2 Introspectable Objects | 346 |

| | | |
|-----------|---|------------|
| 32.3 | Pyramid Introspection Categories | 347 |
| 32.4 | Introspection in the Toolbar | 357 |
| 32.5 | Disabling Introspection | 358 |
| 33 | Extending An Existing Pyramid Application | 359 |
| 33.1 | The Difference Between “Extensible” and “Pluggable” Applications | 359 |
| 33.2 | Rules for Building An Extensible Application | 360 |
| 33.2.1 | Fundamental Plugpoints | 361 |
| 33.3 | Extending an Existing Application | 361 |
| 33.3.1 | If The Application Has Configuration Decorations | 361 |
| 33.3.2 | Extending the Application | 362 |
| 33.3.3 | Overriding Views | 363 |
| 33.3.4 | Overriding Routes | 364 |
| 33.3.5 | Overriding Assets | 364 |
| 34 | Advanced Configuration | 365 |
| 34.1 | Conflict Detection | 365 |
| 34.1.1 | Manually Resolving Conflicts | 367 |
| 34.1.2 | Automatic Conflict Resolution | 370 |
| 34.1.3 | Methods Which Provide Conflict Detection | 370 |
| 34.2 | Including Configuration from External Sources | 371 |
| 34.3 | Two-Phase Configuration | 371 |
| 34.4 | More Information | 372 |
| 35 | Extending Pyramid Configuration | 373 |
| 35.1 | Adding Methods to the Configurator via <code>add_directive</code> | 373 |
| 35.2 | Using <code>config.action</code> in a Directive | 374 |
| 35.3 | Adding Configuration Introspection | 377 |
| 35.3.1 | Introspectable Relationships | 378 |
| 36 | Creating Pyramid Scaffolds | 381 |
| 36.1 | Basics | 381 |
| 36.2 | Supporting Older Pyramid Versions | 383 |
| 36.3 | Examples | 384 |
| 37 | Upgrading Pyramid | 385 |
| 37.1 | Deprecation and Removal Policy | 386 |
| 37.2 | Consulting the Change History | 387 |
| 37.3 | Testing Your Application Under a New Pyramid Release | 387 |
| 37.4 | My Application Doesn’t Have Any Tests or Has Few Tests | 388 |
| 37.5 | Upgrading to the Very Latest Pyramid Release | 389 |
| 38 | Thread Locals | 391 |
| 38.1 | Why and How Pyramid Uses Thread Local Variables | 391 |

| | | |
|-----------|--|------------|
| 38.2 | Why You Shouldn't Abuse Thread Locals | 392 |
| 39 | Using the Zope Component Architecture in Pyramid | 395 |
| 39.1 | Using the ZCA Global API in a Pyramid Application | 396 |
| 39.1.1 | Disusing the Global ZCA API | 396 |
| 39.1.2 | Enabling the ZCA Global API by Using <code>hook_zca</code> | 397 |
| 39.1.3 | Enabling the ZCA Global API by Using The ZCA Global Registry | 398 |
| II | Tutorials | 401 |
| 40 | SQLAlchemy + URL Dispatch Wiki Tutorial | 403 |
| 40.1 | Background | 403 |
| 40.2 | Design | 404 |
| 40.2.1 | Overall | 404 |
| 40.2.2 | Models | 404 |
| 40.2.3 | Views | 404 |
| 40.2.4 | Security | 405 |
| 40.2.5 | Summary | 405 |
| 40.3 | Installation | 407 |
| 40.3.1 | Before You Begin | 407 |
| 40.3.2 | Making a Project | 409 |
| 40.3.3 | Installing the Project in Development Mode | 410 |
| 40.3.4 | Running the Tests | 410 |
| 40.3.5 | Exposing Test Coverage Information | 411 |
| 40.3.6 | Initializing the Database | 412 |
| 40.3.7 | Starting the Application | 413 |
| 40.3.8 | Decisions the <code>alchemy</code> Scaffold Has Made For You | 414 |
| 40.4 | Basic Layout | 415 |
| 40.4.1 | Application Configuration with <code>__init__.py</code> | 415 |
| 40.4.2 | View Declarations via <code>views.py</code> | 418 |
| 40.4.3 | Content Models with <code>models.py</code> | 419 |
| 40.5 | Defining the Domain Model | 422 |
| 40.5.1 | Making Edits to <code>models.py</code> | 422 |
| 40.5.2 | Changing <code>scripts/initializedb.py</code> | 423 |
| 40.5.3 | Installing the Project and re-initializing the Database | 424 |
| 40.5.4 | Viewing the Application in a Browser | 425 |
| 40.6 | Defining Views | 425 |
| 40.6.1 | Declaring Dependencies in Our <code>setup.py</code> File | 426 |
| 40.6.2 | Running <code>setup.py develop</code> | 427 |
| 40.6.3 | Changing the <code>views.py</code> File | 428 |
| 40.6.4 | Adding Templates | 433 |
| 40.6.5 | Adding Routes to <code>__init__.py</code> | 437 |
| 40.6.6 | Viewing the Application in a Browser | 438 |

| | | |
|-----------|---|------------|
| 40.7 | Adding Authorization | 439 |
| 40.7.1 | Access Control | 439 |
| 40.7.2 | Login, Logout | 443 |
| 40.7.3 | Seeing Our Changes | 447 |
| 40.7.4 | Viewing the Application in a Browser | 455 |
| 40.8 | Adding Tests | 455 |
| 40.8.1 | Testing the Models | 455 |
| 40.8.2 | Testing the Views | 456 |
| 40.8.3 | Functional tests | 456 |
| 40.8.4 | Viewing the results of all our edits to <code>tests.py</code> | 456 |
| 40.8.5 | Running the Tests | 461 |
| 40.9 | Distributing Your Application | 463 |
| 41 | ZODB + Traversal Wiki Tutorial | 465 |
| 41.1 | Background | 465 |
| 41.2 | Design | 466 |
| 41.2.1 | Overall | 466 |
| 41.2.2 | Models | 466 |
| 41.2.3 | Views | 466 |
| 41.2.4 | Security | 467 |
| 41.2.5 | Summary | 467 |
| 41.3 | Installation | 469 |
| 41.3.1 | Preparation | 469 |
| 41.3.2 | Make a Project | 469 |
| 41.3.3 | Install the Project in “Development Mode” | 470 |
| 41.3.4 | Run the Tests | 470 |
| 41.3.5 | Expose Test Coverage Information | 471 |
| 41.3.6 | Start the Application | 471 |
| 41.3.7 | Visit the Application in a Browser | 472 |
| 41.3.8 | Decisions the <code>zodb</code> Scaffold Has Made For You | 472 |
| 41.4 | Basic Layout | 472 |
| 41.4.1 | Application Configuration with <code>__init__.py</code> | 473 |
| 41.4.2 | Resources and Models with <code>models.py</code> | 474 |
| 41.4.3 | Views With <code>views.py</code> | 475 |
| 41.4.4 | Configuration in <code>development.ini</code> | 476 |
| 41.5 | Defining the Domain Model | 478 |
| 41.5.1 | Delete the Database | 478 |
| 41.5.2 | Edit <code>models.py</code> | 478 |
| 41.5.3 | Look at the Result of Our Edits to <code>models.py</code> | 479 |
| 41.5.4 | View the Application in a Browser | 480 |
| 41.6 | Defining Views | 480 |
| 41.6.1 | Declaring Dependencies in Our <code>setup.py</code> File | 481 |
| 41.6.2 | Adding View Functions | 482 |

| | | |
|------------|--|------------|
| 41.6.3 | Viewing the Result of all Our Edits to <code>views.py</code> | 486 |
| 41.6.4 | Adding Templates | 488 |
| 41.6.5 | Viewing the Application in a Browser | 492 |
| 41.7 | Adding Authorization | 492 |
| 41.7.1 | Access Control | 493 |
| 41.7.2 | Login, Logout | 496 |
| 41.7.3 | Seeing Our Changes | 500 |
| 41.7.4 | Viewing the Application in a Browser | 507 |
| 41.8 | Adding Tests | 507 |
| 41.8.1 | Test the Models | 507 |
| 41.8.2 | Test the Views | 508 |
| 41.8.3 | Functional tests | 508 |
| 41.8.4 | View the results of all our edits to <code>tests.py</code> | 508 |
| 41.8.5 | Running the Tests | 513 |
| 41.9 | Distributing Your Application | 514 |
| 42 | Running a Pyramid Application under <code>mod_wsgi</code> | 517 |
| III | API Documentation | 521 |
| 43 | <code>pyramid.authentication</code> | 523 |
| 43.1 | Authentication Policies | 523 |
| 43.2 | Helper Classes | 533 |
| 44 | <code>pyramid.authorization</code> | 535 |
| 45 | <code>pyramid.compat</code> | 537 |
| 46 | <code>pyramid.config</code> | 541 |
| 47 | <code>pyramid.decorator</code> | 581 |
| 48 | <code>pyramid.events</code> | 583 |
| 48.1 | Functions | 583 |
| 48.2 | Event Types | 584 |
| 49 | <code>pyramid.exceptions</code> | 589 |
| 50 | <code>pyramid.httpexceptions</code> | 591 |
| 50.1 | HTTP Exceptions | 591 |
| 51 | <code>pyramid.i18n</code> | 605 |
| 52 | <code>pyramid.interfaces</code> | 609 |

| | |
|---|------------|
| 52.1 Event-Related Interfaces | 609 |
| 52.2 Other Interfaces | 611 |
| 53 pyramid.location | 627 |
| 54 pyramid.paster | 629 |
| 55 pyramid.path | 631 |
| 56 pyramid.registry | 635 |
| 57 pyramid.renderers | 637 |
| 58 pyramid.request | 643 |
| 59 pyramid.response | 671 |
| 59.1 Functions | 677 |
| 60 pyramid.scaffolds | 679 |
| 61 pyramid.scripting | 681 |
| 62 pyramid.security | 683 |
| 62.1 Authentication API Functions | 683 |
| 62.2 Authorization API Functions | 684 |
| 62.3 Constants | 685 |
| 62.4 Return Values | 686 |
| 63 pyramid.session | 687 |
| 64 pyramid.settings | 693 |
| 65 pyramid.static | 695 |
| 66 pyramid.testing | 697 |
| 67 pyramid.threadlocal | 701 |
| 68 pyramid.traversal | 703 |
| 69 pyramid.tweens | 709 |
| 70 pyramid.url | 711 |
| 71 pyramid.view | 713 |
| 72 pyramid.wsgi | 717 |

| | |
|------------------------------|------------|
| IV Glossary and Index | 719 |
| Glossary | 721 |

Front Matter

The Pyramid Web Framework, Version 1.1

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, Patricio Paez, Michael Merickel, Eric Ongerth, Niall O'Higgins, Christoph Zwerschke, John Anderson, Atsushi Odagiri, Kirk Strauser, JD Navarro, Joe Dallago, Savoir-Faire Linux, Łukasz Fidosz, Christopher Lambacher, Claus Conrad, Chris Beelby, Phil Jenvey and a number of people with only pseudonyms on GitHub.

Cover Designer: Hugues Laflamme of Kemeneur.

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.

The *Logging* is adapted, with permission, from the Pylons documentation logging chapter, originally written by Phil Jenvey.

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>

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
```

Example blocks representing UNIX shell commands are prefixed with a `$` character, e.g.:

```
$ $VENV/bin/nosetests
```

(See *virtualenv* for the meaning of `$VENV`)

Example blocks representing Windows `cmd.exe` commands are prefixed with a drive letter and/or a directory name, e.g.:

```
c:\examples> %VENV%\Scripts\nosetests
```

(See *virtualenv* for the meaning of `%VENV%`)

Sometimes, when it's unknown which directory is current, Windows `cmd.exe` example block commands are prefixed only with a `>` character, e.g.:

```
> %VENV%\Scripts\nosetests
```

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> %VENV%\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.

When multiple objects are imported from the same package, the following convention is used:

```
from foo import (  
    bar,  
    baz,  
)
```

It may look unusual, but it has advantages:

- It allows one to swap out the higher-level package `foo` for something else that provides the similar API. An example would be swapping out one Database for another (e.g. graduating from SQLite to PostgreSQL).
- Looks more neat in cases where a large number of objects get imported from that package.
- Adding/removing imported objects from the package is quicker and results in simpler diffs.

Welcome to “The Pyramid Web 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 Documentation

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, Hugues Laflamme, Mike Orr, John Shipman, Chris Beelby, Patricio Paez, Simon Oram, Nat Hardwick, Ian Bicking, Jim Fulton, Michael Merickel, 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 Python developer to create 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.

Pyramid attempts to follow these design and engineering principles:

Simplicity Pyramid takes a “*pay only for what you eat*” approach. 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 tries to solve only 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 easier for us to maintain complete and up-to-date documentation. It is our goal that no aspect of Pyramid is undocumented.

Speed Pyramid is designed to provide noticeably fast execution for common tasks such as templating and simple response generation.

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”.

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

1.1 What Makes Pyramid Unique

Understandably, people don’t usually want to hear about squishy engineering principles, they want to hear about concrete stuff that solves their problems. With that in mind, what would make someone want to use Pyramid instead of one of the many other web frameworks available today? What makes Pyramid unique?

This is a hard question to answer, because there are lots of excellent choices, and it’s actually quite hard to make a wrong choice, particularly in the Python web framework market. But one reasonable answer is this: you can write very small applications in Pyramid without needing to know a lot. “What?”, you say, “that can’t possibly be a unique feature, lots of other web frameworks let you do that!” Well, you’re right. But unlike many other systems, you can also write very large applications in Pyramid if you learn a little more about it. Pyramid will allow you to become productive quickly, and will grow with you; it won’t hold you back when your application is small and it won’t get in your way when your application becomes large. “Well that’s fine,” you say, “lots of other frameworks let me write large apps too.” Absolutely. But other Python web frameworks don’t seamlessly let you do both. They seem to fall into two non-overlapping categories: frameworks for “small apps” and frameworks for “big apps”. The “small app” frameworks typically sacrifice “big app” features, and vice versa.

We don’t think it’s a universally reasonable suggestion to write “small apps” in a “small framework” and “big apps” in a “big framework”. You can’t really know to what size every application will eventually grow. We don’t really want to have to rewrite a previously small application in another framework when it gets “too big”. We believe the current binary distinction between frameworks for small and large applications is just false; a well-designed framework should be able to be good at both. Pyramid strives to be that kind of framework.

To this end, Pyramid provides a set of features that, combined, are unique amongst Python web frameworks. Lots of other frameworks contain some combination of these features; Pyramid of course actually stole many of them from those other frameworks. But Pyramid is the only one that has all of them in one place, documented appropriately, and useful a la carte without necessarily paying for the entire banquet. These are detailed below.

1.1.1 Single-file applications

You can write a Pyramid application that lives entirely in one Python file, not unlike existing Python microframeworks. This is beneficial for one-off prototyping, bug reproduction, and very small applications. These applications are easy to understand because all the information about the application lives in a single place, and you can deploy them without needing to understand much about Python distributions and packaging. Pyramid isn't really marketed as a microframework, but it allows you to do almost everything that frameworks that are marketed as micro offer in very similar ways.

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

def hello_world(request):
    return Response('Hello %(name)s!' % request.matchdict)

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

See also:

See also *Creating Your First Pyramid Application*.

1.1.2 Decorator-based configuration

If you like the idea of framework configuration statements living next to the code it configures, so you don't have to constantly switch between files to refer to framework configuration when adding new code, you can use Pyramid decorators to localize the configuration. For example:

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

@view_config(route_name='fred')
def fred_view(request):
    return Response('fred')
```

However, unlike some other systems, using decorators for Pyramid configuration does not make your application difficult to extend, test or reuse. The `view_config` decorator, for example, does not actually *change* the input or output of the function it decorates, so testing it is a “WYSIWYG” operation; you don’t need to understand the framework to test your own code, you just behave as if the decorator is not there. You can also instruct Pyramid to ignore some decorators, or use completely imperative configuration instead of decorators to add views. Pyramid decorators are inert instead of eager: you detect and activate them with a *scan*.

Example: *Adding View Configuration Using the @view_config Decorator.*

1.1.3 URL generation

Pyramid is capable of generating URLs for resources, routes, and static assets. Its URL generation APIs are easy to use and flexible. If you use Pyramid’s various APIs for generating URLs, you can change your configuration around arbitrarily without fear of breaking a link on one of your web pages.

Example: *Generating Route URLs.*

1.1.4 Static file serving

Pyramid is perfectly willing to serve static files itself. It won’t make you use some external web server to do that. You can even serve more than one set of static files in a single Pyramid web application (e.g. `/static` and `/static2`). You can also, optionally, place your files on an external web server and ask Pyramid to help you generate URLs to those files, so you can use Pyramid’s internal fileserving while doing development, and a faster static file server in production without changing any code.

Example: *Serving Static Assets.*

1.1.5 Fully Interactive Development

When developing a Pyramid application, several interactive features are available. Pyramid can automatically utilize changed templates when rendering pages and automatically restart the application to incorporate changed python code. Plain old `print()` calls used for debugging can display to a console.

Pyramid’s debug toolbar comes activated when you use a Pyramid scaffold to render a project. This toolbar overlays your application in the browser, and allows you access to framework data such as the routes configured, the last renderings performed, the current set of packages installed, SQLAlchemy queries run, logging data, and various other facts. When an exception occurs, you can use its interactive debugger to poke around right in your browser to try to determine the cause of the exception. It’s handy.

Example: *The Debug Toolbar.*

1.1.6 Debugging settings

Pyramid has debugging settings that allow you to print Pyramid runtime information to the console when things aren't behaving as you're expecting. For example, you can turn on "debug_notfound", which prints an informative message to the console every time a URL does not match any view. You can turn on "debug_authorization", which lets you know why a view execution was allowed or denied by printing a message to the console. These features are useful for those WTF moments.

There are also a number of commands that you can invoke within a Pyramid environment that allow you to introspect the configuration of your system: `proutes` shows all configured routes for an application in the order they'll be evaluated for matching; `pviews` shows all configured views for any given URL. These are also WTF-crushers in some circumstances.

Examples: *Debugging View Authorization Failures* and *Command-Line Pyramid*.

1.1.7 Add-ons

Pyramid has an extensive set of add-ons held to the same quality standards as the Pyramid core itself. Add-ons are packages which provide functionality that the Pyramid core doesn't. Add-on packages already exist which let you easily send email, let you use the Jinja2 templating system, let you use XML-RPC or JSON-RPC, let you integrate with jQuery Mobile, etc.

Examples: <http://docs.pylonsproject.org/en/latest/docs/pyramid.html#pyramid-add-on-documentation>

1.1.8 Class-based and function-based views

Pyramid has a structured, unified concept of a *view callable*. View callables can be functions, methods of classes, or even instances. When you add a new view callable, you can choose to make it a function or a method of a class; in either case, Pyramid treats it largely the same way. You can change your mind later, and move code between methods of classes and functions. A collection of similar view callables can be attached to a single class as methods, if that floats your boat, and they can share initialization code as necessary. All kinds of views are easy to understand and use and operate similarly. There is no phony distinction between them; they can be used for the same purposes.

Here's a view callable defined as a function:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(route_name='aview')
5 def aview(request):
6     return Response('one')
```

Here's a few views defined as methods of a class instead:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 class AView(object):
5     def __init__(self, request):
6         self.request = request
7
8     @view_config(route_name='view_one')
9     def view_one(self):
10         return Response('one')
11
12     @view_config(route_name='view_two')
13     def view_two(self):
14         return Response('two')
```

See also:

See also *@view_config Placement*.

1.1.9 Asset specifications

Asset specifications are strings that contain both a Python package name and a file or directory name, e.g. `MyPackage:static/index.html`. Use of these specifications is omnipresent in Pyramid. An asset specification can refer to a template, a translation directory, or any other package-bound static resource. This makes a system built on Pyramid extensible, because you don't have to rely on globals ("*the* static directory") or lookup schemes ("*the* ordered set of template directories") to address your files. You can move files around as necessary, and include other packages that may not share your system's templates or static files without encountering conflicts.

Because asset specifications are used heavily in Pyramid, we've also provided a way to allow users to override assets. Say you love a system that someone else has created with Pyramid but you just need to change "that one template" to make it all better. No need to fork the application. Just override the asset specification for that template with your own inside a wrapper, and you're good to go.

Examples: *Understanding Asset Specifications* and *Overriding Assets*.

1.1.10 Extensible templating

Pyramid has a structured API that allows for pluggability of “renderers”. Templating systems such as Mako, Genshi, Chameleon, and Jinja2 can be treated as renderers. Renderer bindings for all of these templating systems already exist for use in Pyramid. But if you’d rather use another, it’s not a big deal. Just copy the code from an existing renderer package, and plug in your favorite templating system. You’ll then be able to use that templating system from within Pyramid just as you’d use one of the “built-in” templating systems.

Pyramid does not make you use a single templating system exclusively. You can use multiple templating systems, even in the same project.

Example: *Using Templates Directly*.

1.1.11 Rendered views can return dictionaries

If you use a *renderer*, you don’t have to return a special kind of “webby” Response object from a view. Instead, you can return a dictionary, and Pyramid will take care of converting that dictionary to a Response using a template on your behalf. This makes the view easier to test, because you don’t have to parse HTML in your tests; just make an assertion instead that the view returns “the right stuff” in the dictionary it returns. You can write “real” unit tests instead of functionally testing all of your views.

For example, instead of returning a Response object from a `render_to_response` call:

```
1 from pyramid.renderers import render_to_response
2
3 def myview(request):
4     return render_to_response('myapp:templates/mytemplate.pt', {'a':1},
5                               request=request)
```

You can return a Python dictionary:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='myapp:templates/mytemplate.pt')
4 def myview(request):
5     return {'a':1}
```

When this view callable is called by Pyramid, the `{'a':1}` dictionary will be rendered to a response on your behalf. The string passed as `renderer=` above is an *asset specification*. It is in the form `packagename:directoryname/filename.ext`. In this case, it refers to the `mytemplate.pt` file in the `templates` directory within the `myapp` Python package. Asset specifications are omnipresent in Pyramid: see *Asset specifications* for more information.

Example: *Renderers*.

1.1.12 Event system

Pyramid emits *events* during its request processing lifecycle. You can subscribe any number of listeners to these events. For example, to be notified of a new request, you can subscribe to the `NewRequest` event. To be notified that a template is about to be rendered, you can subscribe to the `BeforeRender` event, and so forth. Using an event publishing system as a framework notification feature instead of hardcoded hook points tends to make systems based on that framework less brittle.

You can also use Pyramid's event system to send your *own* events. For example, if you'd like to create a system that is itself a framework, and may want to notify subscribers that a document has just been indexed, you can create your own event type (`DocumentIndexed` perhaps) and send the event via Pyramid. Users of this framework can then subscribe to your event like they'd subscribe to the events that are normally sent by Pyramid itself.

Example: *Using Events* and *Event Types*.

1.1.13 Built-in internationalization

Pyramid ships with internationalization-related features in its core: localization, pluralization, and creating message catalogs from source files and templates. Pyramid allows for a plurality of message catalog via the use of translation domains: you can create a system that has its own translations without conflict with other translations in other domains.

Example: *Internationalization and Localization*.

1.1.14 HTTP caching

Pyramid provides an easy way to associate views with HTTP caching policies. You can just tell Pyramid to configure your view with an `http_cache` statement, and it will take care of the rest:

```
@view_config(http_cache=3600) # 60 minutes
def myview(request): ....
```

Pyramid will add appropriate `Cache-Control` and `Expires` headers to responses generated when this view is invoked.

See the `add_view()` method's `http_cache` documentation for more information.

1.1.15 Sessions

Pyramid has built-in HTTP sessioning. This allows you to associate data with otherwise anonymous users between requests. Lots of systems do this. But Pyramid also allows you to plug in your own sessioning system by creating some code that adheres to a documented interface. Currently there is a binding package for the third-party Redis sessioning system that does exactly this. But if you have a specialized need (perhaps you want to store your session data in MongoDB), you can. You can even switch between implementations without changing your application code.

Example: *Sessions*.

1.1.16 Speed

The Pyramid core is, as far as we can tell, at least marginally faster than any other existing Python web framework. It has been engineered from the ground up for speed. It only does as much work as absolutely necessary when you ask it to get a job done. Extraneous function calls and suboptimal algorithms in its core codepaths are avoided. It is feasible to get, for example, between 3500 and 4000 requests per second from a simple Pyramid view on commodity dual-core laptop hardware and an appropriate WSGI server (`mod_wsgi` or `gunicorn`). In any case, performance statistics are largely useless without requirements and goals, but if you need speed, Pyramid will almost certainly never be your application's bottleneck; at least no more than Python will be a bottleneck.

Example: <http://blog.curiasolutions.com/the-great-web-framework-shootout/>

1.1.17 Exception views

Exceptions happen. Rather than deal with exceptions that might present themselves to a user in production in an ad-hoc way, Pyramid allows you to register an *exception view*. Exception views are like regular Pyramid views, but they're only invoked when an exception "bubbles up" to Pyramid itself. For example, you might register an exception view for the `Exception` exception, which will catch *all* exceptions, and present a pretty "well, this is embarrassing" page. Or you might choose to register an exception view for only specific kinds of application-specific exceptions, such as an exception that happens when a file is not found, or an exception that happens when an action cannot be performed because the user doesn't have permission to do something. In the former case, you can show a pretty "Not Found" page; in the latter case you might show a login form.

Example: *Custom Exception Views*.

1.1.18 No singletons

Pyramid is written in such a way that it requires your application to have exactly zero “singleton” data structures. Or, put another way, Pyramid doesn’t require you to construct any “mutable globals”. Or put even a different way, an import of a Pyramid application needn’t have any “import-time side effects”. This is esoteric-sounding, but if you’ve ever tried to cope with parameterizing a Django “settings.py” file for multiple installations of the same application, or if you’ve ever needed to monkey-patch some framework fixture so that it behaves properly for your use case, or if you’ve ever wanted to deploy your system using an asynchronous server, you’ll end up appreciating this feature. It just won’t be a problem. You can even run multiple copies of a similar but not identically configured Pyramid application within the same Python process. This is good for shared hosting environments, where RAM is at a premium.

1.1.19 View predicates and many views per route

Unlike many other systems, Pyramid allows you to associate more than one view per route. For example, you can create a route with the pattern `/items` and when the route is matched, you can shuffle off the request to one view if the request method is GET, another view if the request method is POST, etc. A system known as “view predicates” allows for this. Request method matching is the most basic thing you can do with a view predicate. You can also associate views with other request parameters such as the elements in the query string, the Accept header, whether the request is an XHR request or not, and lots of other things. This feature allows you to keep your individual views “clean”; they won’t need much conditional logic, so they’ll be easier to test.

Example: *View Configuration Parameters*.

1.1.20 Transaction management

Pyramid’s *scaffold* system renders projects that include a *transaction management* system, stolen from Zope. When you use this transaction management system, you cease being responsible for committing your data anymore. Instead, Pyramid takes care of committing: it commits at the end of a request or aborts if there’s an exception. Why is that a good thing? Having a centralized place for transaction management is a great thing. If, instead of managing your transactions in a centralized place, you sprinkle `session.commit` calls in your application logic itself, you can wind up in a bad place. Wherever you manually commit data to your database, it’s likely that some of your other code is going to run *after* your commit. If that code goes on to do other important things after that commit, and an error happens in the later code, you can easily wind up with inconsistent data if you’re not extremely careful. Some data will have been written to the database that probably should not have. Having a centralized commit point saves you from needing to think about this; it’s great for lazy people who also care about data integrity. Either the request completes successfully, and all changes are committed, or it does not, and all changes are aborted.

Also, Pyramid’s transaction management system allows you to synchronize commits between multiple databases, and allows you to do things like conditionally send email if a transaction commits, but otherwise keep quiet.

Example: *SQLAlchemy + URL Dispatch Wiki Tutorial* (note the lack of commit statements anywhere in application code).

1.1.21 Configuration conflict detection

When a system is small, it’s reasonably easy to keep it all in your head. But when systems grow large, you may have hundreds or thousands of configuration statements which add a view, add a route, and so forth. Pyramid’s configuration system keeps track of your configuration statements, and if you accidentally add two that are identical, or Pyramid can’t make sense out of what it would mean to have both statements active at the same time, it will complain loudly at startup time. It’s not dumb though: it will automatically resolve conflicting configuration statements on its own if you use the configuration `include()` system: “more local” statements are preferred over “less local” ones. This allows you to intelligently factor large systems into smaller ones.

Example: *Conflict Detection*.

1.1.22 Configuration extensibility

Unlike other systems, Pyramid provides a structured “include” mechanism (see `include()`) that allows you to combine applications from multiple Python packages. All the configuration statements that can be performed in your “main” Pyramid application can also be performed by included packages including the addition of views, routes, subscribers, and even authentication and authorization policies. You can even extend or override an existing application by including another application’s configuration in your own, overriding or adding new views and routes to it. This has the potential to allow you to create a big application out of many other smaller ones. For example, if you want to reuse an existing application that already has a bunch of routes, you can just use the `include` statement with a `route_prefix`; the new application will live within your application at a URL prefix. It’s not a big deal, and requires little up-front engineering effort.

For example:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     config = Configurator()
5     config.include('pyramid_jinja2')
6     config.include('pyramid_exclog')
7     config.include('some.other.guys.package', route_prefix='/someotherguy')
```

See also:

See also *Including Configuration from External Sources* and *Rules for Building An Extensible Application*.

1.1.23 Flexible authentication and authorization

Pyramid includes a flexible, pluggable authentication and authorization system. No matter where your user data is stored, or what scheme you'd like to use to permit your users to access your data, you can use a predefined Pyramid plugpoint to plug in your custom authentication and authorization code. If you want to change these schemes later, you can just change it in one place rather than everywhere in your code. It also ships with prebuilt well-tested authentication and authorization schemes out of the box. But what if you don't want to use Pyramid's built-in system? You don't have to. You can just write your own bespoke security code as you would in any other system.

Example: *Enabling an Authorization Policy*.

1.1.24 Traversal

Traversal is a concept stolen from *Zope*. It allows you to create a tree of resources, each of which can be addressed by one or more URLs. Each of those resources can have one or more *views* associated with it. If your data isn't naturally treelike (or you're unwilling to create a treelike representation of your data), you aren't going to find traversal very useful. However, traversal is absolutely fantastic for sites that need to be arbitrarily extensible: it's a lot easier to add a node to a tree than it is to shoehorn a route into an ordered list of other routes, or to create another entire instance of an application to service a department and glue code to allow disparate apps to share data. It's a great fit for sites that naturally lend themselves to changing departmental hierarchies, such as content management systems and document management systems. Traversal also lends itself well to systems that require very granular security ("Bob can edit *this* document" as opposed to "Bob can edit documents").

Examples: *Hello Traversal World* and *Much Ado About Traversal*.

1.1.25 Tweens

Pyramid has a sort of internal WSGI-middleware-ish pipeline that can be hooked by arbitrary add-ons named "tweens". The debug toolbar is a "tween", and the `pyramid_tm` transaction manager is also. Tweens are more useful than WSGI *middleware* in some circumstances because they run in the context of Pyramid itself, meaning you have access to templates and other renderers, a "real" request object, and other niceties.

Example: *Registering Tweens*.

1.1.26 View response adapters

A lot is made of the aesthetics of what *kinds* of objects you’re allowed to return from view callables in various frameworks. In a previous section in this document we showed you that, if you use a *renderer*, you can usually return a dictionary from a view callable instead of a full-on *Response* object. But some frameworks allow you to return strings or tuples from view callables. When frameworks allow for this, code looks slightly prettier, because fewer imports need to be done, and there is less code. For example, compare this:

```
1 def aview(request):  
2     return "Hello world!"
```

To this:

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

The former is “prettier”, right?

Out of the box, if you define the former view callable (the one that simply returns a string) in Pyramid, when it is executed, Pyramid will raise an exception. This is because “explicit is better than implicit”, in most cases, and by default, Pyramid wants you to return a *Response* object from a view callable. This is because there’s usually a heck of a lot more to a response object than just its body. But if you’re the kind of person who values such aesthetics, we have an easy way to allow for this sort of thing:

```
1 from pyramid.config import Configurator  
2 from pyramid.response import Response  
3  
4 def string_response_adapter(s):  
5     response = Response(s)  
6     response.content_type = 'text/html'  
7     return response  
8  
9 if __name__ == '__main__':  
10     config = Configurator()  
11     config.add_response_adapter(string_response_adapter, basestring)
```

Do that once in your Pyramid application at startup. Now you can return strings from any of your view callables, e.g.:

1. PYRAMID INTRODUCTION

```
1 def helloview(request):
2     return "Hello world!"
3
4 def goodbyeview(request):
5     return "Goodbye world!"
```

Oh noes! What if you want to indicate a custom content type? And a custom status code? No fear:

```
1 from pyramid.config import Configurator
2
3 def tuple_response_adapter(val):
4     status_int, content_type, body = val
5     response = Response(body)
6     response.content_type = content_type
7     response.status_int = status_int
8     return response
9
10 def string_response_adapter(body):
11     response = Response(body)
12     response.content_type = 'text/html'
13     response.status_int = 200
14     return response
15
16 if __name__ == '__main__':
17     config = Configurator()
18     config.add_response_adapter(string_response_adapter, basestring)
19     config.add_response_adapter(tuple_response_adapter, tuple)
```

Once this is done, both of these view callables will work:

```
1 def aview(request):
2     return "Hello world!"
3
4 def anotherview(request):
5     return (403, 'text/plain', "Forbidden")
```

Pyramid defaults to explicit behavior, because it's the most generally useful, but provides hooks that allow you to adapt the framework to localized aesthetic desires.

See also:

See also *Changing How Pyramid Treats View Responses*.

1.1.27 “Global” response object

“Constructing these response objects in my view callables is such a chore! And I’m way too lazy to register a response adapter, as per the prior section,” you say. Fine. Be that way:

```
1 def aview(request):
2     response = request.response
3     response.body = 'Hello world!'
4     response.content_type = 'text/plain'
5     return response
```

See also:

See also *Varying Attributes of Rendered Responses*.

1.1.28 Automating repetitive configuration

Does Pyramid’s configurator allow you to do something, but you’re a little adventurous and just want it a little less verbose? Or you’d like to offer up some handy configuration feature to other Pyramid users without requiring that we change Pyramid? You can extend Pyramid’s *Configurator* with your own directives. For example, let’s say you find yourself calling `pyramid.config.Configurator.add_view()` repetitively. Usually you can take the boring away by using existing shortcuts, but let’s say that this is a case where there is no such shortcut:

```
1 from pyramid.config import Configurator
2
3 config = Configurator()
4 config.add_route('xhr_route', '/xhr/{id}')
5 config.add_view('my.package.GET_view', route_name='xhr_route',
6                 xhr=True, permission='view', request_method='GET')
7 config.add_view('my.package.POST_view', route_name='xhr_route',
8                 xhr=True, permission='view', request_method='POST')
9 config.add_view('my.package.HEAD_view', route_name='xhr_route',
10                xhr=True, permission='view', request_method='HEAD')
```

Pretty tedious right? You can add a directive to the Pyramid configurator to automate some of the tedium away:

```
1 from pyramid.config import Configurator
2
3 def add_protected_xhr_views(config, module):
4     module = config.maybe_dotted(module)
5     for method in ('GET', 'POST', 'HEAD'):
6         view = getattr(module, 'xhr_%s_view' % method, None)
7         if view is not None:
8             config.add_view(view, route_name='xhr_route', xhr=True,
9                             permission='view', request_method=method)
10
11 config = Configurator()
12 config.add_directive('add_protected_xhr_views', add_protected_xhr_views)
```

Once that's done, you can call the directive you've just added as a method of the Configurator object:

```
1 config.add_route('xhr_route', '/xhr/{id}')
2 config.add_protected_xhr_views('my.package')
```

Your previously repetitive configuration lines have now morphed into one line.

You can share your configuration code with others this way too by packaging it up and calling `add_directive()` from within a function called when another user uses the `include()` method against your code.

See also:

See also *Adding Methods to the Configurator via `add_directive`*.

1.1.29 Programmatic Introspection

If you're building a large system that other users may plug code into, it's useful to be able to get an enumeration of what code they plugged in *at application runtime*. For example, you might want to show them a set of tabs at the top of the screen based on an enumeration of views they registered.

This is possible using Pyramid's *introspector*.

Here's an example of using Pyramid's introspector from within a view callable:

```

1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='bar')
5 def show_current_route_pattern(request):
6     introspector = request.registry.introspector
7     route_name = request.matched_route.name
8     route_intr = introspector.get('routes', route_name)
9     return Response(str(route_intr['pattern']))

```

See also:

See also *Pyramid Configuration Introspection*.

1.1.30 Python 3 Compatibility

Pyramid and most of its add-ons are Python 3 compatible. If you develop a Pyramid application today, you won't need to worry that five years from now you'll be backwatered because there are language features you'd like to use but your framework doesn't support newer Python versions.

1.1.31 Testing

Every release of Pyramid has 100% statement coverage via unit and integration tests, as measured by the coverage tool available on PyPI. It also has greater than 95% decision/condition coverage as measured by the `instrumental` tool available on PyPI. It is automatically tested by the Jenkins tool on Python 2.6, Python 2.7, Python 3.2 and PyPy after each commit to its GitHub repository. Official Pyramid add-ons are held to a similar testing standard. We still find bugs in Pyramid and its official add-ons, but we've noticed we find a lot more of them while working on other projects that don't have a good testing regime.

Example: <http://jenkins.pylonsproject.org/>

1.1.32 Support

It's our goal that no Pyramid question go unanswered. Whether you ask a question on IRC, on the Pylons-discuss maillist, or on StackOverflow, you're likely to get a reasonably prompt response. We don't tolerate "support trolls" or other people who seem to get their rocks off by berating fellow users in our various official support channels. We try to keep it well-lit and new-user-friendly.

Example: Visit `irc://freenode.net#pyramid` (the `#pyramid` channel on `irc.freenode.net` in an IRC client) or the pylons-discuss maillist at <http://groups.google.com/group/pylons-discuss/>.

1.1.33 Documentation

It's a constant struggle, but we try to maintain a balance between completeness and new-user-friendliness in the official narrative Pyramid documentation (concrete suggestions for improvement are always appreciated, by the way). We also maintain a “cookbook” of recipes, which are usually demonstrations of common integration scenarios, too specific to add to the official narrative docs. In any case, the Pyramid documentation is comprehensive.

Example: The rest of this documentation and the cookbook at http://docs.pylonsproject.org/projects/pyramid_cookbook/dev/.

1.2 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.

1.3 Pyramid and Other Web Frameworks

The first release of Pyramid's predecessor (named `repoze.bfg`) was made in July of 2008. At the end of 2010, we changed the name of `repoze.bfg` to Pyramid. 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 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. Out of the box, Pyramid supports imperative and decorator-based configuration; *ZCML* may be used via an add-on package named `pyramid_zcml`.

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 2.6 or better to run Pyramid.

Python Versions

As of this writing, Pyramid has been tested under Python 2.6, Python 2.7, Python 3.2, and Python 3.3. Pyramid does not run under any version of Python before 2.6.

Pyramid is known to run on all popular UNIX-like systems such as Linux, Mac OS X, and FreeBSD as well as on Windows platforms. It is also known to run on *PyPy* (1.9+).

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

Some Pyramid dependencies may attempt to build C extensions for performance speedups. If a compiler or Python headers are unavailable the dependency will fall back to using pure Python instead.

2.1.1 For Mac OS X Users

From Python.org:

Python comes pre-installed on Mac OS X, but due to Apple’s release cycle, it’s often one or even two years old. The overwhelming recommendation of the “MacPython” community is to upgrade your Python by downloading and installing a newer version from the Python standard release page.

It is recommended to download one of the *installer* versions, unless you prefer to install your Python through a package manager (e.g., macports or homebrew) or to build your Python from source.

Unless you have a need for a specific earlier version, it is recommended to install the latest 2.x or 3.x version of Python.

If you use an installer for your Python, then you can skip to the section *Installing Pyramid on a UNIX System*.

2.1.2 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.

Package Manager Method

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

For example, on a Debian or Ubuntu system, use the following command:

```
$ sudo apt-get install python2.7-dev
```

This command will install both the Python interpreter and its development header files. Note that the headers are required by some (optional) C extensions in software depended upon by Pyramid, not by Pyramid itself.

Once these steps are performed, the Python interpreter will usually be invocable via `python2.7` 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. For an explanation, see <https://github.com/Pylons/pyramid/issues/747>.

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, you can install a Python 2.7 interpreter from *source*, on the same system, using the following commands:

```
$ cd ~
$ mkdir tmp
$ mkdir opt
$ cd tmp
$ wget http://www.python.org/ftp/python/2.7.3/Python-2.7.3.tgz
$ tar xvzf Python-2.7.3.tgz
$ cd Python-2.7.3
$ ./configure --prefix=$HOME/opt/Python-2.7.3
$ make && make install
```

Once these steps are performed, the Python interpreter will be invokable via `$HOME/opt/Python-2.7.3/bin/python` from a shell prompt.

2.1.3 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 2.7-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:\Python27` 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. To do so, invoke `import setuptools` within the Python interpreter you’d like to run Pyramid under.

The following command will not display anything if setuptools is already installed:

```
$ python2.7 -c 'import setuptools'
```

Running the same command will yield the following output if setuptools is not yet installed:

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ImportError: No module named setuptools
```

If `import setuptools` raises an `ImportError` as it does above, you will need to install setuptools manually.

If you are using a “system” Python (one installed by your OS distributor or a third-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.

2.2.1 Installing Setuptools

To install setuptools by hand under Python 2, first download `ez_setup.py` then invoke it using the Python interpreter into which you want to install setuptools.

```
$ 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
```

2.2.2 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.



Python 3.3 includes `pyenv` out of the box, which provides similar functionality to `virtualenv`. We however suggest using `virtualenv` instead, which works well with Python 3.3. This isn't a recommendation made for technical reasons; it's made because it's not feasible for the authors of this guide to explain setup using multiple virtual environment systems. We are aiming to not need to make the installation documentation Turing-complete.

If you insist on using `pyenv`, you'll need to understand how to install software such as `setuptools` into the virtual environment manually, which this guide does not cover.

```
$ 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
```

2.2.3 Creating the Virtual Python Environment

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

```
$ export VENV=~/.env
$ virtualenv $VENV
New python executable in /home/foo/.env/bin/python
Installing setuptools.....done.
```

2. INSTALLING PYRAMID

You can either follow the use of the environment variable, `$VENV`, or replace it with the root directory of the *virtualenv*. In that case, the *export* command can be skipped. If you choose the former approach, ensure that it's an absolute path.



Avoid using the `--system-site-packages` option when creating the *virtualenv* unless you know what you are doing. For versions of *virtualenv* prior to 1.7, make sure to use the `--no-site-packages` option, because this option was formerly not the default and may produce undesirable results.



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

2.2.4 Installing Pyramid Into the Virtual Python Environment

After you've got your *virtualenv* installed, you may install Pyramid itself using the following commands:

```
$ $VENV/bin/easy_install ``pyramid==1.5.1``
```

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



If you see any warnings and/or errors related to failing to compile the C extensions, in most cases you may safely ignore those errors. If you wish to use the C extensions, please verify that you have a functioning compiler and the Python header files installed.

2.3 Installing Pyramid on a Windows System

You can use Pyramid on Windows under Python 2 or 3.

1. Download and install the most recent Python 2.7.x or 3.3.x version for your system.
2. Download and install the Python for Windows extensions. Carefully read the `README.txt` file at the end of the list of builds, and follow its directions. Make sure you get the proper 32- or 64-bit build and Python version.
3. Install latest *setuptools* distribution into the Python from step 1 above: download `ez_setup.py` and run it using the `python` interpreter of your Python 2.7 or 3.3 installation using a command prompt:

```
# modify the command according to the python version, e.g.:
# for Python 2.7:
c:\> c:\Python27\python ez_setup.py
# for Python 3.3:
c:\> c:\Python33\python ez_setup.py
```

4. Install *virtualenv*:

```
# modify the command according to the python version, e.g.:
# for Python 2.7:
c:\> c:\Python27\Scripts\easy_install virtualenv
# for Python 3.3:
c:\> c:\Python33\Scripts\easy_install virtualenv
```

5. Make a *virtualenv* workspace:

```
c:\> set VENV=c:\env
# modify the command according to the python version, e.g.:
# for Python 2.7:
c:\> c:\Python27\Scripts\virtualenv %VENV%
# for Python 3.3:
c:\> c:\Python33\Scripts\virtualenv %VENV%
```

You can either follow the use of the environment variable, `%VENV%`, or replace it with the root directory of the *virtualenv*. In that case, the `set` command can be skipped. If you choose the former approach, ensure that it's an absolute path.

6. (Optional) Consider using `%VENV%\Scripts\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:\env> %VENV%\Scripts\easy_install ``pyramid==1.5.1``
```

2.4 What Gets Installed

When you `easy_install` Pyramid, various other libraries such as WebOb, PasteDeploy, and others are installed.

Additionally, as chronicled in *Creating a Pyramid Project*, scaffolds will be registered, which make it easy to start a new Pyramid project.

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. It assumes you already have Pyramid installed. If you do not, head over to the *Installing Pyramid* section.

3.1 Hello World

Here's one of the very simplest Pyramid applications:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5
6 def hello_world(request):
7     return Response('Hello %(name)s!' % request.matchdict)
8
9 if __name__ == '__main__':
10     config = Configurator()
11     config.add_route('hello', '/hello/{name}')
12     config.add_view(hello_world, route_name='hello')
13     app = config.make_wsgi_app()
14     server = make_server('0.0.0.0', 8080, app)
15     server.serve_forever()
```

3. 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.

On UNIX:

```
$ $VENV/bin/python helloworld.py
```

On Windows:

```
C:\> %VENV%\Scripts\python.exe helloworld.py
```

This command will not return and nothing will be printed to the console. When port 8080 is visited by a browser on the URL `/hello/world`, the server will simply serve up the text “Hello world!”. If your application is running on your local system, using `http://localhost:8080/hello/world` in a browser will show this result.

Each time you visit a URL served by the application in a browser, a logging line will be emitted to the console displaying the hostname, the date, the request method and path, and some additional information. This output is done by the `wsgiref` server we’ve used to serve this application. It logs an “access log” in Apache combined logging format to the console.

Press `Ctrl-C` (or `Ctrl-Break` on Windows) to stop the application.

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

3.1.1 Imports

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

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
```

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.

Like many other Python web frameworks, Pyramid uses the *WSGI* protocol to connect an application and a web server together. The `wsgiref` server is used in this example as a WSGI server for convenience, as it is shipped within the Python standard library.

The script also imports the `pyramid.response.Response` class for later use. An instance of this class will be used to create a web response.

3.1.2 View Callable Declarations

The above script, beneath its set of imports, defines a function named `hello_world`.

```
1 def hello_world(request):  
2     return Response('Hello %(name)s!' % request.matchdict)
```

The function accepts a single argument (`request`) and it returns an instance of the `pyramid.response.Response` class. The single argument to the class' constructor is a string computed from parameters matched from the URL. This value becomes the body of the response.

This function 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 *WSGI* server which called Pyramid and it is 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, a string is passed as the body to the response.

3.1.3 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:

```
1 if __name__ == '__main__':  
2     config = Configurator()  
3     config.add_route('hello', '/hello/{name}')  
4     config.add_view(hello_world, route_name='hello')  
5     app = config.make_wsgi_app()  
6     server = make_server('0.0.0.0', 8080, app)  
7     server.serve_forever()
```

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

3.1.4 Configurator Construction

3. CREATING YOUR FIRST PYRAMID APPLICATION

```
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 operating system 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 command line.

Using the if clause is necessary – or at least best practice – because code in a Python `.py` file may be eventually imported via the Python `import` statement by another `.py` file. `.py` files that are imported by other `.py` files are referred to as *modules*. By using the `if __name__ == '__main__':` idiom, the script above is indicating that it does not want the code within the if statement to execute if this module is imported from another; 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 an *application registry* associated with the application.

3.1.5 Adding Configuration

```
1 config.add_route('hello', '/hello/{name}')
```

```
2 config.add_view(hello_world, route_name='hello')
```

First line above calls the `pyramid.config.Configurator.add_route()` method, which registers a *route* to match any URL path that begins with `/hello/` followed by a string.

The second line registers the `hello_world` function as a *view callable* and makes sure that it will be called when the `hello` route is matched.

3.1.6 WSGI Application Creation

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

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 sets 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, but 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 calls to its `add_view` and `add_route` methods.

3.1.7 WSGI Application Serving

```
1 server = make_server('0.0.0.0', 8080, app)
2 server.serve_forever()
```

Finally, we actually serve the application to requestors by starting up a WSGI server. We happen to use the `wsgiref.make_server` server maker for this purpose. We pass in as the first argument `'0.0.0.0'`, which means “listen on all TCP interfaces.” By default, the 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 also specify a TCP port number to listen on, which is 8080, passing it as the second argument. The final argument is the `app` object (a *router*), which is the application we wish to serve. Finally, we call the server's `serve_forever` method, which starts the main loop in which it will wait for requests from the outside world.

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

3.1.8 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.

3.2 References

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

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

Application Configuration

Most people already understand “configuration” as settings that influence the operation of an application. For instance, it’s easy to think of the values in a `.ini` file parsed at application startup time as “configuration”. However, if you’re reasonably open-minded, it’s easy to think of *code* as configuration too. Since Pyramid, like most other web application platforms, is a *framework*, it calls into code that you write (as opposed to a *library*, which is code that exists purely for you to call). The act of plugging application code that you’ve written into Pyramid is also referred to within this documentation as “configuration”; you are configuring Pyramid to call the code that makes up your application.

There are two ways to configure a Pyramid application: *imperative configuration* and *declarative configuration*. Both are described below.

4.1 Imperative Configuration

“Imperative configuration” just means configuration done by Python statements, one after the next. Here’s one of the simplest Pyramid applications, configured imperatively:

```
1 from wsgiref.simple_server import make_server
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()
```

4. APPLICATION CONFIGURATION

```
10 config.add_view(hello_world)
11 app = config.make_wsgi_app()
12 server = make_server('0.0.0.0', 8080, app)
13 server.serve_forever()
```

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.

4.2 Declarative Configuration

It's sometimes painful to have all configuration done by 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:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4 from pyramid.view import view_config
5
6 @view_config()
7 def hello(request):
8     return Response('Hello')
9
10 if __name__ == '__main__':
11     config = Configurator()
12     config.scan()
13     app = config.make_wsgi_app()
14     server = make_server('0.0.0.0', 8080, app)
15     server.serve_forever()
```

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.

The combination of *configuration decoration* and the invocation of a *scan* is collectively known as *declarative configuration*.

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

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

4.3 Summary

There are two ways to configure a Pyramid application: declaratively and imperatively. You can choose the mode you're most comfortable with; both are completely equivalent. Examples in this documentation will use both modes interchangeably.

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 *scaffold* to generate a basic Pyramid *project*.

A project is a directory that contains at least one Python *package*. You'll use a scaffold 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: 1) a package is more easily extended with new code and 2) 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 scaffolds that you can use to generate a project. Each scaffold makes different configuration assumptions about what type of application you're trying to construct.

These scaffolds are rendered using the `pcreate` command that is installed as part of Pyramid.

5.1 Scaffolds Included with Pyramid

The convenience scaffolds 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*).

The included scaffolds are these:

starter URL mapping via *URL dispatch* and no persistence mechanism.

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

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

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 `pcreate` command installed within the `virtualenv`. We'll choose the starter scaffold for this purpose. When we invoke `pcreate`, it will create a directory that represents our project.

In *Installing Pyramid* we called the `virtualenv` directory `env`; the following commands assume that our current working directory is the `env` directory.

The below example uses the `pcreate` command to create a project with the `starter` scaffold.

On UNIX:

```
$ $VENV/bin/pcreate -s starter MyProject
```

Or on Windows:

```
> %VENV%\Scripts\pcreate -s starter MyProject
```

Here's sample output from a run of `pcreate` on UNIX for a project we name `MyProject`:

```
$ $VENV/bin/pcreate -s starter MyProject
Creating template pyramid
Creating directory ./MyProject
# ... more output ...
Running /Users/chrism/projects/pyramid/bin/python setup.py egg_info
```

As a result of invoking the `pcreate` command, a directory named `MyProject` is created. 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 `.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 debug your application. It contains configuration that enables an interactive debugger and settings optimized for development.

Another `.ini` file named `production.ini` will also be created in the project directory. It contains 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.

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.

We created this project within an `env` virtualenv directory. However, note that this is not mandatory. The project directory can go more or less anywhere on your filesystem. You don't need to put it in a special "web server" directory, and you don't need to put it within a virtualenv directory. The author uses Linux mainly, and tends to put project directories which he creates within his `~/projects` directory. On Windows, it's a good idea to put project directories within a directory that contains no space characters, so it's wise to *avoid* a path that contains i.e. `My Documents`. As a result, the author, when he uses Windows, just puts his projects in `C:\projects`.



You'll need to avoid using `pcreate` to create a project with the same name as a Python standard library component. In particular, this means you should avoid using the names `site` or `test`, both of which conflict with Python standard library packages. You should also avoid using the name `pyramid`, which will conflict with Pyramid itself.

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 `pcreate`-generated project directory. The `python` you're invoking should be the one that lives in the `bin` (or `Scripts` on Windows) directory of your virtual Python environment. Your terminal's current working directory *must* be the newly created project directory.

On UNIX:

```
$ cd MyProject
$ $VENV/bin/python setup.py develop
```

Or on Windows:

```
> cd MyProject
> %VENV%\Scripts\python.exe setup.py develop
```

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

5. CREATING A PYRAMID PROJECT

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

This will install a *distribution* representing your project into the virtual environment interpreter's library set so it can be found by import statements and by other console scripts such as pserve, pshell, proutes and pviews.

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*).

On UNIX:

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

Or on Windows:


```
> %VENV%\Scripts\python.exe setup.py test -q
```

Here's sample output from a test run on UNIX:

```
$ $VENV/bin/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
```

The tests themselves are found in the `tests.py` module in your `pcreate` generated project. Within a project generated by the `starter` scaffold, a single sample test exists.

 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).

5.5 Running The Project Application

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

On UNIX:

```
$ $VENV/bin/pserve development.ini
```

On Windows:

```
> %VENV%\Scripts\pserve development.ini
```

Here's sample output from a run of `pserve` on UNIX:

```
$ $VENV/bin/pserve development.ini
Starting server in PID 16601.
serving on http://0.0.0.0:6543
```

When you use `pserve` to start the application implied by the default rendering of a scaffold, it will respond to requests on *all* IP addresses possessed by your system, not just requests to `localhost`. This is what the `0.0.0.0` in `serving on http://0.0.0.0:6543` means. The server will respond to requests made to `127.0.0.1` and on any external IP address. For example, your system might be configured to have an external IP address `192.168.1.50`. If that's the case, if you use a browser running on the same system as Pyramid, it will be able to access the application via `http://127.0.0.1:6543/` as well as via `http://192.168.1.50:6543/`. However, *other people* on other computers on the same network will also be able to visit your Pyramid application in their browser by visiting `http://192.168.1.50:6543/`.

If you want to restrict access such that only a browser running on the same machine as Pyramid will be able to access your Pyramid application, edit the `development.ini` file, and replace the `host` value in the `[server:main]` section. Change it from `0.0.0.0` to `127.0.0.1`. For example:

5. CREATING A PYRAMID PROJECT

```
[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 6543
```

You can change the port on which the server runs on by changing the same portion of the `development.ini` file. For example, you can change the `port = 6543` line in the `development.ini` file's `[server:main]` section to `port = 8080` to run the server on port 8080 instead of port 6543.

You can shut down a server started this way by pressing `Ctrl-C`.

The default server used to run your Pyramid application when a project is created from a scaffold is named *Waitress*. This server is what prints the `serving on...` line when you run `pserve`. It's a good idea to use this server during development, because it's very simple. It can also be used for light production. Setting your application up under a different server is not advised until you've done some development work under the default server, particularly if you're not yet experienced with Python web development. Python web server setup can be complex, and you should get some confidence that your application works in a default environment before trying to optimize it or make it "more like production". It's awfully easy to get sidetracked trying to set up a nondefault server for hours without actually starting to do any development. One of the nice things about Python web servers is that they're largely interchangeable, so if your application works under the default server, it will almost certainly work under any other server in production if you eventually choose to use a different one. Don't worry about it right now.

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*.

5.5.1 Reloading Code

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

```
$ $VENV/bin/pserve development.ini --reload
Starting subprocess with file monitor
Starting server in PID 16601.
serving on http://0.0.0.0:6543
```

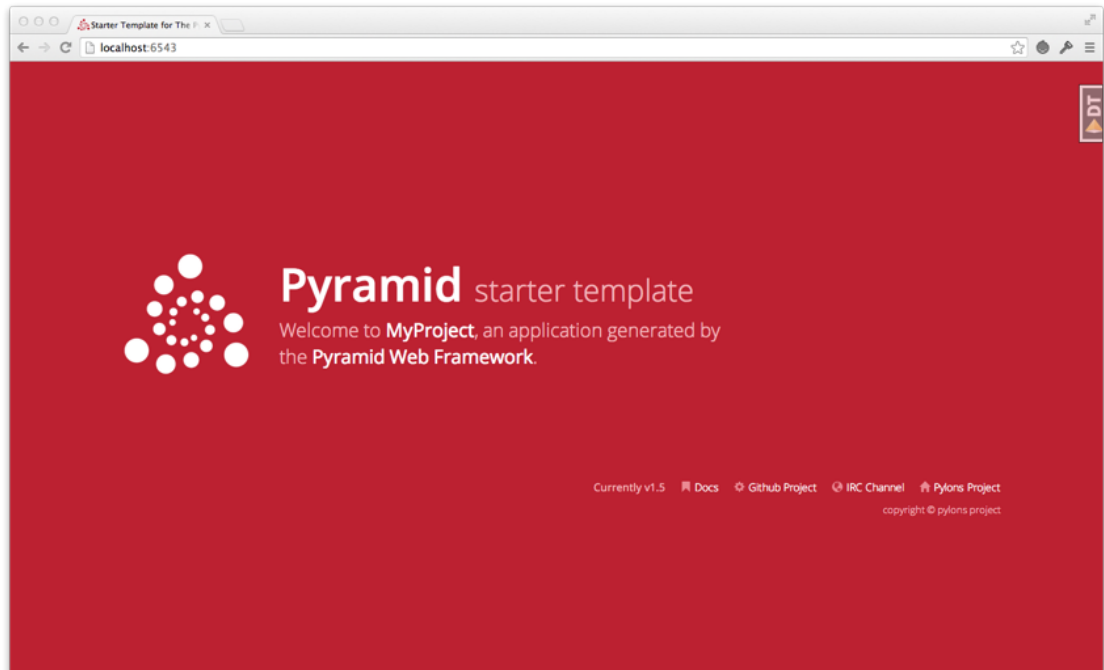
Now if you make a change to any of your project's `.py` files or `.ini` files, you'll see the server restart automatically:

```
development.ini changed; reloading...
----- Restarting -----
Starting server in PID 16602.
serving on http://0.0.0.0:6543
```

Changes to template files (such as `.pt` or `.mak` files) won't cause the server to restart. Changes to template files don't require a server restart as long as the `pyramid.reload_templates` setting in the `development.ini` file is `true`. Changes made to template files when this setting is `true` will take effect immediately without a server restart.

5.6 Viewing the Application

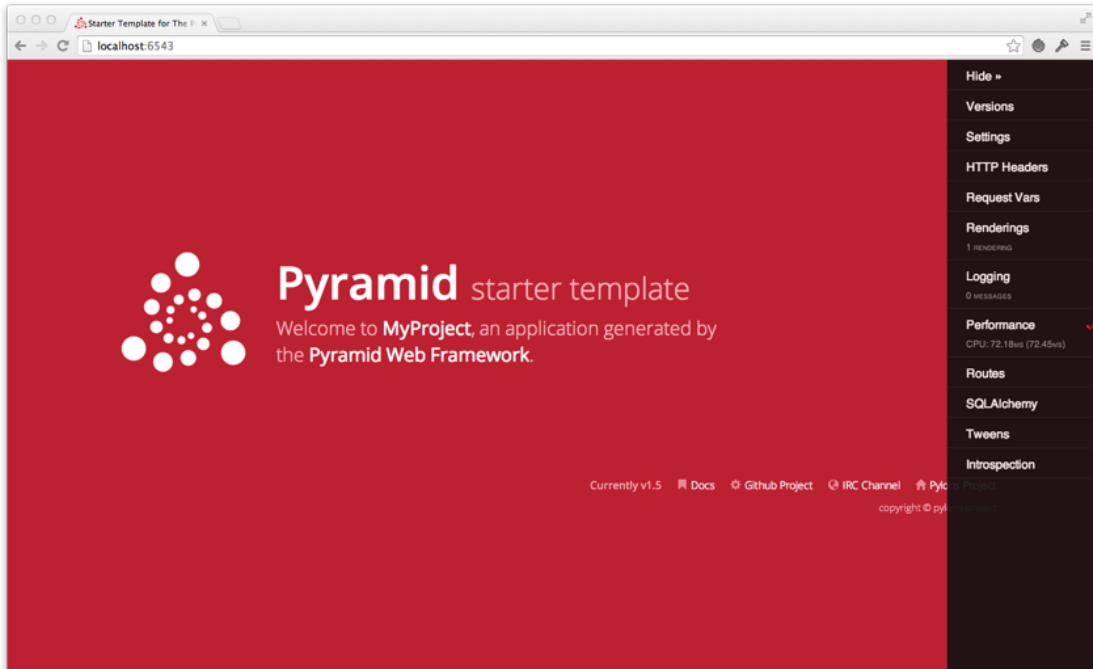
Once your application is running via `pserve`, you may visit `http://localhost:6543/` in your browser. You will see something in your browser like what is displayed in the following image:



This is the page shown by default when you visit an unmodified `pcreate` generated starter application in a browser.

5.6.1 The Debug Toolbar

If you click on the image shown at the right hand top of the page (“^DT”), you’ll be presented with a debug toolbar that provides various niceties while you’re developing. This image will float above every HTML page served by Pyramid while you develop an application, and allows you show the toolbar as necessary. Click on Hide to hide the toolbar and show the image again.



If you don’t see the debug toolbar image on the right hand top of the page, it means you’re browsing from a system that does not have debugging access. By default, for security reasons, only a browser originating from localhost (127.0.0.1) can see the debug toolbar. To allow your browser on a remote system to access the server, add a line within the `[app:main]` section of the `development.ini` file in the form `debugtoolbar.hosts = X.X.X.X`. For example, if your Pyramid application is running on a remote system, and you’re browsing from a host with the IP address 192.168.1.1, you’d add something like this to enable the toolbar when your system contacts Pyramid:

```
[app:main]
# .. other settings ...
debugtoolbar.hosts = 192.168.1.1
```

For more information about what the debug toolbar allows you to do, see the documentation for `pyramid_debugtoolbar`.

The debug toolbar will not be shown (and all debugging will be turned off) when you use the `production.ini` file instead of the `development.ini` file to run the application.

You can also turn the debug toolbar off by editing `development.ini` and commenting out a line. For example, instead of:

```
1 [app:main]
2 ...
3 pyramid.includes =
4     pyramid_debugtoolbar
```

Put a hash mark at the beginning of the `pyramid_debugtoolbar` line:

```
1 [app:main]
2 ...
3 pyramid.includes =
4 #     pyramid_debugtoolbar
```

Then restart the application to see that the toolbar has been turned off.

Note that if you comment out the `pyramid_debugtoolbar` line, the `#` *must* be in the first column. If you put it anywhere else, and then attempt to restart the application, you'll receive an error that ends something like this:

```
ImportError: No module named #pyramid_debugtoolbar
```

5.7 The Project Structure

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

All Pyramid `pcreate`-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
|   |-- static
|   |   |-- pyramid-16x16.png
|   |   |-- pyramid.png
|   |   |-- theme.css
|   |   '-- theme.min.css
|   |-- templates
|   |   '-- mytemplate.pt
|   |-- tests.py
|   '-- views.py
|-- production.ini
|-- README.txt
'-- setup.py
```

5.8 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. `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.
6. `setup.py` is the file you’ll use to test and distribute your application. It is a standard *setuptools* `setup.py` file.

5.8.1 development.ini

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

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

```

1  ###
2  # app configuration
3  # http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/environment.html
4  ###
5
6  [app:main]
7  use = egg:MyProject
8
9  pyramid.reload_templates = true
10 pyramid.debug_authorization = false
11 pyramid.debug_notfound = false
12 pyramid.debug_routematch = false
13 pyramid.default_locale_name = en
14 pyramid.includes =
15     pyramid_debugtoolbar
16
17 # By default, the toolbar only appears for clients from IP addresses
18 # '127.0.0.1' and '::1'.
19 # debugtoolbar.hosts = 127.0.0.1 ::1
20
21 ###
22 # wsgi server configuration
23 ###
24
25 [server:main]
26 use = egg:waitress#main
27 host = 0.0.0.0
28 port = 6543
29
30 ###
31 # logging configuration
32 # http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/logging.html
33 ###
34
35 [loggers]
36 keys = root, myproject
37
38 [handlers]
39 keys = console

```

```
40
41 [formatters]
42 keys = generic
43
44 [logger_root]
45 level = INFO
46 handlers = console
47
48 [logger_myproject]
49 level = DEBUG
50 handlers =
51 qualname = myproject
52
53 [handler_console]
54 class = StreamHandler
55 args = (sys.stderr,)
56 level = NOTSET
57 formatter = generic
58
59 [formatter_generic]
60 format = %(asctime)s %(levelname)-5.5s [% (name)s] [% (threadName)s] %(message)s
```

This file contains several sections including `[app:main]`, `[server:main]` and several other sections related to logging configuration.

The `[app:main]` section represents configuration for your Pyramid application. The `use` setting is the only setting required to be present in the `[app:main]` section. Its default value, `egg:MyProject`, indicates that our `MyProject` project contains the application that should be served. Other settings added to this section are passed as keyword arguments to the function named `main` in our package's `__init__.py` module. You can provide startup-time configuration parameters to your application by adding more settings to this section.



See *Entry Points and PasteDeploy .ini Files* for more information about the meaning of the `use = egg:MyProject` value in this section.

The `pyramid.reload_templates` setting in the `[app:main]` section is a Pyramid -specific setting which is passed into the framework. If it exists, and its value is `true`, supported template changes will not require an application restart to be detected. See *Automatically Reloading Templates* for more information.



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

The `pyramid.includes` setting in the `[app:main]` section tells Pyramid to “include” configuration from another package. In this case, the line `pyramid.includes = pyramid_debugtoolbar` tells Pyramid to include configuration from the `pyramid_debugtoolbar` package. This turns on a debugging panel in development mode which will be shown on the right hand side of the screen. Including the debug toolbar will also make it possible to interactively debug exceptions when an error occurs.

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.

The name `main` in `[app:main]` signifies that this is the default application run by `pserve` 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). This means that any remote system which has TCP access to your system can see your Pyramid application. The sections that live between the markers `# Begin logging configuration` and `# End logging configuration` represent Python’s standard library logging module configuration for your application. The sections between these two markers are passed to the logging module’s config file configuration engine when the `pserve` or `pshell` commands are executed. The default configuration sends application logging output to the standard error output of your terminal. For more information about logging configuration, see *Logging*.

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.

5.8.2 `production.ini`

The `production.ini` file is a *PasteDeploy* configuration file with a purpose much like that of `development.ini`. However, it disables the debug toolbar, and filters all log messages except those above the `WARN` level. It also turns off template development options such that templates are not automatically reloaded when changed, and turns off all debugging options. This file is appropriate to use instead of `development.ini` when you put your application into production.

It’s important to use `production.ini` (and *not* `development.ini`) to benchmark your application and put it into production. `development.ini` configures your system with a debug toolbar that helps development, but the inclusion of this toolbar slows down page rendering times by over an order of magnitude. The debug toolbar is also a potential security risk if you have it configured incorrectly.

5.8.3 MANIFEST.in

The `MANIFEST.in` file is a *distutils* configuration file which specifies the non-Python files that should be included when a *distribution* of your Pyramid project is created when you run `python setup.py sdist`. Due to the information contained in the default `MANIFEST.in`, an *sdist* of your Pyramid project will include `.txt` files, `.ini` files, `.rst` files, graphics files, and template files, as well as `.py` files. See <http://docs.python.org/distutils/sourcedist.html#the-manifest-in-template> for more information about the syntax and usage of `MANIFEST.in`.

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 scaffolds 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.

5.8.4 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, packaging, and distributing your application.



`setup.py` is the de facto 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 and *The Hitchhiker's Guide to Packaging*.

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  with open(os.path.join(here, 'README.txt')) as f:
7      README = f.read()
8  with open(os.path.join(here, 'CHANGES.txt')) as f:
9      CHANGES = f.read()
10
11  requires = [
12      'pyramid',
13      'pyramid_chameleon',
14      'pyramid_debugtoolbar',
15      'waitress',
16  ]
17
18  setup(name='MyProject',
19        version='0.0',
20        description='MyProject',
21        long_description=README + '\n\n' + CHANGES,
22        classifiers=[
23            "Programming Language :: Python",
24            "Framework :: Pyramid",
25            "Topic :: Internet :: WWW/HTTP",
26            "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
27        ],
28        author='',
29        author_email='',
30        url='',
31        keywords='web pyramid pylons',
32        packages=find_packages(),
33        include_package_data=True,
34        zip_safe=False,
35        install_requires=requires,
36        tests_require=requires,
37        test_suite="myproject",
38        entry_points="""\
39    [paste.app_factory]
40    main = myproject:main
41    """,
42    )

```

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, when adding Python package dependencies, 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 install and use your application.

5.9 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 entry point for commands such as `pserve`, `pshell`, `pviews`, and others.
2. A `templates` directory, which contains *Chameleon* (or other types of) templates.
3. A `tests.py` module, which contains unit test code for the application.
4. A `views.py` module, which contains view code for the application.

These are purely conventions established by the scaffold: 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.

5.9.1 `__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
3
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6     """
7     config = Configurator(settings=settings)
8     config.include('pyramid_chameleon')
9     config.add_static_view('static', 'static', cache_max_age=3600)
10    config.add_route('home', '/')
11    config.scan()
12    return config.make_wsgi_app()
```

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

Within this function, application configuration is performed.

Line 7 creates an instance of a *Configurator*.

Line 8 adds support for Chameleon templating bindings, allowing us to specify renderers with the `.pt` extension.

Line 9 registers a static view, which will serve up the files from the `myproject:static` *asset specification* (the `static` directory of the `myproject` package).

Line 10 adds a *route* to the configuration. This route is later used by a view in the `views` module.

Line 11 calls `config.scan()`, which picks up view registrations declared elsewhere in the package (in this case, in the `views.py` module).

Line 12 returns a *WSGI* application to the caller of the function (Pyramid's `pserve`).

5.9.2 `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*.

5. CREATING A PYRAMID PROJECT

```
1 from pyramid.view import view_config
2
3
4 @view_config(route_name='home', renderer='templates/mytemplate.pt')
5 def my_view(request):
6     return {'project': 'MyProject'}
```

Lines 4-6 define and register a *view callable* named `my_view`. The function named `my_view` is decorated with a `view_config` decorator (which is processed by the `config.scan()` line in our `__init__.py`). The `view_config` decorator asserts that this view be found when a *route* named `home` is matched. In our case, because our `__init__.py` maps the route named `home` to the URL pattern `/`, this route will match when a visitor visits the root URL. The `view_config` decorator 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 `templates/mytemplate.pt`, which is a *asset specification* that specifies the `mytemplate.pt` file within the `templates` directory of the `myproject` package. The asset specification could have also been specified as `myproject:templates/mytemplate.pt`; the leading package name and colon is optional. The template file pointed to is a *Chameleon* ZPT template file (`templates/my_template.pt`).

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 is configured to invoke a *renderer* on a template. The dictionary the view returns (on line 6) provides the value the renderer substitutes into the template when generating HTML. The renderer then returns the HTML in a *response*.



Dictionaries provide values to *templates*.



When the application is run with the scaffold's *default development.ini* configuration *logging is set up* to aid debugging. If an exception is raised, uncaught tracebacks are displayed after the startup messages on *the console running the server*. Also `print()` statements may be inserted into the application for debugging to send output to this console.



`development.ini` has a setting that controls how templates are reloaded, `pyramid.reload_templates`.

- When set to `True` (as in the scaffold `development.ini`) changed templates automatically reload without a server restart. This is convenient while developing, but slows template rendering speed.
- When set to `False` (the default value), changing templates requires a server restart to reload them. Production applications should use `pyramid.reload_templates = False`.

See also:

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

See also:

Pyramid can also dynamically reload changed Python files. See also *Reloading Code*.

See also:

See also the *The Debug Toolbar*, which provides interactive access to your application's internals and, should an exception occur, allows interactive access to traceback execution stack frames from the Python interpreter.

5.9.3 static

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

5.9.4 templates/mytemplate.pt

The single *Chameleon* template that 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 `@view_config` as the `renderer` of the `my_view` view callable in the `views.py` 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*.

5.9.5 tests.py

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

```
1 import unittest
2
3 from pyramid import testing
4
5
6 class ViewTests(unittest.TestCase):
7     def setUp(self):
8         self.config = testing.setUp()
9
10    def tearDown(self):
11        testing.tearDown()
12
13    def test_my_view(self):
14        from .views import my_view
15        request = testing.DummyRequest()
16        info = my_view(request)
17        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.

5.10 Modifying Package Structure

It is best practice for your application's code layout to not stray too much from accepted Pyramid scaffold 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 scaffold are in no way magical or required. Despite the choices made for you by any scaffold, 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 starter scaffold 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 `myproject` package directory (the same directory which holds `views.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*.)
- *Move* the content from the existing `views.py` file to a file inside the new `views` directory named, say, `blog.py`. Because the `templates` directory remains in the `myproject` package, the template *asset specification* values in `blog.py` must now be fully qualified with the project's package name (`myproject:templates/blog.pt`).

You can then continue to add view callable functions to the `blog.py` module, but you can also add other `.py` files which contain view callable functions to the `views` directory. As long as you use the `@view_config` directive to register views in conjunction with `config.scan()` they will be picked up automatically when the application is restarted.

5.11 Using the Interactive Shell

It is possible to use the `pshell` command to load a Python interpreter prompt with a similar configuration as would be loaded if you were running your Pyramid application via `pserve`. This can be a useful debugging tool. See *The Interactive Shell* for more details.

5.12 What Is This `pserve` Thing

The code generated by an Pyramid scaffold assumes that you will be using the `pserve` command to start your application while you do development. `pserve` is a command that reads a *PasteDeploy* `.ini` file (e.g. `development.ini`) and configures a server to serve a Pyramid application based on the data in the file.

`pserve` is by no means the only way to start up and serve a Pyramid application. As we saw in *Creating Your First Pyramid Application*, `pserve` needn't be invoked at all to run a Pyramid application. The use of `pserve` to run a Pyramid application is purely conventional based on the output of its scaffolding. But we strongly recommend using `pserve` while developing your application, because many other convenience introspection commands (such as `pviews`, `prequest`, `proutes` and others) are also implemented in terms of configuration availability of this `.ini` file format. It also configures Pyramid logging and provides the `--reload` switch for convenient restarting of the server when code changes.

5.13 Using an Alternate WSGI Server

Pyramid scaffolds generate projects which use the *Waitress* WSGI server. Waitress is a server that is suited for development and light production usage. It's not the fastest nor the most featureful WSGI server. Instead, its main feature is that it works on all platforms that Pyramid needs to run on, making it a good choice as a default server from the perspective of Pyramid's developers.

Any WSGI server is capable of running a Pyramid application. But we suggest you stick with the default server for development, and that you wait to investigate other server options until you're ready to deploy your application to production. Unless for some reason you need to develop on a non-local system, investigating alternate server options is usually a distraction until you're ready to deploy. But we recommend developing using the default configuration on a local system that you have complete control over; it will provide the best development experience.

One popular production alternative to the default Waitress server is *mod_wsgi*. You can use *mod_wsgi* to serve your Pyramid application using the Apache web server rather than any "pure-Python" server like Waitress. It is fast and featureful. See *Running a Pyramid Application under mod_wsgi* for details.

Another good production alternative is *Green Unicorn* (aka *gunicorn*). It's faster than Waitress and slightly easier to configure than *mod_wsgi*, although it depends, in its default configuration, on having a buffering HTTP proxy in front of it. It does not, as of this writing, work on Windows.

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:

```
$ pserve development.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 `pserve development.ini` and the time the line `serving on 0.0.0.0:6543 ...` is output to your console.

6.1 The Startup Process

The easiest and best-documented way to start and serve a Pyramid application is to use the `pserve` 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 `pserve development.ini`.

1. The `pserve` command is invoked under your shell with the argument `development.ini`. As a result, Pyramid recognizes that it is meant to begin to run and serve an application using the information contained within the `development.ini` file.

2. The 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's `paste.app_factory` *entry point* 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 "composite" (e.g. `[composite:main]`), refer to the documentation for that particular composite to understand how to make it refer to your Pyramid application. In most cases, a Pyramid application built from a scaffold will have a single `[app:main]` section in it, and this will be the application served.
3. The framework finds all logging related configuration in the `.ini` file and uses it to configure the Python standard library logging system for this application. See *Logging Configuration* for more information.
4. The application's *constructor* named by the entry point reference 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
3
4 def main(global_config, **settings):
5     """ This function returns a Pyramid WSGI application.
6         """
7     config = Configurator(settings=settings)
8     config.include('pyramid_chameleon')
9     config.add_static_view('static', 'static', cache_max_age=3600)
10    config.add_route('home', '/')
11    config.scan()
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 (if `[DEFAULT]` is present). 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:main]` section (except for the `use=` setting) when this function is called by when you run `pserve`.

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

```
1  ###
2  # app configuration
3  # http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/environment.html
4  ###
5
6  [app:main]
7  use = egg:MyProject
8
9  pyramid.reload_templates = true
10 pyramid.debug_authorization = false
11 pyramid.debug_notfound = false
12 pyramid.debug_routematch = false
13 pyramid.default_locale_name = en
14 pyramid.includes =
15     pyramid_debugtoolbar
16
17 # By default, the toolbar only appears for clients from IP addresses
18 # '127.0.0.1' and '::1'.
19 # debugtoolbar.hosts = 127.0.0.1 ::1
20
21 ###
22 # wsgi server configuration
23 ###
24
25 [server:main]
26 use = egg:waitress#main
27 host = 0.0.0.0
28 port = 6543
29
30 ###
31 # logging configuration
32 # http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/logging.html
33 ###
34
35 [loggers]
36 keys = root, myproject
37
38 [handlers]
39 keys = console
40
41 [formatters]
42 keys = generic
43
44 [logger_root]
45 level = INFO
46 handlers = console
```

```
47 [logger_myproject]
48 level = DEBUG
49 handlers =
50 qualname = myproject
51
52 [handler_console]
53 class = StreamHandler
54 args = (sys.stderr,)
55 level = NOTSET
56 formatter = generic
57
58 [formatter_generic]
59
60 format = %(asctime)s %(levelname)-5.5s %(name)s [%s] %(message)s
```

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 `{'pyramid.reload_templates': 'true', 'pyramid.debug_authorization': 'false', 'pyramid.debug_notfound': 'false', 'pyramid.debug_routematch': 'false', 'pyramid.debug_templates': 'true', 'pyramid.default_locale_name': 'en'}`. See *Environment Variables and .ini File Settings* for the meanings of these keys.

5. The main function first constructs a `Configurator` instance, passing the settings dictionary captured via the `**settings` kwarg as its settings argument.

The settings dictionary contains all the options in the `[app:main]` section of our `.ini` file except the use option (which is internal to PasteDeploy) such as `pyramid.reload_templates`, `pyramid.debug_authorization`, etc.

6. The main function then calls various methods on the instance of the class `Configurator` created in the previous step. The intent of calling these methods is to populate an *application registry*, which represents the Pyramid configuration related to the application.
7. 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.
8. An `ApplicationCreated` event is emitted (see *Using Events* for more information about events).
9. Assuming there were no errors, the main function in `myproject` returns the router instance created by `pyramid.config.Configurator.make_wsgi_app()` back to `pserve`. As far as `pserve` is concerned, it is “just another WSGI application”.

10. `pserve` starts the WSGI *server* defined within the `[server:main]` section. In our case, this is the Waitress server (`use = egg:waitress#main`), 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.

6.2 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.

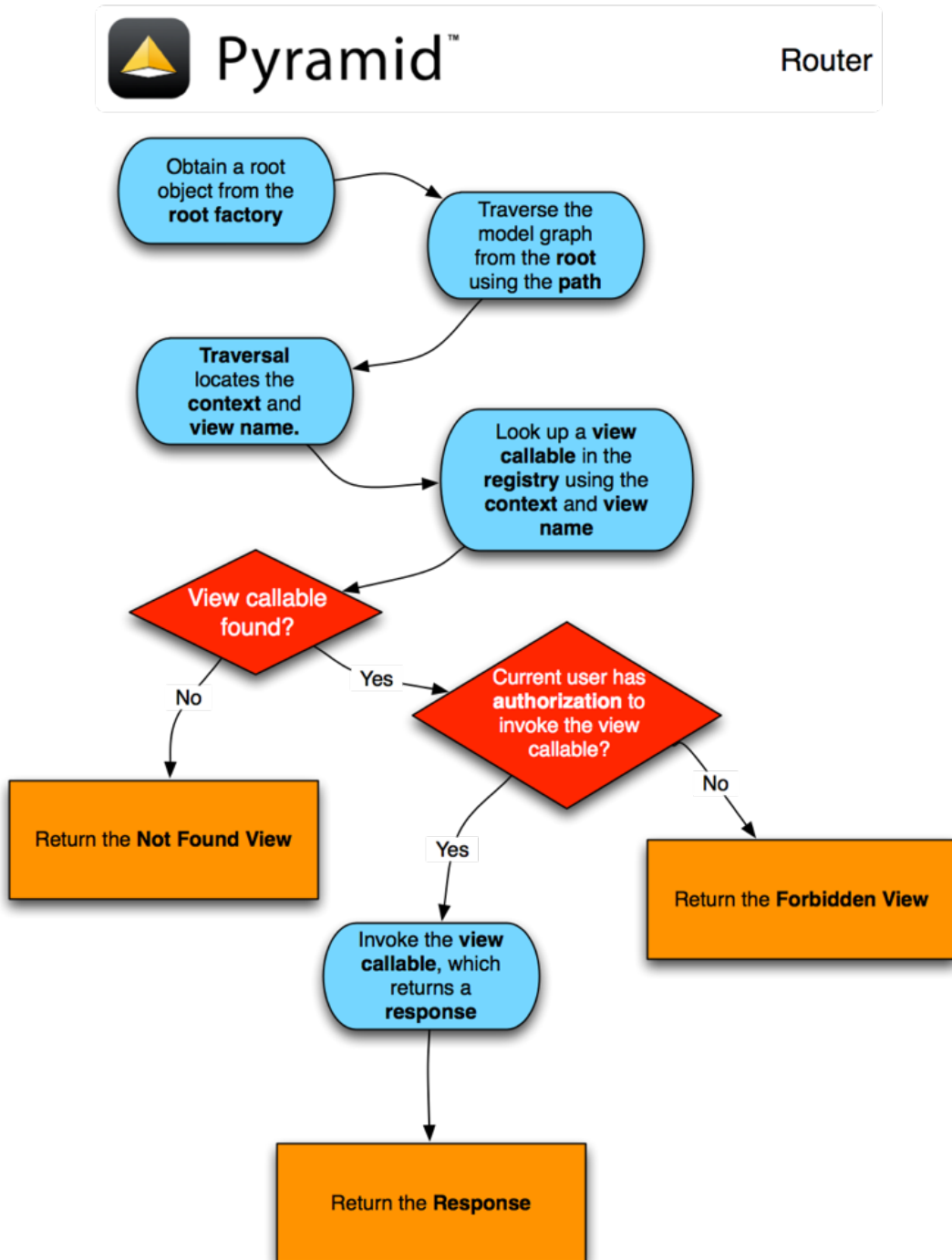
Request Processing

Once a Pyramid application is up and running, it is ready to accept requests and return responses. What happens from the time a *WSGI* request enters a Pyramid application through to the point that Pyramid hands off a response back to WSGI for upstream processing?

1. A user initiates a request from his browser to the hostname and port number of the WSGI server used by the Pyramid application.
2. The WSGI server used by the Pyramid application passes the WSGI environment to the `__call__` method of the Pyramid *router* object.
3. A *request* object is created based on the WSGI environment.
4. The *application registry* and the *request* object created in the last step are pushed on to the *thread local* stack that Pyramid uses to allow the functions named `get_current_request()` and `get_current_registry()` to work.
5. A `NewRequest` *event* is sent to any subscribers.
6. If any *route* has been defined within application configuration, the Pyramid *router* calls a *URL dispatch* “route mapper.” The job of the mapper is to examine the request to determine whether any user-defined *route* matches the current WSGI environment. The *router* passes the request as an argument to the mapper.
7. If any route matches, the route mapper adds attributes to the request: `matchdict` and `matched_route` attributes are added to the request object. The former contains a dictionary representing the matched dynamic elements of the request’s `PATH_INFO` value, the latter contains the `IRoute` object representing the route which matched. The root object associated with the route found is also generated: if the *route configuration* which matched has an associated *factory* argument, this factory is used to generate the root object, otherwise a default *root factory* is used.

8. If a route match was *not* found, and a `root_factory` argument was passed to the *Configurator* constructor, that callable is used to generate the root object. If the `root_factory` argument passed to the *Configurator* constructor was `None`, a default root factory is used to generate a root object.
9. The Pyramid router calls a “traverser” function with the root object and the request. The traverser function attempts to traverse the root object (using any existing `__getitem__` on the root object and subobjects) to find a *context*. If the root object has no `__getitem__` method, the root itself is assumed to be the context. The exact traversal algorithm is described in *Traversal*. The traverser function returns a dictionary, which contains a *context* and a *view name* as well as other ancillary information.
10. The request is decorated with various names returned from the traverser (such as `context`, `view_name`, and so forth), so they can be accessed via e.g. `request.context` within *view* code.
11. A `ContextFound` *event* is sent to any subscribers.
12. Pyramid looks up a *view* callable using the context, the request, and the view name. If a view callable doesn’t exist for this combination of objects (based on the type of the context, the type of the request, and the value of the view name, and any *predicate* attributes applied to the view configuration), Pyramid raises a `HTTPNotFound` exception, which is meant to be caught by a surrounding *exception view*.
13. If a view callable was found, Pyramid attempts to call it. If an *authorization policy* is in use, and the view configuration is protected by a *permission*, Pyramid determines whether the view callable being asked for can be executed by the requesting user based on credential information in the request and security information attached to the context. If the view execution is allowed, Pyramid calls the view callable to obtain a response. If view execution is forbidden, Pyramid raises a `HTTPForbidden` exception.
14. If any exception is raised within a *root factory*, by *traversal*, by a *view callable* or by Pyramid itself (such as when it raises `HTTPNotFound` or `HTTPForbidden`), the router catches the exception, and attaches it to the request as the `exception` attribute. It then attempts to find a *exception view* for the exception that was caught. If it finds an exception view callable, that callable is called, and is presumed to generate a response. If an *exception view* that matches the exception cannot be found, the exception is reraised.
15. The following steps occur only when a *response* could be successfully generated by a normal *view callable* or an *exception view* callable. Pyramid will attempt to execute any *response callback* functions attached via `add_response_callback()`. A `NewResponse` *event* is then sent to any subscribers. The response object’s `__call__` method is then used to generate a WSGI response. The response is sent back to the upstream WSGI server.

-
16. Pyramid will attempt to execute any *finished callback* functions attached via `add_finished_callback()`.
 17. The *thread local* stack is popped.



This is a very high-level overview that leaves out various details. For more detail about subsystems invoked by the Pyramid router such as traversal, URL dispatch, views, and event processing, see *URL Dispatch*, *Views*, and *Using Events*.

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. A view callable is a specific bit of code, defined in your application, that receives the *request* and returns a *response* object.

8.1 High-Level Operational Overview

If any 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 the *view lookup* process to find a matching view.

If no route pattern in the route map matches the information in the *request* provided in your application, Pyramid will fail over to using *traversal* to perform resource location and view lookup.

8.2 Route Configuration

Route configuration is the act of adding a new *route* to an application. A route has a *name*, which acts as an identifier to be used for URL generation. The name also allows developers to associate a view configuration with the route. A route also has 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`). It also optionally has a *factory* and a set of *route predicate* attributes.

8.2.1 Configuring a Route to Match a View

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}')
config.add_view(myview, route_name='myroute')
```

When a *view callable* added to the configuration by way of `add_view()` becomes associated with a route via its `route_name` predicate, that view callable will always be found and invoked when the associated route pattern matches during a request.

More commonly, you will not use any `add_view` statements in your project's "setup" code. You will instead use `add_route` statements, and use a *scan* to associate view callables with routes. For example, if this is a portion of your project's `__init__.py`:

```
config.add_route('myroute', '/prefix/{one}/{two}')
config.scan('mypackage')
```

Note that we don't call `add_view()` in this setup code. However, the above *scan* execution `config.scan('mypackage')` will pick up each *configuration decoration*, including any objects decorated with the `pyramid.view.view_config` decorator in the `mypackage` Python package. For example, if you have a `views.py` in your package, a scan will pick up any of its configuration decorators, so we can add one there that references `myroute` as a `route_name` parameter:

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

@view_config(route_name='myroute')
def myview(request):
    return Response('OK')
```

The above combination of `add_route` and `scan` is completely equivalent to using the previous combination of `add_route` and `add_view`.

8.2.2 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
```

If a pattern is a valid URL it won't be ever matched against an incoming request. Instead it can be useful for generating external URLs. See *External routes* for details.

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 replacement marker in a pattern must begin with an uppercase or lowercase ASCII letter or an underscore, and can be composed only of uppercase or lowercase ASCII letters, underscores, and numbers. For example: `a`, `a_b`, `_b`, and `b9` are all valid replacement marker names, but `0a` is not.



A replacement marker could not start with an underscore until Pyramid 1.2. Previous versions required that the replacement marker start with an uppercase or lowercase letter.

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*:

8. 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:

```
http://example.com/foo/La%20Pe%C3%B1a
```

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

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

Literal strings in the path segment should represent the *decoded* value of the `PATH_INFO` provided to Pyramid. You don't want to use a URL-encoded value or a bytestring representing the literal's UTF-8 in the pattern. For example, rather than this:

```
/Foo%20Bar/{baz}
```

You'll want to use something like this:

```
/Foo Bar/{baz}
```

For patterns that contain “high-order” characters in its literals, you'll want to use a Unicode value as the pattern as opposed to any URL-encoded or UTF-8-encoded value. For example, you might be tempted to use a bytestring pattern like this:

```
/La Pe\xc3\xb1a/{x}
```

But this will either cause an error at startup time or it won't match properly. You'll want to use a Unicode value as the pattern instead rather than raw bytestring escapes. You can use a high-order Unicode value as the pattern by using Python source file encoding plus the “real” character in the Unicode pattern in the source, like so:

8. URL DISPATCH

```
/La Peña/{x}
```

Or you can ignore source file encoding and use equivalent Unicode escape characters in the pattern.

```
/La Pe\xfla/{x}
```

Dynamic segment names cannot contain high-order characters, so this applies only to literals in the pattern.

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:

```
/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':u''}
foo/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.

8.2.3 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 a different way of mapping URLs to code that Pyramid provides, named *traversal*, which does not depend on pattern ordering).

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}
members/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.

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

8.3 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. `PATH_INFO` represents the path portion of the URL that was requested.

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 request's `PATH_INFO` against the pattern declared. This checking happens in the order that the routes were declared via `pyramid.config.Configurator.add_route()`.

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

If any route matches, the route matching process stops and the *view lookup* subsystem takes over to find the most reasonable view callable for the matched route. Most often, there's only one view that will match (a view configured with a `route_name` argument matching the matched route). To gain a better understanding of how routes and views are associated in a real application, you can use the `pviews` command, as documented in *Displaying Matching Views for a Given URL*.

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

8.3.1 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`.

8.3.2 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`.

8.4 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.

8.4.1 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}')
2 config.add_view('mypackage.views.site_view', route_name='idea')
```

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.

8.4.2 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}')
2 config.add_route('user', 'users/{user}')
3 config.add_route('tag', 'tags/{tag}')
4
5 config.add_view('mypackage.views.idea_view', route_name='idea')
6 config.add_view('mypackage.views.user_view', route_name='user')
7 config.add_view('mypackage.views.tag_view', route_name='tag')
```

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.

8.4.3 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}', factory='myproject.resources.Idea')
2 config.add_view('myproject.views.idea_view', route_name='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.

See *Route Factories* for more details about how to use route factories.

8.5 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', '')
```

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

```
1 config.add_route('root', '/')
```

8.6 Generating Route URLs

Use the `pyramid.request.Request.route_url()` method 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.

```
url = request.route_url('foo', 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`).

To generate only the *path* portion of a URL from a route, use the `pyramid.request.Request.route_path()` API instead of `route_url()`.

```
url = request.route_path('foo', a='1', b='2', c='3')
```

This will return the string `/1/2/3` rather than a full URL.

Replacement values passed to `route_url` or `route_path` must be Unicode or bytestrings encoded in UTF-8. One exception to this rule exists: if you’re trying to replace a “remainder” match value (a `*stararg` replacement value), the value may be a tuple containing Unicode strings or UTF-8 strings.

Note that URLs and paths generated by `route_path` and `route_url` are always URL-quoted string types (they contain no non-ASCII characters). Therefore, if you’ve added a route like so:

```
config.add_route('la', u'/La Peña/{city}')
```

And you later generate a URL using `route_path` or `route_url` like so:

```
url = request.route_path('la', city=u'Québec')
```

You will wind up with the path encoded to UTF-8 and URL quoted like so:

```
/La%20Pe%C3%B1a/Qu%C3%A9bec
```

If you have a `*stararg` remainder dynamic part of your route pattern:

```
config.add_route('abc', 'a/b/c/*foo')
```

And you later generate a URL using `route_path` or `route_url` using a *string* as the replacement value:

```
url = request.route_path('abc', foo=u'Québec/biz')
```

The value you pass will be URL-quoted except for embedded slashes in the result:

```
/a/b/c/Qu%C3%A9bec/biz
```

You can get a similar result by passing a tuple composed of path elements:

```
url = request.route_path('abc', foo=(u'Québec', u'biz'))
```

Each value in the tuple will be url-quoted and joined by slashes in this case:

```
/a/b/c/Qu%C3%A9bec/biz
```

8.7 Static Routes

Routes may be added with a `static` keyword argument. For example:

```
1 config = Configurator()
2 config.add_route('page', '/page/{action}', static=True)
```

Routes added with a `True` `static` keyword argument will never be considered for matching at request time. Static routes are useful for URL generation purposes only. As a result, it is usually nonsensical to provide other non-name and non-pattern arguments to `add_route()` when `static` is passed as `True`, as none of the other arguments will ever be employed. A single exception to this rule is use of the `pregenerator` argument, which is not ignored when `static` is `True`.

External routes are implicitly static.

New in version 1.1: the `static` argument to `add_route()`

8.8 External Routes

New in version 1.5.

Route patterns that are valid URLs, are treated as external routes. Like *static routes* they are useful for URL generation purposes only and are never considered for matching at request time.

```
1 >>> config = Configurator()
2 >>> config.add_route('youtube', 'https://youtube.com/watch/{video_id}')
3 ...
4 >>> request.route_url('youtube', video_id='oHg5SJYRHA0')
5 >>> "https://youtube.com/watch/oHg5SJYRHA0"
```

Most pattern replacements and calls to `pyramid.request.Request.route_url()` will work as expected. However, calls to `pyramid.request.Request.route_path()` against external patterns will raise an exception, and passing `_app_url` to `route_url()` to generate a URL against a route that has an external pattern will also raise an exception.

8.9 Redirecting to Slash-Appended Routes

For behavior like Django's `APPEND_SLASH=True`, use the `append_slash` argument to `pyramid.config.Configurator.add_notfound_view()` or the equivalent `append_slash` argument to the `pyramid.view.notfound_view_config` decorator.

Adding `append_slash=True` 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. If the following routes are configured in your application:

```
1 from pyramid.httpexceptions import HTTPNotFound
2
3 def notfound(request):
4     return HTTPNotFound('Not found, bro.')
5
6 def no_slash(request):
7     return Response('No slash')
8
```

```

9 def has_slash(request):
10     return Response('Has slash')
11
12 def main(g, **settings):
13     config = Configurator()
14     config.add_route('noslash', 'no_slash')
15     config.add_route('haslash', 'has_slash/')
16     config.add_view(no_slash, route_name='noslash')
17     config.add_view(has_slash, route_name='haslash')
18     config.add_notfound_view(notfound, append_slash=True)

```

If a request enters the application with the `PATH_INFO` value of `/no_slash`, the first route will match and the browser will show “No slash”. 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. As a result, the `notfound` view will be called and it will return a “Not found, bro.” body.

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. As a result, the `notfound` view will never actually be called.

The following application uses the `pyramid.view.notfound_view_config` and `pyramid.view.view_config` decorators and a *scan* to do exactly the same job:

```

1 from pyramid.httpexceptions import HTTPNotFound
2 from pyramid.view import notfound_view_config, view_config
3
4 @notfound_view_config(append_slash=True)
5 def notfound(request):
6     return HTTPNotFound('Not found, bro.')
7
8 @view_config(route_name='noslash')
9 def no_slash(request):
10     return Response('No slash')
11
12 @view_config(route_name='haslash')
13 def has_slash(request):
14     return Response('Has slash')
15
16 def main(g, **settings):
17     config = Configurator()
18     config.add_route('noslash', 'no_slash')
19     config.add_route('haslash', 'has_slash/')
20     config.scan()

```



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.

See *pyramid.view* and *Changing the Not Found View* for a more general description of how to configure a view and/or a *Not Found View*.

8.10 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_ROUTEMATCH` environment variable or the `pyramid.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 $ PYRAMID_DEBUG_ROUTEMATCH=true $VENV/bin/pserve development.ini
2 Starting server in PID 13586.
3 serving on 0.0.0.0:6543 view at http://127.0.0.1:6543
4 2010-12-16 14:45:19,956 no route matched for url \
5                               http://localhost:6543/wontmatch
6 2010-12-16 14:45:20,010 no route matched for url \
7                               http://localhost:6543/favicon.ico
8 2010-12-16 14:41:52,084 route matched for url \
9                               http://localhost:6543/static/logo.png; \
10                              route_name: 'static/', ....
```

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

You can also use the `proutes` command to see a display of all the routes configured in your application; for more information, see *Displaying All Application Routes*.

8.11 Using a Route Prefix to Compose Applications

New in version 1.2.

The `pyramid.config.Configurator.include()` method allows configuration statements to be included from separate files. See *Rules for Building An Extensible Application* for information about this method. Using `pyramid.config.Configurator.include()` allows you to build your application from small and potentially reusable components.

The `pyramid.config.Configurator.include()` method accepts an argument named `route_prefix` which can be useful to authors of URL-dispatch-based applications. If `route_prefix` is supplied to the `include` method, it must be a string. This string represents a route prefix that will be prepended to all route patterns added by the *included* configuration. Any calls to `pyramid.config.Configurator.add_route()` within the included callable will have their pattern prefixed with the value of `route_prefix`. This can be used to help mount a set of routes at a different location than the included callable's author intended while still maintaining the same route names. For example:

```
1 from pyramid.config import Configurator
2
3 def users_include(config):
4     config.add_route('show_users', '/show')
5
6 def main(global_config, **settings):
7     config = Configurator()
8     config.include(users_include, route_prefix='/users')
```

In the above configuration, the `show_users` route will have an effective route pattern of `/users/show`, instead of `/show` because the `route_prefix` argument will be prepended to the pattern. The route will then only match if the URL path is `/users/show`, and when the `pyramid.request.Request.route_url()` function is called with the route name `show_users`, it will generate a URL with that same path.

Route prefixes are recursive, so if a callable executed via an `include` itself turns around and includes another callable, the second-level route prefix will be prepended with the first:

```
1 from pyramid.config import Configurator
2
3 def timing_include(config):
4     config.add_route('show_times', '/times')
5
6 def users_include(config):
7     config.add_route('show_users', '/show')
8     config.include(timing_include, route_prefix='/timing')
9
10 def main(global_config, **settings):
11     config = Configurator()
12     config.include(users_include, route_prefix='/users')
```

In the above configuration, the `show_users` route will still have an effective route pattern of `/users/show`. The `show_times` route however, will have an effective pattern of `/users/timing/times`.

Route prefixes have no impact on the requirement that the set of route *names* in any given Pyramid configuration must be entirely unique. If you compose your URL dispatch application out of many small subapplications using `pyramid.config.Configurator.include()`, it's wise to use a dotted name for your route names, so they'll be unlikely to conflict with other packages that may be added in the future. For example:

```
1 from pyramid.config import Configurator
2
3 def timing_include(config):
4     config.add_route('timing.show_times', '/times')
5
6 def users_include(config):
7     config.add_route('users.show_users', '/show')
8     config.include(timing_include, route_prefix='/timing')
9
10 def main(global_config, **settings):
11     config = Configurator()
12     config.include(users_include, route_prefix='/users')
```

8.12 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('route_to_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('ymd', '/{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. URL DISPATCH

```
8
9 ymd_to_int = integers('year', 'month', 'day')
10
11 config.add_route('ymd', '/{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 ('ymd', '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('ymd', '/{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 'ymd', 'ym', or 'y'.

You can also caption the predicates by setting the `__text__` attribute. This will help you with the `pviews` command (see *Displaying All Application Routes*) and the `pyramid_debugtoolbar`.

If a predicate is a class just add `__text__` property in a standard manner.

```

1 class DummyCustomPredicate1(object):
2     def __init__(self):
3         self.__text__ = 'my custom class predicate'
4
5 class DummyCustomPredicate2(object):
6     __text__ = 'my custom class predicate'

```

If a predicate is a method you'll need to assign it after method declaration (see PEP 232)

```

1 def custom_predicate():
2     pass
3 custom_predicate.__text__ = 'my custom method predicate'

```

If a predicate is a classmethod using `@classmethod` will not work, but you can still easily do it by wrapping it in classmethod call.

```

1 def classmethod_predicate():
2     pass
3 classmethod_predicate.__text__ = 'my classmethod predicate'
4 classmethod_predicate = classmethod(classmethod_predicate)

```

Same will work with `staticmethod`, just use `staticmethod` instead of `classmethod`.

See also:

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

8.13 Route Factories

Although it is not a particular common need in basic applications, 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',
2                 factory='myproject.resources.root_factory')
3 config.add_view('myproject.views.theview', route_name='abc')

```

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.

A factory must be a callable which accepts a request and returns an arbitrary Python object. For example, the below class can be used as a factory:

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

A route factory is actually conceptually identical to the *root factory* described at *The Resource Tree*.

Supplying a different resource factory for 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*.

8.14 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') ]
```

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.

8.15 Route View Callable Registration and Lookup Details

When a request enters the system which matches the pattern of the route, the usual 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 is associated with 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 an `add_view` statement mentions a `route_name` attribute, a *view configuration* is registered at startup time. This view configuration uses a 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.

As we can see from the above description, technically, URL dispatch doesn't actually map a URL pattern directly to a view callable. Instead, URL dispatch is a *resource location* mechanism. A Pyramid *resource location* subsystem (i.e., *URL dispatch* or *traversal*) finds a *resource* object that is the *context* of a *request*. Once the *context* is determined, a separate subsystem named *view lookup* is then responsible for finding and invoking a *view callable* based on information available in the context and the request. When URL dispatch is used, 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.

If no route is matched using *URL dispatch*, Pyramid falls back to *traversal* to handle the *request*.

8.16 References

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

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. They are the “meat” of any interesting web 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*.

This chapter describes how view callables should be defined. We’ll have to wait until a following chapter (entitled *View Configuration*) to find out how we actually tell Pyramid to wire up view callables to particular URL patterns and other request circumstances.

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 a `__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 represents 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 a *Response* object in the view callable code and returning it directly or by raising special kinds of exceptions from within the body of a view callable.

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__` method 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 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.

9.4 View Callable Responses

A view callable may 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')
```

Pyramid provides a range of different “exception” classes which inherit from `pyramid.response.Response`. For example, an instance of the class `pyramid.httpexceptions.HTTPFound` is also a valid response object because it inherits from `Response`. For examples, see *HTTP Exceptions* and *Using a View Callable to do an HTTP Redirect*.



You can also return objects from view callables that aren't instances of `pyramid.response.Response` in various circumstances. This can be helpful when writing tests and when attempting to share code between view callables. See *Renderers* for the common way to allow for this. A much less common way to allow for view callables to return non-Response objects is documented in *Changing How Pyramid Treats View Responses*.

9.5 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. It is usually caught and logged there.

However, for convenience, a special set of exceptions exists. When one of these exceptions is raised within a view callable, it will always cause Pyramid to generate a response. These are known as *HTTP exception* objects.

9.5.1 HTTP Exceptions

All `pyramid.httpexceptions` classes which are documented as inheriting from the `pyramid.httpexceptions.HTTPException` are *http exception* objects. Instances of an HTTP exception object may either be *returned* or *raised* from within view code. In either case (return or raise) the instance will be used as the view's response.

For example, the `pyramid.httpexceptions.HTTPUnauthorized` exception can be raised. This will cause a response to be generated with a 401 `Unauthorized` status:

```
1 from pyramid.httpexceptions import HTTPUnauthorized
2
3 def aview(request):
4     raise HTTPUnauthorized()
```

An HTTP exception, instead of being raised, can alternately be *returned* (HTTP exceptions are also valid response objects):

```
1 from pyramid.httpexceptions import HTTPUnauthorized
2
3 def aview(request):
4     return HTTPUnauthorized()
```

A shortcut for creating an HTTP exception is the `pyramid.httpexceptions.exception_response()` function. This function accepts an HTTP status code and returns the corresponding HTTP exception. For example, instead of importing and constructing a `HTTPUnauthorized` response object, you can use the `exception_response()` function to construct and return the same object.

```
1 from pyramid.httpexceptions import exception_response
2
3 def aview(request):
4     raise exception_response(401)
```

This is the case because 401 is the HTTP status code for “HTTP Unauthorized”. Therefore, `raise exception_response(401)` is functionally equivalent to `raise HTTPUnauthorized()`. Documentation which maps each HTTP response code to its purpose and its associated HTTP exception object is provided within `pyramid.httpexceptions`.

New in version 1.1: The `exception_response()` function.

9.5.2 How Pyramid Uses HTTP Exceptions

HTTP exceptions are meant to be used directly by application developers. However, Pyramid itself will raise two HTTP exceptions at various points during normal operations:

- `HTTPNotFound` gets raised when a view to service a request is not found.
- `HTTPForbidden` gets raised when authorization was forbidden by a security policy.

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

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

9.6 Custom Exception Views

The machinery which allows HTTP exceptions to be raised and 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 within 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.

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

9. VIEWS

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, a custom traverser, or a custom view or route predicate 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 helloworld.exceptions import ValidationFailure
3
4 @view_config(context=ValidationFailure, route_name='home')
5 def failed_validation(exc, request):
6     response = Response('Failed validation: %s' % exc.msg)
7     response.status_int = 500
8     return response
```

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 or imperative `add_view` styles.



Pyramid's *exception view* handling logic is implemented as a tween factory function: `pyramid.tweens.excview_tween_factory()`. If Pyramid exception view handling is desired, and tween factories are specified via the `pyramid.tweens` configuration setting, the `pyramid.tweens.excview_tween_factory()` function must be added to the `pyramid.tweens` configuration setting list explicitly. If it is not present, Pyramid will not perform exception view handling.

9.7 Using a View Callable to do an HTTP Redirect

You can issue an HTTP redirect by using the `pyramid.httpexceptions.HTTPFound` class. Raising or returning an instance of this class will cause the client to receive a “302 Found” response.

To do so, you can *return* a `pyramid.httpexceptions.HTTPFound` instance.

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

Alternately, you can *raise* an `HTTPFound` exception instead of returning one.

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

When the instance is raised, it is caught by the default *exception response* handler and turned into a response.

9.8 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

9. VIEWS

`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['somesame']` 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 `; 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.

9.9 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 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__` method 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`.

9.10 Passing Configuration Variables to a View

For information on passing a variable from the configuration .ini files to a view, see *Deployment Settings*.

9.11 Pylons-1.0-Style “Controller” Dispatch

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.

Renderers

A view callable 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.view import view_config
2
3 @view_config(renderer='json')
4 def hello_world(request):
5     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. In such cases, the *renderer* is bypassed entirely.

Various types of *renderers* exist, including serialization *renderers* and *renderers* which use templating systems.

10.1 Writing View Callables Which Use a Renderer

As we've seen, a view callable needn't always return a Response object. Instead, it 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. In practice, renderers obtain application data values from Python dictionaries so, in practice, view callables which use renderers return Python dictionaries.

View callables can *explicitly call* renderers, but typically don't. Instead view configuration declares the renderer used to render a view callable's results. This is done with the `renderer` attribute. For example, this call to `add_view()` associates the `json` renderer with a view callable:

```
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.

Pyramid defines several *Built-In Renderers*, and additional renderers can be added by developers to the system as necessary. See *Adding and Changing Renderers*.

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

As already mentioned, if the *view callable* associated with a *view configuration* returns a Response object (or its instance), any renderer associated with the view configuration is ignored, and the response is passed back to Pyramid unchanged. For example:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 @view_config(renderer='json')
5 def view(request):
6     return Response('OK') # json renderer avoided
```

Likewise for an *HTTP exception* response:

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

You can of course also return the `request.response` attribute instead to avoid rendering:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='json')
4 def view(request):
5     request.response.body = 'OK'
6     return request.response # json renderer avoided
```

10.2 Built-In Renderers

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



Bindings for officially supported templating languages can be found at *Available Add-On Template System Bindings*.

10.2.1 string: String Renderer

The `string` renderer 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.view import view_config
2
3 @view_config(renderer='string')
4 def hello_world(request):
5     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:

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

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

10.2.2 JSON Renderer

The `json` renderer renders view callable results to *JSON*. By default, 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.view import view_config
2
3 @view_config(renderer='json')
4 def hello_world(request):
5     return {'content': 'Hello!'}
```

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

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

The return value needn't be a dictionary, but the return value must contain values serializable by the configured serializer (by default `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 using the api of the `request.response` attribute. See *Varying Attributes of Rendered Responses*.

Serializing Custom Objects

Some objects are not, by default, JSON-serializable (such as datetimes and other arbitrary Python objects). You can, however, register code that makes non-serializable objects serializable in two ways:

- By defining a `__json__` method on objects in your application.
- For objects you don't "own", you can register JSON renderer that knows about an *adapter* for that kind of object.

Using a Custom `__json__` Method

Custom objects can be made easily JSON-serializable in Pyramid by defining a `__json__` method on the object's class. This method should return values natively JSON-serializable (such as ints, lists, dictionaries, strings, and so forth). It should accept a single additional argument, `request`, which will be the active request object at render time.

```

1 from pyramid.view import view_config
2
3 class MyObject(object):
4     def __init__(self, x):
5         self.x = x
6
7     def __json__(self, request):
8         return {'x':self.x}
9
10 @view_config(renderer='json')
11 def objects(request):
12     return [MyObject(1), MyObject(2)]
13
14 # the JSON value returned by ``objects`` will be:
15 # [{"x": 1}, {"x": 2}]

```

Using the `add_adapter` Method of a Custom JSON Renderer

If you aren't the author of the objects being serialized, it won't be possible (or at least not reasonable) to add a custom `__json__` method to their classes in order to influence serialization. If the object passed to the renderer is not a serializable type, and has no `__json__` method, usually a `TypeError` will be raised during serialization. You can change this behavior by creating a custom JSON renderer and adding adapters to handle custom types. The renderer will attempt to adapt non-serializable objects using the registered adapters. A short example follows:

```
1 from pyramid.renderers import JSON
2
3 if __name__ == '__main__':
4     config = Configurator()
5     json_renderer = JSON()
6     def datetime_adapter(obj, request):
7         return obj.isoformat()
8     json_renderer.add_adapter(datetime.datetime, datetime_adapter)
9     config.add_renderer('json', json_renderer)
```

The `add_adapter` method should accept two arguments: the *class* of the object that you want this adapter to run for (in the example above, `datetime.datetime`), and the adapter itself.

The adapter should be a callable. It should accept two arguments: the object needing to be serialized and `request`, which will be the current request object at render time. The adapter should raise a `TypeError` if it can't determine what to do with the object.

See `pyramid.renderers.JSON` and *Adding and Changing Renderers* for more information.

New in version 1.4: Serializing custom objects.

10.2.3 JSONP Renderer

New in version 1.1.

`pyramid.renderers.JSONP` is a JSONP renderer factory helper which implements a hybrid json/jsonp renderer. JSONP is useful for making cross-domain AJAX requests.

Unlike other renderers, a JSONP renderer needs to be configured at startup time “by hand”. Configure a JSONP renderer using the `pyramid.config.Configurator.add_renderer()` method:

```
from pyramid.config import Configurator
from pyramid.renderers import JSONP

config = Configurator()
config.add_renderer('jsonp', JSONP(param_name='callback'))
```

Once this renderer is registered via `add_renderer()` as above, you can use `jsonp` as the `renderer=` parameter to `@view_config` or `pyramid.config.Configurator.add_view()`:

```
from pyramid.view import view_config

@view_config(renderer='jsonp')
def myview(request):
    return {'greeting': 'Hello world'}
```

When a view is called that uses a JSONP renderer:

- If there is a parameter in the request's HTTP query string (aka `request.GET`) that matches the `param_name` of the registered JSONP renderer (by default, `callback`), the renderer will return a JSONP response.
- If there is no callback parameter in the request's query string, the renderer will return a 'plain' JSON response.

Javascript library AJAX functionality will help you make JSONP requests. For example, JQuery has a `getJSON` function, and has equivalent (but more complicated) functionality in its `ajax` function.

For example (Javascript):

```
var api_url = 'http://api.geonames.org/timezoneJSON' +
    '?lat=38.301733840000004' +
    '&lng=-77.45869621' +
    '&username=fred' +
    '&callback=?';
jqxhr = $.getJSON(api_url);
```

The string `callback=?` above in the url param to the JQuery `getJSON` function indicates to jQuery that the query should be made as a JSONP request; the `callback` parameter will be automatically filled in for you and used.

The same custom-object serialization scheme defined used for a “normal” JSON renderer in *Serializing Custom Objects* can be used when passing values to a JSONP renderer too.

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 use the API of the `pyramid.response.Response` attribute available as `request.response` during their execution, to influence associated response behavior.

For example, if you need to change the response status from within a view callable that uses a renderer, assign the status attribute to the response attribute of 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}
```

Note that mutations of `request.response` in views which return a `Response` object directly will have no effect unless the response object returned *is* `request.response`. For example, the following example calls `request.response.set_cookie`, but this call will have no effect, because a different `Response` object is returned.

```
1 from pyramid.response import Response
2
3 def view(request):
4     request.response.set_cookie('abc', '123') # this has no effect
5     return Response('OK') # because we're returning a different response
```

If you mutate `request.response` and you'd like the mutations to have an effect, you must return `request.response`:

```
1 def view(request):
2     request.response.set_cookie('abc', '123')
3     return request.response
```

For more information on attributes of the request, see the API documentation in `pyramid.request`. For more information on the API of `request.response`, see `pyramid.request.Request.response`.

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`:

```
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 should conform to the `pyramid.interfaces.IRendererFactory` interface. It should be capable of creating an object that conforms to the `pyramid.interfaces.IRenderer` interface. A typical class that follows this setup is as follows:

```
1 class RendererFactory:
2     def __init__(self, info):
3         """ Constructor: info will be an object having 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 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 an *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()`, where `config` is an instance of `pyramid.config.Configurator()`:

```
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 `MyAMFRenderer` for each view configuration which includes `amf` as its `renderer` value. The `name` passed to the `MyAMFRenderer` 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:

```
config.add_renderer(name='.jinja2', 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* `.jinja2` 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 `MyJinja2Renderer` for each view configuration which includes anything ending with `.jinja2` in its `renderer` value. The name passed to the `MyJinja2Renderer` constructor will be the full value that was set as `renderer=` in the view configuration.

10.4.2 Adding a Default 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:

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

10.4.3 Changing an Existing Renderer

Pyramid supports overriding almost every aspect of its setup through its *Conflict Resolution* mechanism. This means that in most cases overriding a renderer is as simple as using the `pyramid.config.Configurator.add_renderer()` method to re-define the template extension. For example, if you would like to override the `.txt` extension to specify a new renderer you could do the following:

```
json_renderer = pyramid.renderers.JSON()
config.add_renderer('json', json_renderer)
```

After doing this, any views registered with the `json` renderer will use the new renderer.

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.events import subscriber
2 from pyramid.events 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.

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.

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)
```

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. The path can alternately be an *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 to another location.

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*:

```

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 supported *Mako bindings*. 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()`, or a `renderer=` argument to view configuration (see *Templates Used as Renderers via Configuration*), the renderer representing the template will be provided with a number of *system* values. These values are provided to the template:

request The value provided as the `request` keyword argument to `render_to_response` or `render` *or* the request object passed to the view when the `renderer=` argument to view configuration is being used to render the template.

req An alias for `request`.

context The current Pyramid *context* if `request` was provided as a keyword argument to `render_to_response` or `render`, or `None` if the `request` keyword argument was not provided. This value will always be provided if the template is rendered as the result of a `renderer=` argument to view configuration being used.

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`.

view The view callable object that was used to render this template. If the view callable is a method of a class-based view, this will be an instance of the class that the method was defined on. If the view callable is a function or instance, it will be that function or instance. Note that this value will only be automatically present when a template is rendered as a result of a `renderer=` argument; it will be `None` when the `render_to_response` or `render` APIs are used.

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 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. See *Writing View Callables Which Use a Renderer*.

See also:

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.

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.

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 use the API of the `pyramid.response.Response` object exposed as `request.response` 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 Debugging Templates

A `NameError` exception resulting from rendering a template with an undefined variable (e.g. `${wrong}`) might end up looking like this:

```
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 output tells you which template the error occurred in, as well as displaying the arguments passed to the template itself.

11.5 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 $VENV/bin/pserve myproject.ini
```

To use a setting in the application `.ini` file for the same purpose, set the `pyramid.reload_templates` key to `true` within the application's configuration section, e.g.:

```
1 [app:main]
2 use = egg:MyProject
3 pyramid.reload_templates = true
```

11.6 Available Add-On Template System Bindings

The Pylons Project maintains several packages providing bindings to different templating languages including the following:

| Template Language | Pyramid Bindings | Default Extensions |
|-------------------|--------------------------------|--|
| Chameleon | <code>pyramid_chameleon</code> | <code>.pt</code> , <code>.txt</code> |
| Jinja2 | <code>pyramid_jinja2</code> | <code>.jinja2</code> |
| Mako | <code>pyramid_mako</code> | <code>.mak</code> , <code>.mako</code> |

View Configuration

View lookup is the Pyramid subsystem responsible for finding and invoking a *view callable*. *View configuration* controls how *view lookup* operates in your application. During any given request, view configuration information is compared against request data by the view lookup subsystem in order to find the “best” view callable for that request.

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 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 two ways:

- By running a *scan* against application source code which has a `pyramid.view.view_config` decorator attached to a Python object as per *Adding View Configuration Using the @view_config Decorator*.
- By using the `pyramid.config.Configurator.add_view()` method as per *Adding View Configuration Using add_view()*.

12.1.1 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.

View predicate attributes are an important part of view configuration that enables the *view lookup* subsystem to find and invoke the appropriate view. The greater the 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. The fewer the number of predicates which are supplied to a particular view configuration, the more likely it is that the associated view callable will be invoked. A view with five predicates will always be found and evaluated before a view with two, for example.

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 *Not Found View*.

Other 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:

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.

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 unchanged). Note that if the view callable itself returns a *response* (see *View Callable Responses*), the specified renderer implementation is never called.

http_cache When you supply an `http_cache` value to a view configuration, the `Expires` and `Cache-Control` headers of a response generated by the associated view callable are modified. The value for `http_cache` may be one of the following:

- A nonzero integer. If it's a nonzero integer, it's treated as a number of seconds. This number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=3600` instructs the requesting browser to 'cache this response for an hour, please'.

- A `datetime.timedelta` instance. If it's a `datetime.timedelta` instance, it will be converted into a number of seconds, and that number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=datetime.timedelta(days=1)` instructs the requesting browser to 'cache this response for a day, please'.
- Zero (0). If the value is zero, the `Cache-Control` and `Expires` headers present in all responses from this view will be composed such that client browser cache (and any intermediate caches) are instructed to never cache the response.
- A two-tuple. If it's a two tuple (e.g. `http_cache=(1, {'public':True})`), the first value in the tuple may be a nonzero integer or a `datetime.timedelta` instance; in either case this value will be used as the number of seconds to cache the response. The second value in the tuple must be a dictionary. The values present in the dictionary will be used as input to the `Cache-Control` response header. For example: `http_cache=(3600, {'public':True})` means 'cache for an hour, and add `public` to the `Cache-Control` header of the response'. All keys and values supported by the `webob.cachecontrol.CacheControl` interface may be added to the dictionary. Supplying `{'public':True}` is equivalent to calling `response.cache_control.public = True`.

Providing a non-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value)` within your view's body.

Providing a two-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value[0], **value[1])` within your view's body.

If you wish to avoid influencing, the `Expires` header, and instead wish to only influence `Cache-Control` headers, pass a tuple as `http_cache` with the first element of `None`, e.g.: `(None, {'public':True})`.

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 Configuration*. 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 a 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)`. The decorator may also be an iterable of decorators, in which case they will be applied one after the other to the view, in reverse order. For example:

```
@view_config(..., decorator=(decorator2, decorator1))
def myview(request):
    ...
```

Is similar to doing:

```
@view_config(...)
@decorator2
@decorator1
def myview(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. A `name` argument is typically only used when your application uses *traversal*. 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 only 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 be a string (typically `"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 or a sequence of strings. 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 any value supplied has a `=` sign in it, e.g. `request_param="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.

match_param This param may be either a single string of the format “key=value” or a dict of key/value pairs.

This argument ensures that the view will only be called when the *request* has key/value pairs in its *matchdict* that equal those supplied in the predicate. e.g. `match_param="action=edit"` would require the `action` parameter in the *matchdict* match the right hand side of the expression (`edit`) for the view to “match” the current request.

If the `match_param` is a dict, every key/value pair must match for the predicate to pass.

If `match_param` is not supplied, the view will be invoked without consideration of the keys and values in `request.matchdict`.

New in version 1.2.

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.

check_csrf If specified, this value should be one of `None`, `True`, `False`, or a string representing the ‘check name’. If the value is `True` or a string, CSRF checking will be performed. If the value is `False` or `None`, CSRF checking will not be performed.

If the value provided is a string, that string will be used as the ‘check name’. If the value provided is `True`, `csrf_token` will be used as the check name.

If CSRF checking is performed, the checked value will be the value of `request.params[check_name]`. This value will be compared against the value of `request.session.get_csrf_token()`, and the check will pass if these two values are the same. If the check passes, the associated view will be permitted to execute. If the check fails, the associated view will not be permitted to execute.

Note that using this feature requires a *session factory* to have been configured.

New in version 1.4a2.

physical_path If specified, this value should be a string or a tuple representing the *physical path* of the context found via traversal for this predicate to match as true. For example: `physical_path='/'` or `physical_path='/a/b/c'` or `physical_path=(' ', 'a', 'b', 'c')`. This is not a path prefix match or a regex, it's a whole-path match. It's useful when you want to always potentially show a view when some object is traversed to, but you can't be sure about what kind of object it will be, so you can't use the `context` predicate. The individual path elements inbetween slash characters or in tuple elements should be the Unicode representation of the name of the resource and should not be encoded in any way.

New in version 1.4a3.

`effective_principals`

If specified, this value should be a *principal* identifier or a sequence of principal identifiers. If the `pyramid.request.Request.effective_principals()` method indicates that every principal named in the argument list is present in the current request, this predicate will return `True`; otherwise it will return `False`. For example: `effective_principals=pyramid.security.Authenticated` or `effective_principals=('fred', 'group:admins')`.

New in version 1.4a4.

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.

predicates Pass a key/value pair here to use a third-party predicate registered via `pyramid.config.Configurator.add_view_predicate()`. More than one key/value pair can be used at the same time. See *View and Route Predicates* for more information about third-party predicates.

New in version 1.4a1.

Inverting Predicate Values

You can invert the meaning of any predicate value by wrapping it in a call to `pyramid.config.not_`.

```
1 from pyramid.config import not_  
2  
3 config.add_view(  
4     'mypackage.views.my_view',  
5     route_name='ok',  
6     request_method=not_('POST')  
7 )
```

The above example will ensure that the view is called if the request method is *not* POST, at least if no other view is more specific.

This technique of wrapping a predicate value in `not_` can be used anywhere predicate values are accepted:

- `pyramid.config.Configurator.add_view()`
- `pyramid.view.view_config()`

New in version 1.5.

12.1.2 Adding View Configuration Using the `@view_config` Decorator



Using this feature tends to slow down application startup slightly, as more work is performed at application startup to scan for view configuration declarations. For maximum startup performance, use the view configuration method described in *Adding View Configuration Using `add_view()`* instead.

The `view_config` decorator can be used to associate *view configuration* information with a function, method, or class that acts as a Pyramid view callable.

Here's an example of the `view_config` decorator that lives within 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(route_name='ok', request_method='POST', permission='read')  
6 def my_view(request):  
7     return Response('OK')
```

Using this decorator as above replaces the need to add this imperative configuration stanza:

```
1 config.add_view('mypackage.views.my_view', route_name='ok',
2               request_method='POST', 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 `pyramid.view.view_config` declarations, you *must* use the `scan` method of a `pyramid.config.Configurator`:

```
1 # config is assumed to be an instance of the
2 # pyramid.config.Configurator class
3 config.scan()
```

Please see *Declarative Configuration* 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.

All arguments to the `view_config` decorator mean precisely the same thing as they would if they were passed as arguments to the `pyramid.config.Configurator.add_view()` method save for the `view` argument. Usage of the `view_config` decorator is a form of *declarative configuration*, while `pyramid.config.Configurator.add_view()` is a form of *imperative configuration*. However, they both do the same thing.

@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(route_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. 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(route_name='hello')
5 class MyView(object):
6     def __init__(self, request):
7         self.request = request
8
9     def __call__(self):
10        return Response('hello')
```

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
3
4 @view_config(route_name='edit')
5 @view_config(route_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 a method of a class:

```
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(route_name='hello')
9     def amethod(self):
10         return Response('hello')
```

When the decorator is used against a method of a class, 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', route_name='hello')
5 class MyView(object):
6     def __init__(self, request):
7         self.request = request
8
9     def amethod(self):
10         return Response('hello')
```

12.1.3 Adding View Configuration Using `add_view()`

The `pyramid.config.Configurator.add_view()` method within *pyramid.config* is used to configure a view “imperatively” (without a `view_config` decorator). 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, route_name='hello')
```

The first argument, a *view callable*, is the only required argument. It must either be a Python object which is the view itself or a *dotted Python name* to such an object. In the above example, the `view callable` is the `hello_world` function.

When you use only `add_view()` to add view configurations, you don't need to issue a *scan* in order for the view configuration to take effect.

12.2 @view_defaults Class Decorator

New in version 1.3.

If you use a class as a view, you can use the `pyramid.view.view_defaults` class decorator on the class to provide defaults to the view configuration information used by every `@view_config` decorator that decorates a method of that class.

For instance, if you've got a class that has methods that represent "REST actions", all which are mapped to the same route, but different request methods, instead of this:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 class RESTView(object):
5     def __init__(self, request):
6         self.request = request
7
8     @view_config(route_name='rest', request_method='GET')
9     def get(self):
10         return Response('get')
11
12     @view_config(route_name='rest', request_method='POST')
13     def post(self):
14         return Response('post')
```

```

15     @view_config(route_name='rest', request_method='DELETE')
16     def delete(self):
17         return Response('delete')
18

```

You can do this:

```

1  from pyramid.view import view_defaults
2  from pyramid.view import view_config
3  from pyramid.response import Response
4
5  @view_defaults(route_name='rest')
6  class RESTView(object):
7      def __init__(self, request):
8          self.request = request
9
10     @view_config(request_method='GET')
11     def get(self):
12         return Response('get')
13
14     @view_config(request_method='POST')
15     def post(self):
16         return Response('post')
17
18     @view_config(request_method='DELETE')
19     def delete(self):
20         return Response('delete')

```

In the above example, we were able to take the `route_name='rest'` argument out of the call to each individual `@view_config` statement, because we used a `@view_defaults` class decorator to provide the argument as a default to each view method it possessed.

Arguments passed to `@view_config` will override any default passed to `@view_defaults`.

The `view_defaults` class decorator can also provide defaults to the `pyramid.config.Configurator.add_view()` directive when a decorated class is passed to that directive as its `view` argument. For example, instead of this:

```

1  from pyramid.response import Response
2  from pyramid.config import Configurator
3
4  class RESTView(object):
5      def __init__(self, request):

```

```
6         self.request = request
7
8     def get(self):
9         return Response('get')
10
11    def post(self):
12        return Response('post')
13
14    def delete(self):
15        return Response('delete')
16
17    def main(global_config, **settings):
18        config = Configurator()
19        config.add_route('rest', '/rest')
20        config.add_view(
21            RESTView, route_name='rest', attr='get', request_method='GET')
22        config.add_view(
23            RESTView, route_name='rest', attr='post', request_method='POST')
24        config.add_view(
25            RESTView, route_name='rest', attr='delete', request_method='DELETE')
26        return config.make_wsgi_app()
```

To reduce the amount of repetition in the `config.add_view` statements, we can move the `route_name='rest'` argument to a `@view_defaults` class decorator on the `RESTView` class:

```
1  from pyramid.view import view_defaults
2  from pyramid.response import Response
3  from pyramid.config import Configurator
4
5  @view_defaults(route_name='rest')
6  class RESTView(object):
7      def __init__(self, request):
8          self.request = request
9
10     def get(self):
11         return Response('get')
12
13     def post(self):
14         return Response('post')
15
16     def delete(self):
17         return Response('delete')
18
19    def main(global_config, **settings):
20        config = Configurator()
```

```

21     config.add_route('rest', '/rest')
22     config.add_view(RESTView, attr='get', request_method='GET')
23     config.add_view(RESTView, attr='post', request_method='POST')
24     config.add_view(RESTView, attr='delete', request_method='DELETE')
25     return config.make_wsgi_app()

```

`pyramid.view.view_defaults` accepts the same set of arguments that `pyramid.view.view_config` does, and they have the same meaning. Each argument passed to `view_defaults` provides a default for the view configurations of methods of the class it's decorating.

Normal Python inheritance rules apply to defaults added via `view_defaults`. For example:

```

1  @view_defaults(route_name='rest')
2  class Foo(object):
3      pass
4
5  class Bar(Foo):
6      pass

```

The `Bar` class above will inherit its view defaults from the arguments passed to the `view_defaults` decorator of the `Foo` class. To prevent this from happening, use a `view_defaults` decorator without any arguments on the subclass:

```

1  @view_defaults(route_name='rest')
2  class Foo(object):
3      pass
4
5  @view_defaults()
6  class Bar(Foo):
7      pass

```

The `view_defaults` decorator only works as a class decorator; using it against a function or a method will produce nonsensical results.

12.2.1 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_route('add', '/add.html', factory='mypackage.Blog')
4 config.add_view('myproject.views.add_entry', route_name='add',
5                 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*.

12.2.2 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 `pyramid.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.

12.3 Influencing HTTP Caching

New in version 1.1.

When a non-None `http_cache` argument is passed to a view configuration, Pyramid will set `Expires` and `Cache-Control` response headers in the resulting response, causing browsers to cache the response data for some time. See `http_cache` in *Non-Predicate Arguments* for the allowable values and what they mean.

Sometimes it's undesirable to have these headers set as the result of returning a response from a view, even though you'd like to decorate the view with a view configuration decorator that has `http_cache`. Perhaps there's an alternate branch in your view code that returns a response that should never be cacheable, while the "normal" branch returns something that should always be cacheable. If this is the case, set the `prevent_auto` attribute of the `response.cache_control` object to a non-False value. For example, the below view callable is configured with a `@view_config` decorator that indicates any response from the view should be cached for 3600 seconds. However, the view itself prevents caching from taking place unless there's a `should_cache` GET or POST variable:

```
from pyramid.view import view_config

@view_config(http_cache=3600)
def view(request):
    response = Response()
    if not 'should_cache' in request.params:
        response.cache_control.prevent_auto = True
    return response
```

Note that the `http_cache` machinery will overwrite or add to caching headers you set within the view itself unless you use `prevent_auto`.

You can also turn off the effect of `http_cache` entirely for the duration of a Pyramid application lifetime. To do so, set the `PYRAMID_PREVENT_HTTP_CACHE` environment variable or the `pyramid.prevent_http_cache` configuration value setting to a true value. For more information, see *Preventing HTTP Caching*.

Note that setting `pyramid.prevent_http_cache` will have no effect on caching headers that your application code itself sets. It will only prevent caching headers that would have been set by the Pyramid HTTP caching machinery invoked as the result of the `http_cache` argument to view configuration.

12.4 Debugging View Configuration

See *Displaying Matching Views for a Given URL* for information about how to display each of the view callables that might match for a given URL. This can be an effective way to figure out why a particular view callable is being called instead of the one you'd like to be called.

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 scaffolds, 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.

13.1 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:

```
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.

13.2 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.request.Request.static_url()`.

For example, `add_static_view()` may be fed a name argument which is `http://example.com/images`:

```
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.

13.2.1 Generating Static Asset URLs

When a `add_static_view()` method is used to register a static asset directory, a special helper API named `pyramid.request.Request.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.renderers import render_to_response
2
3 def my_view(request):
4     css_url = request.static_url('mypackage:assets/1/foo.css')
5     js_url = request.static_url('mypackage:assets/2/foo.js')
6     return render_to_response('templates/my_template.pt',
7                               dict(css_url=css_url, js_url=js_url),
8                               request=request)
```

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 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 request.static_url('mypackage:images/logo.png')
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.

13.3 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.

13.3.1 Root-Relative Custom Static View (URL Dispatch Only)

The `pyramid.static.static_view` helper class generates a Pyramid view callable. This view callable can serve static assets from a directory. An instance of this class is actually used by the `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_view` static view cannot be made root-relative when you use traversal unless it's registered as a *Not Found View*.

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_view` class inside a `static.py` file in your application root as below.

```
1 from pyramid.static import static_view
2 static_view = static_view('/path/to/static/dir', use_subpath=True)
```



For better cross-system flexibility, use an *asset specification* as the argument to `static_view` instead of a physical absolute filesystem path, e.g. `mypackage:static` instead of `/path/to/mypackage/static`.

Subsequently, you may wire the files that are served by this view up to be accessible as `/<filename>` using a configuration method in your application's startup code.

```
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')
5 config.add_view('myapp.static.static_view', route_name='catchall_static')
```

The special name `*subpath` above is used by the `static_view` view callable to signify the path of the file relative to the directory you're serving.

13.3.2 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 pyramid.response import FileResponse
3
4 def favicon_view(request):
5     here = os.path.dirname(__file__)
6     icon = os.path.join(here, 'static', 'favicon.ico')
7     return FileResponse(icon, request=request)
```

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 creates a `pyramid.response.FileResponse` using the file path as the response’s `path` argument and the request as the response’s `request` argument. `pyramid.response.FileResponse` will serve the file as quickly as possible when it’s used this way. It makes sure to set the right content length and `content_type` too based on the file extension of the file you pass.

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 config.add_view('myapp.views.favicon_view', route_name='favicon')
```

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.

13.4 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:

- Individual template files.
- A directory containing multiple template files.
- Individual static files served up by an instance of the `pyramid.static.static_view` helper class.
- A directory of static files served up by an instance of the `pyramid.static.static_view` helper class.
- Any other asset (or set of assets) addressed by code that uses the setuptools *pkg_resources* API.

13.4.1 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 as a basis for its *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 `pyramid.response.Response` class, which is a subclass of the `webob.Response` class. Users can also return an instance of `pyramid.response.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, header list, and `app_iter` (body) values.

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.

14.1 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.json_body The JSON-decoded contents of the body of the request. See *Dealing With A JSON-Encoded Request Body*.

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 was not provided).



Full API documentation for the Pyramid request object is available in *pyramid.request*.

14.1.1 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`, `matchdict`, and `matched_route`. These attributes are documented further within the `pyramid.request.Request` API documentation.

14.1.2 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`.

14.1.3 Methods

There are methods of request objects documented in `pyramid.request.Request` but you'll find that you won't use very many of them. Here are a couple that might be useful:

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 `pyramid.response.Response` object. You can also use this for subrequests, or testing.

14.1.4 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.

14.1.5 Multidict

Several attributes of a `WebOb` request are “multidict”; structures (such as `request.GET`, `request.POST`, and `request.params`). A multidict 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.

API documentation for a multidict exists as `pyramid.interfaces.IMultiDict`.

14.1.6 Dealing With A JSON-Encoded Request Body

New in version 1.1.

`pyramid.request.Request.json_body` is a property that returns a *JSON*-decoded representation of the request body. If the request does not have a body, or the body is not a properly JSON-encoded value, an exception will be raised when this attribute is accessed.

This attribute is useful when you invoke a Pyramid view callable via e.g. jQuery's `$.ajax` function, which has the potential to send a request with a JSON-encoded body.

Using `request.json_body` is equivalent to:

```
from json import loads
loads(request.body, encoding=request.charset)
```

Here's how to construct an AJAX request in Javascript using *jQuery* that allows you to use the `request.json_body` attribute when the request is sent to a Pyramid application:

```
jQuery.ajax({type:'POST',
             url: 'http://localhost:6543/', // the pyramid server
             data: JSON.stringify({'a':1}),
             contentType: 'application/json; charset=utf-8'});
```

When such a request reaches a view in your application, the `request.json_body` attribute will be available in the view callable body.

```
@view_config(renderer='string')
def aview(request):
    print(request.json_body)
    return 'OK'
```

For the above view, printed to the console will be:

```
{u'a': 1}
```

For bonus points, here's a bit of client-side code that will produce a request that has a body suitable for reading via `request.json_body` using Python's `urllib2` instead of a Javascript AJAX request:

```
import urllib2
import json

json_payload = json.dumps({'a':1})
headers = {'Content-Type': 'application/json; charset=utf-8'}
req = urllib2.Request('http://localhost:6543/', json_payload, headers)
resp = urllib2.urlopen(req)
```

14.1.7 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.

14. REQUEST AND RESPONSE OBJECTS

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 scaffold, because these all use the `pyramid_tm` package. The cleanup done by `DBSession.remove` is unnecessary when `pyramid_tm` *middleware* is configured into the application.

14.1.8 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.

14.2 Response

The Pyramid response object can be imported as `pyramid.response.Response`. This class is a subclass of the `webob.Response` class. The subclass does not add or change any functionality, so the WebOb Response documentation will be completely relevant for this class as well.

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 typically derives from this underlying state. Here are some 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.

14.2.1 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.

14.2.2 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.:

```
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.

14.2.3 Exception Responses

To facilitate error responses like 404 Not Found, the module `pyramid.httpexceptions` 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` module. This import location contains subclasses and replacements that mirror those in the `webob.exc` module.

Each class is named `pyramid.httpexceptions.HTTP*`, where `*` is the reason for the error. For instance, `pyramid.httpexceptions.HTTPNotFound` subclasses `pyramid.response.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)
```

14.2.4 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.

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.

15.1 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 limitations:

- 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 `pyramid.config.Configurator.set_session_factory()` method.

```
1 from pyramid.session import SignedCookieSessionFactory
2 my_session_factory = SignedCookieSessionFactory('itsaseekreet')
3
4 from pyramid.config import Configurator
5 config = Configurator()
6 config.set_session_factory(my_session_factory)
```



By default the `SignedCookieSessionFactory()` implementation is *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. And if you use this sessioning implementation, and you inadvertently create a cross-site scripting vulnerability in your application, because the session data is stored unencrypted in a cookie, it will also be easier for evildoers to obtain the current user's cross-site scripting token. In short, 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", and you are sure your application has no cross-site scripting vulnerabilities.

15.2 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')
```

The first time this view is invoked produces Fred was not in the session. Subsequent invocations produce Fred was in the session, assuming of course that the client side maintains the session's identity across multiple requests.

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.

15.3 Using Alternate Session Factories

The following session factories exist at the time of this writing.

| Session Factory | Back-end | Description |
|------------------------|----------|---|
| pyramid_redis_sessions | Redis | Server-side session library for Pyramid, using Redis for storage. |
| pyramid_beaker | Beaker | Session factory for Pyramid backed by the Beaker sessioning system. |

15.4 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.

15.5 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.

15.5.1 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.

15.5.2 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='')`

```
>>> request.session.flash('info message')
>>> request.session.pop_flash()
['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.

```
>>> request.session.flash('info message')
>>> request.session.pop_flash()
['info message']
>>> request.session.pop_flash()
[]
```

15.5.3 Using the `session.peak_flash` Method

Once one or more messages has been added to a flash queue by the `session.flash()` API, the `session.peak_flash()` API can be used to “peek” at that queue. Unlike `session.pop_flash()`, the queue is not popped from flash storage.

`peak_flash(queue='')`

```
>>> request.session.flash('info message')
>>> request.session.peak_flash()
['info message']
>>> request.session.peak_flash()
['info message']
>>> request.session.pop_flash()
['info message']
>>> request.session.peak_flash()
[]
```

15.6 Preventing Cross-Site Request Forgery Attacks

Cross-site request forgery attacks are a phenomenon whereby a user who is logged in to your website might inadvertently load a URL because it is linked from, or embedded in, an attacker's website. If the URL is one that may modify or delete data, the consequences can be dire.

You can avoid most of these attacks by issuing a unique token to the browser and then requiring that it be present in all potentially unsafe requests. Pyramid sessions provide facilities to create and check CSRF tokens.

To use CSRF tokens, you must first enable a *session factory* as described in *Using The Default Session Factory* or *Using Alternate Session Factories*.

15.6.1 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, then the existing token will be returned. If no CSRF token previously existed for this session, then 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, or supply it as a request header in AJAX requests.

For example, include the CSRF token as a hidden field:

```
<form method="post" action="/myview">
  <input type="hidden" name="csrf_token" value="{request.session.get_csrf_token()}">
  <input type="submit" value="Delete Everything">
</form>
```

Or, include it as a header in a jQuery AJAX request:

```
var csrfToken = ${request.session.get_csrf_token()};
$.ajax({
  type: "POST",
  url: "/myview",
  headers: { 'X-CSRF-Token': csrfToken }
}).done(function() {
  alert("Deleted");
});
```

The handler for the URL that receives the request should then require that the correct CSRF token is supplied.

15.6.2 Checking CSRF Tokens Manually

In request handling code, you can check the presence and validity of a CSRF token with `pyramid.session.check_csrf_token(request)`. If the token is valid, it will return True, otherwise it will raise HTTPBadRequest. Optionally, you can specify raises=False to have the check return False instead of raising an exception.`

By default, it checks for a GET or POST parameter named `csrf_token` or a header named `X-CSRF-Token`.

```
from pyramid.session import check_csrf_token

def myview(request):
    # Require CSRF Token
    check_csrf_token(request)

    # ...
```

15.6.3 Checking CSRF Tokens With A View Predicate

A convenient way to require a valid CSRF Token for a particular view is to include `check_csrf=True` as a view predicate. See `pyramid.config.Configurator.add_route()`.

```
@view_config(request_method='POST', check_csrf=True, ...)
def myview(request):
    ...
```

15.6.4 Using the `session.new_csrf_token` Method

To explicitly create a new CSRF token, 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()
```

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*.

16.1 Configuring an Event Listener Imperatively

You can imperatively configure a subscriber function to be called for some event type via the `add_subscriber()` method:

```
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.

See also:

See also *Configurator*.

16.2 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.

16.3 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`).

16.4 Creating Your Own Events

In addition to using the events that the Pyramid framework creates, you can create your own events for use in your application. This can be useful to decouple parts of your application.

For example, suppose your application has to do many things when a new document is created. Rather than putting all this logic in the view that creates the document, you can create the document in your view and then fire a custom event. Subscribers to the custom event can take other actions, such as indexing the document, sending email, or sending a message to a remote system.

An event is simply an object. There are no required attributes or method for your custom events. In general, your events should keep track of the information that subscribers will need. Here are some example custom event classes:

```
1 class DocCreated(object):
2     def __init__(self, doc, request):
3         self.doc = doc
4         self.request = request
5
6 class UserEvent(object):
7     def __init__(self, user):
8         self.user = user
9
10 class UserLoggedIn(UserEvent):
11     pass
```

Some Pyramid applications choose to define custom events classes in an `events` module.

You can subscribe to custom events in the same way that you subscribe to Pyramid events – either imperatively or with a decorator. You can also use custom events with *subscriber predicates*. Here’s an example of subscribing to a custom event with a decorator:

```
1 from pyramid.events import subscriber
2 from .events import DocCreated
3 from .index import index_doc
4
5 @subscriber(DocCreated)
6 def index_doc(event):
7     # index the document using our application's index_doc function
8     index_doc(event.doc, event.request)
```

The above example assumes that the application defines a `DocCreated` event class and an `index_doc` function.

To fire your custom events use the `pyramid.registry.Registry.notify()` method, which is most often accessed as `request.registry.notify`. For example:


```
1 from .events import DocCreated
2
3 def new_doc_view(request):
4     doc = MyDoc()
5     event = DocCreated(doc, request)
6     request.registry.notify(event)
7     return {'document': doc}
```

This example view will notify all subscribers to the custom `DocCreated` event.

Note that when you fire an event, all subscribers are run synchronously so it's generally not a good idea to create event handlers that may take a long time to run. Although event handlers could be used as a central place to spawn tasks on your own message queues.

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.

17.1 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 |
|---------------------------|---|
| PYRAMID_RELOAD_TEMPLATES | <code>pyramid.reload_templates</code> or <code>reload_templates</code> |

17.2 Reloading Assets

Don't cache any asset file data when this value is true.

See also:

See also *Overriding Assets*.

| Environment Variable Name | Config File Setting Name |
|---------------------------|--|
| PYRAMID_RELOAD_ASSETS | pyramid.reload_assets or reload_assets |



For backwards compatibility purposes, aliases can be used for configuring asset reloading: PYRAMID_RELOAD_RESOURCES (envvar) and pyramid.reload_resources (config file).

17.3 Debugging Authorization

Print view authorization failure and success information to stderr when this value is true.

See also:

See also *Debugging View Authorization Failures*.

| Environment Variable Name | Config File Setting Name |
|-----------------------------|--|
| PYRAMID_DEBUG_AUTHORIZATION | pyramid.debug_authorization or debug_authorization |

17.4 Debugging Not Found Errors

Print view-related NotFound debug messages to stderr when this value is true.

See also:

See also *NotFound Errors*.

| Environment Variable Name | Config File Setting Name |
|---------------------------|--|
| PYRAMID_DEBUG_NOTFOUND | pyramid.debug_notfound or debug_notfound |

17.5 Debugging Route Matching

Print debugging messages related to *url dispatch* route matching when this value is true.

See also:

See also *Debugging Route Matching*.

| Environment Variable Name | Config File Setting Name |
|---------------------------|---|
| PYRAMID_DEBUG_ROUTEMATCH | pyramid.debug_routematch or debug_routematch |

17.6 Preventing HTTP Caching

Prevent the `http_cache` view configuration argument from having any effect globally in this process when this value is true. No http caching-related response headers will be set by the Pyramid `http_cache` view configuration feature when this is true.

See also:

See also *Influencing HTTP Caching*.

| Environment Variable Name | Config File Setting Name |
|----------------------------|---|
| PYRAMID_PREVENT_HTTP_CACHE | pyramid.prevent_http_cache or prevent_http_cache |

17.7 Debugging All

Turns on all debug* settings.

| Environment Variable Name | Config File Setting Name |
|---------------------------|--------------------------------|
| PYRAMID_DEBUG_ALL | pyramid.debug_all or debug_all |

17.8 Reloading All

Turns on all `reload*` settings.

| Environment Variable Name | Config File Setting Name |
|---------------------------------|--|
| <code>PYRAMID_RELOAD_ALL</code> | <code>pyramid.reload_all</code> or <code>reload_all</code> |

17.9 Default Locale Name

The value supplied here is used as the default locale name when a *locale negotiator* is not registered.

See also:

See also *Localization-Related Deployment Settings*.

| Environment Variable Name | Config File Setting Name |
|--|--|
| <code>PYRAMID_DEFAULT_LOCALE_NAME</code> | <code>pyramid.default_locale_name</code> or <code>default_locale_name</code> |

17.10 Including Packages

`pyramid.includes` instructs your application to include other packages. Using the setting is equivalent to using the `pyramid.config.Configurator.include()` method.

| Config File Setting Name |
|-------------------------------|
| <code>pyramid.includes</code> |

The value assigned to `pyramid.includes` should be a sequence. The sequence can take several different forms.

1. It can be a string.

If it is a string, the package names can be separated by spaces:

```
package1 package2 package3
```

The package names can also be separated by carriage returns:

```
package1
package2
package3
```

2. It can be a Python list, where the values are strings:

```
['package1', 'package2', 'package3']
```

Each value in the sequence should be a *dotted Python name*.

17.10.1 pyramid.includes vs. pyramid.config.Configurator.include()

Two methods exist for including packages: `pyramid.includes` and `pyramid.config.Configurator.include()`. This section explains their equivalence.

Using PasteDeploy

Using the following `pyramid.includes` setting in the PasteDeploy `.ini` file in your application:

```
[app:main]
pyramid.includes = pyramid_debugtoolbar
                  pyramid_tm
```

Is equivalent to using the following statements in your configuration code:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     config = Configurator(settings=settings)
5     # ...
6     config.include('pyramid_debugtoolbar')
7     config.include('pyramid_tm')
8     # ...
```

It is fine to use both or either form.

Plain Python

Using the following `pyramid.includes` setting in your plain-Python Pyramid application:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     settings = {'pyramid.includes': 'pyramid_debugtoolbar pyramid_tm'}
5     config = Configurator(settings=settings)
```

Is equivalent to using the following statements in your configuration code:

```
1 from pyramid.config import Configurator
2
3 if __name__ == '__main__':
4     settings = {}
5     config = Configurator(settings=settings)
6     config.include('pyramid_debugtoolbar')
7     config.include('pyramid_tm')
```

It is fine to use both or either form.

17.11 Explicit Tween Configuration

This value allows you to perform explicit *tween* ordering in your configuration. Tweens are bits of code used by add-on authors to extend Pyramid. They form a chain, and require ordering.

Ideally, you won't need to use the `pyramid.tweens` setting at all. Tweens are generally ordered and included “implicitly” when an add-on package which registers a tween is “included”. Packages are included when you name a `pyramid.includes` setting in your configuration or when you call `pyramid.config.Configurator.include()`.

Authors of included add-ons provide “implicit” tween configuration ordering hints to Pyramid when their packages are included. However, the implicit tween ordering is only best-effort. Pyramid will attempt to provide an implicit order of tweens as best it can using hints provided by add-on authors, but because it's only best-effort, if very precise tween ordering is required, the only surefire way to get it is to use an explicit tween order. You may be required to inspect your tween ordering (see *Displaying “Tweens”*) and add a `pyramid.tweens` configuration value at the behest of an add-on author.

| Config File Setting Name |
|--------------------------|
|--------------------------|

| |
|-----------------------------|
| <code>pyramid.tweens</code> |
|-----------------------------|

The value assigned to `pyramid.tweens` should be a sequence. The sequence can take several different forms.

1. It can be a string.

If it is a string, the tween names can be separated by spaces:

```
pkg.tween_factory1 pkg.tween_factory2 pkg.tween_factory3
```

The tween names can also be separated by carriage returns::

```
pkg.tween_factory1
pkg.tween_factory2
pkg.tween_factory3
```

2. It can be a Python list, where the values are strings:

```
['pkg.tween_factory1', 'pkg.tween_factory2', 'pkg.tween_factory3']
```

Each value in the sequence should be a *dotted Python name*.

17.11.1 PasteDeploy Configuration vs. Plain-Python Configuration

Using the following `pyramid.twens` setting in the PasteDeploy `.ini` file in your application:

```
[app:main]
pyramid.twens = pyramid_debugtoolbar.toolbar.tween_factory
                pyramid.twens.excview_tween_factory
                pyramid_tm.tm_tween_factory
```

Is equivalent to using the following statements in your configuration code:

```
1 from pyramid.config import Configurator
2
3 def main(global_config, **settings):
4     settings['pyramid.twens'] = [
5         'pyramid_debugtoolbar.toolbar.tween_factory',
6         'pyramid.twens.excview_tween_factory',
7         'pyramid_tm.tm_tween_factory',
8     ]
9     config = Configurator(settings=settings)
```

It is fine to use both or either form.

17.12 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
3 pyramid.reload_templates = true
4 pyramid.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 \
  $VENV/bin/pserve 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 `pyramid.debug_`). on in one fell swoop, you can use `PYRAMID_DEBUG_ALL=1` as an environment variable setting or you may use `pyramid.debug_all=true` in the config file. Note that this does not affect settings that do not start with `pyramid.debug_*` such as `pyramid.reload_templates`.

If you want to turn all `pyramid.reload` settings (every setting that starts with `pyramid.reload_`) on in one fell swoop, you can use `PYRAMID_RELOAD_ALL=1` as an environment variable setting or you may use `pyramid.reload_all=true` in the config file. Note that this does not affect settings that do not start with `pyramid.reload_*` such as `pyramid.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.

17.13 Understanding the Distinction Between `reload_templates` and `reload_assets`

The difference between `pyramid.reload_assets` and `pyramid.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 `pyramid.reload_templates` is true, Pyramid takes advantage of the underlying templating systems' ability to check for file modifications to an individual template file. When `pyramid.reload_templates` is true but `pyramid.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 `pyramid.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 `pyramid.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 `pyramid.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. `pyramid.reload_assets` makes the system *very slow* when templates are in use. Never set `pyramid.reload_assets` to True on a production system.

17.14 Adding A Custom Setting

From time to time, you may need to add a custom setting to your application. Here's how:

- If you're using an `.ini` file, change the `.ini` file, adding the setting to the `[app:foo]` section representing your Pyramid application. For example:

```
[app:main]
# .. other settings
debug_frobnosticator = True
```

17. ENVIRONMENT VARIABLES AND .INI FILE SETTINGS

- In the `main()` function that represents the place that your Pyramid WSGI application is created, anticipate that you'll be getting this key/value pair as a setting and do any type conversion necessary.

If you've done any type conversion of your custom value, reset the converted values into the `settings` dictionary *before* you pass the dictionary as settings to the *Configurator*. For example:

```
def main(global_config, **settings):
    # ...
    from pyramid.settings import asbool
    debug_frobnosticator = asbool(settings.get(
        'debug_frobnosticator', 'false'))
    settings['debug_frobnosticator'] = debug_frobnosticator
    config = Configurator(settings=settings)
```



It's especially important that you mutate the `settings` dictionary with the converted version of the variable *before* passing it to the *Configurator*: the configurator makes a *copy* of `settings`, it doesn't use the one you pass directly.

- When creating an *includeme* function that will be later added to your application's configuration you may access the `settings` dictionary through the instance of the *Configurator* that is passed into the function as its only argument. For Example:

```
def includeme(config):
    settings = config.registry.settings
    debug_frobnosticator = settings['debug_frobnosticator']
```

- In the runtime code from where you need to access the new settings value, find the value in the `registry.settings` dictionary and use it. In *view* code (or any other code that has access to the request), the easiest way to do this is via `request.registry.settings`. For example:

```
settings = request.registry.settings
debug_frobnosticator = settings['debug_frobnosticator']
```

If you wish to use the value in code that does not have access to the request and you wish to use the value, you'll need to use the `pyramid.threadlocal.get_current_registry()` API to obtain the current registry, then ask for its `settings` attribute. For example:

```
registry = pyramid.threadlocal.get_current_registry()
settings = registry.settings
debug_frobnosticator = settings['debug_frobnosticator']
```

Logging

Pyramid allows you to make use of the Python standard library `logging` module. This chapter describes how to configure logging and how to send log messages to loggers that you've configured.



This chapter assumes you've used a *scaffold* to create a project which contains `development.ini` and `production.ini` files which help configure logging. All of the scaffolds which ship along with Pyramid do this. If you're not using a scaffold, or if you've used a third-party scaffold which does not create these files, the configuration information in this chapter may not be applicable.

18.1 Logging Configuration

A Pyramid project created from a *scaffold* is configured to allow you to send messages to Python standard library logging package loggers from within your application. In particular, the *PasteDeploy* `development.ini` and `production.ini` files created when you use a scaffold include a basic configuration for the Python logging package.

PasteDeploy `.ini` files use the Python standard library `ConfigParser` format; this is the same format used as the Python *logging module's Configuration file format*. The application-related and logging-related sections in the configuration file can coexist peacefully, and the logging-related sections in the file are used from when you run `pserve`.

The `pserve` command calls the `pyramid.paster.setup_logging()` function, a thin wrapper around the `logging.config.fileConfig()` using the specified `.ini` file if it contains a

18. LOGGING

[loggers] section (all of the scaffold-generated .ini files do). setup_logging reads the logging configuration from the ini file upon which pserve was invoked.

Default logging configuration is provided in both the default development.ini and the production.ini file. The logging configuration in the development.ini file is as follows:

```
1  # Begin logging configuration
2
3  [loggers]
4  keys = root, {{package_logger}}
5
6  [handlers]
7  keys = console
8
9  [formatters]
10 keys = generic
11
12 [logger_root]
13 level = INFO
14 handlers = console
15
16 [logger_{{package_logger}}]
17 level = DEBUG
18 handlers =
19 qualname = {{package}}
20
21 [handler_console]
22 class = StreamHandler
23 args = (sys.stderr,)
24 level = NOTSET
25 formatter = generic
26
27 [formatter_generic]
28 format = %(asctime)s %(levelname)-5.5s [% (name)s] [% (threadName)s] %(message)s
29
30 # End logging configuration
```

The production.ini file uses the WARN level in its logger configuration, but it is otherwise identical.

The name {{package_logger}} above will be replaced with the name of your project's *package*, which is derived from the name you provide to your project. For instance, if you do:

```
1 pcreate -s starter MyApp
```

The logging configuration will literally be:

```

1  # Begin logging configuration
2
3  [loggers]
4  keys = root, myapp
5
6  [handlers]
7  keys = console
8
9  [formatters]
10 keys = generic
11
12 [logger_root]
13 level = INFO
14 handlers = console
15
16 [logger_myapp]
17 level = DEBUG
18 handlers =
19 qualname = myapp
20
21 [handler_console]
22 class = StreamHandler
23 args = (sys.stderr,)
24 level = NOTSET
25 formatter = generic
26
27 [formatter_generic]
28 format = %(asctime)s %(levelname)-5.5s [% (name)s] [% (threadName)s] %(message)s
29
30 # End logging configuration

```

In this logging configuration:

- a logger named `root` is created that logs messages at a level above or equal to the `INFO` level to `stderr`, with the following format:

```

2007-08-17 15:04:08,704 INFO [packagename]
                             Loading resource, id: 86

```

- a logger named `myapp` is configured that logs messages sent at a level above or equal to `DEBUG` to `stderr` in the same format as the root logger.

The root logger will be used by all applications in the Pyramid process that ask for a logger (via `logging.getLogger`) that has a name which begins with anything except your project's package

18. LOGGING

name (e.g. `myapp`). The logger with the same name as your package name is reserved for your own usage in your Pyramid application. Its existence means that you can log to a known logging location from any Pyramid application generated via a scaffold.

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

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

Some scaffolds configure additional loggers for additional subsystems they use (such as SQLAlchemy). Take a look at the `production.ini` and `development.ini` files rendered when you create a project from a scaffold.

18.2 Sending Logging Messages

Python's special `__name__` variable refers to the current module's fully qualified name. From any module in a package named `myapp`, the `__name__` builtin variable will always be something like `myapp`, or `myapp.subpackage` or `myapp.package.subpackage` if your project is named `myapp`. Sending a message to this logger will send it to the `myapp` logger.

To log messages to the package-specific logger configured in your `.ini` file, simply create a logger object using the `__name__` builtin and call methods on it.

```
1 import logging
2 log = logging.getLogger(__name__)
3
4 def myview(request):
5     content_type = 'text/plain'
6     content = 'Hello World!'
7     log.debug('Returning: %s (content-type: %s)', content, content_type)
8     request.response.content_type = content_type
9     return request.response
```

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

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

18.3 Filtering log messages

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

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

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

then add it to the list of loggers:

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

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

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

```
# Begin logging configuration

[loggers]
keys = root, myapp

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

All of the child loggers of the `myapp` logger will inherit the `DEBUG` level unless they're explicitly set differently. Meaning the `myapp.views`, `myapp.models` (and all your app's modules') loggers by default have an effective level of `DEBUG` too.

For more advanced filtering, the logging module provides a `Filter` object; however it cannot be used directly from the configuration file.

18.4 Advanced Configuration

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

```
[handler_filelog]
class = FileHandler
args = ('%(here)s/myapp.log', 'a')
level = INFO
formatter = generic
```

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

```
[handlers]
keys = console, myapp, filelog
```

and finally utilized by a logger.

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

These final 3 lines of configuration directs all of the root logger's output to the `myapp.log` as well as the console.

18.5 Logging Exceptions

To log (or email) exceptions generated by your Pyramid application, use the `pyramid_exclog` package. Details about its configuration are in its documentation.

18.6 Request Logging with Paste's TransLogger

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

Like any standard *middleware* with a Paste entry point, TransLogger can be configured to wrap your application using `.ini` file syntax. First, rename your Pyramid `.ini` file's `[app:main]` section to `[app:mypyramidapp]`, then add a `[filter:translogger]` section, then use a `[pipeline:main]` section file to form a WSGI pipeline with both the translogger and your application in it. For instance, change from this:

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

To this:

```
[app:mypyramidapp]
use = egg:MyProject

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

[pipeline:main]
pipeline = translogger
          mypyramidapp
```

Using PasteDeploy this way to form and serve a pipeline is equivalent to wrapping your app in a TransLogger instance via the bottom of the `main` function of your project's `__init__.py` file:

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

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

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

```
00:50:53,694 INFO [myapp.views] Returning: Hello World!
              (content-type: text/plain)
00:50:53,695 INFO [wsgi] 192.168.1.111 - - [11/Aug/2011:20:09:33 -0700] "GET /hello
HTTP/1.1" 404 - "-"
"Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.6) Gecko/20070725
Firefox/2.0.0.6"
```

18. LOGGING

To direct TransLogger to an `access.log` `FileHandler`, we need to add that `FileHandler` to the list of handlers (named `accesslog`), and ensure that the `wsgi` logger is configured and uses this handler accordingly:

```
# Begin logging configuration

[loggers]
keys = root, myapp, wsgi

[handlers]
keys = console, accesslog

[logger_wsgi]
level = INFO
handlers = accesslog
qualname = wsgi
propagate = 0

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

As mentioned above, non-root loggers by default propagate their log records to the root logger's handlers (currently the console handler). Setting `propagate` to 0 (False) here disables this; so the `wsgi` logger directs its records only to the `accesslog` handler.

Finally, there's no need to use the generic formatter with TransLogger as TransLogger itself provides all the information we need. We'll use a formatter that passes-through the log messages as is. Add a new formatter called `accesslog` by include the following in your configuration file:

```
[formatters]
keys = generic, accesslog

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

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

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

PasteDeploy Configuration Files

Packages generated via a *scaffold* make use of a system created by Ian Bicking named *PasteDeploy*. PasteDeploy defines a way to declare *WSGI* application configuration in an `.ini` file.

Pyramid uses this configuration file format in input to its *WSGI* server runner `pserve`, as well as other commands such as `pviews`, `pshell`, `proutes`, and `ptweens`.

PasteDeploy is not a particularly integral part of Pyramid. It's possible to create a Pyramid application which does not use PasteDeploy at all. We show a Pyramid application that doesn't use PasteDeploy in *Creating Your First Pyramid Application*. However, all Pyramid scaffolds render PasteDeploy configuration files, to provide new developers with a standardized way of setting deployment values, and to provide new users with a standardized way of starting, stopping, and debugging an application.

This chapter is not a replacement for documentation about PasteDeploy; it only contextualizes the use of PasteDeploy within Pyramid. For detailed documentation, see <http://pythonpaste.org/deploy/>.

19.1 PasteDeploy

PasteDeploy is the system that Pyramid uses to allow *deployment settings* to be spelled using an `.ini` configuration file format. It also allows the `pserve` command to work. Its configuration format provides a convenient place to define application *deployment settings* and *WSGI* server settings, and its server runner allows you to stop and start a Pyramid application easily.

19.1.1 Entry Points and PasteDeploy .ini Files

In the *Creating a Pyramid Project* chapter, we breezed over the meaning of a configuration line in the `deployment.ini` file. This was the `use = egg:MyProject` line in the `[app:main]` section. We breezed over it because it's pretty confusing and "too much information" for an introduction to the system. We'll try to give it a bit of attention here. Let's see the config file again:

```
1  ###
2  # app configuration
3  # http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/environment.html
4  ###
5
6  [app:main]
7  use = egg:MyProject
8
9  pyramid.reload_templates = true
10 pyramid.debug_authorization = false
11 pyramid.debug_notfound = false
12 pyramid.debug_routematch = false
13 pyramid.default_locale_name = en
14 pyramid.includes =
15     pyramid_debugtoolbar
16
17 # By default, the toolbar only appears for clients from IP addresses
18 # '127.0.0.1' and ':::1'.
19 # debugtoolbar.hosts = 127.0.0.1 :::1
20
21 ###
22 # wsgi server configuration
23 ###
24
25 [server:main]
26 use = egg:waitress#main
27 host = 0.0.0.0
28 port = 6543
29
30 ###
31 # logging configuration
32 # http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/logging.html
33 ###
34
35 [loggers]
36 keys = root, myproject
37
38 [handlers]
39 keys = console
```

```
40
41 [formatters]
42 keys = generic
43
44 [logger_root]
45 level = INFO
46 handlers = console
47
48 [logger_myproject]
49 level = DEBUG
50 handlers =
51 qualname = myproject
52
53 [handler_console]
54 class = StreamHandler
55 args = (sys.stderr,)
56 level = NOTSET
57 formatter = generic
58
59 [formatter_generic]
60 format = %(asctime)s %(levelname)-5.5s [% (name)s] [% (threadName)s] %(message)s
```

The line in `[app:main]` above that says `use = egg:MyProject` is actually shorthand for a longer spelling: `use = egg:MyProject#main`. The `#main` part is omitted for brevity, as `#main` is a default defined by PasteDeploy. `egg:MyProject#main` is a string which has meaning to PasteDeploy. It points at a *setuptools* entry point named `main` defined in the `MyProject` project.

Take a look at the generated `setup.py` file for this project.

```
1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 with open(os.path.join(here, 'README.txt')) as f:
7     README = f.read()
8 with open(os.path.join(here, 'CHANGES.txt')) as f:
9     CHANGES = f.read()
10
11 requires = [
12     'pyramid',
13     'pyramid_chameleon',
14     'pyramid_debugtoolbar',
15     'waitress',
16 ]
```

```
17
18 setup(name='MyProject',
19       version='0.0',
20       description='MyProject',
21       long_description=README + '\n\n' + CHANGES,
22       classifiers=[
23         "Programming Language :: Python",
24         "Framework :: Pyramid",
25         "Topic :: Internet :: WWW/HTTP",
26         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
27     ],
28     author='',
29     author_email='',
30     url='',
31     keywords='web pyramid pylons',
32     packages=find_packages(),
33     include_package_data=True,
34     zip_safe=False,
35     install_requires=requires,
36     tests_require=requires,
37     test_suite="myproject",
38     entry_points="""\
39     [paste.app_factory]
40     main = myproject:main
41     """,
42 )
```

Note that `entry_points` is assigned 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.

The `egg:` prefix in `egg:MyProject` indicates that this is an entry point *URI* specifier, where the “scheme” is “egg”. An “egg” is created when you run `setup.py install` or `setup.py develop` within your project.

In English, this entry point can thus be referred to as a “PasteDeploy 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”. Indeed, if you open up the `__init__.py` module generated within any scaffold-generated package, you’ll see a `main` function. This is the function called by *PasteDeploy* when the `pserve` command is invoked against our application. It accepts a global configuration object and *returns* an instance of our application.

19.1.2 [DEFAULT] Section of a PasteDeploy .ini File

You can add a [DEFAULT] section to your PasteDeploy .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_config` (see the reference to the `main` function in `__init__.py`).

Command-Line Pyramid

Your Pyramid application can be controlled and inspected using a variety of command-line utilities. These utilities are documented in this chapter.

20.1 Displaying Matching Views for a Given URL

For a big application with several views, it can be hard to keep the view configuration details in your head, even if you defined all the views yourself. You can use the `pviews` command in a terminal window to print a summary of matching routes and views for a given URL in your application. The `pviews` command accepts two arguments. The first argument to `pviews` is the path to your application's `.ini` file and section name inside the `.ini` file which points to your application. This should be of the format `config_file#section_name`. The second argument is the URL to test for matching views. The `section_name` may be omitted; if it is, it's considered to be `main`.

Here is an example for a simple view configuration using *traversal*:

```
1 $ $VENV/bin/pviews development.ini#tutorial /FrontPage
2
3 URL = /FrontPage
4
5     context: <tutorial.models.Page object at 0xa12536c>
6     view name:
7
8     View:
9     ----
10     tutorial.views.view_page
11     required permission = view
```

20. COMMAND-LINE PYRAMID

The output always has the requested URL at the top and below that all the views that matched with their view configuration details. In this example only one view matches, so there is just a single *View* section. For each matching view, the full code path to the associated view callable is shown, along with any permissions and predicates that are part of that view configuration.

A more complex configuration might generate something like this:

```
1 $ $VENV/bin/pviews development.ini#shootout /about
2
3 URL = /about
4
5     context: <shootout.models.RootFactory object at 0xa56668c>
6     view name: about
7
8     Route:
9     -----
10    route name: about
11    route pattern: /about
12    route path: /about
13    subpath:
14    route predicates (request method = GET)
15
16        View:
17        -----
18        shootout.views.about_view
19        required permission = view
20        view predicates (request_param testing, header X/header)
21
22    Route:
23    -----
24    route name: about_post
25    route pattern: /about
26    route path: /about
27    subpath:
28    route predicates (request method = POST)
29
30        View:
31        -----
32        shootout.views.about_view_post
33        required permission = view
34        view predicates (request_param test)
35
36        View:
37        -----
38        shootout.views.about_view_post2
39        required permission = view
```

```
40 | view predicates (request_param test2)
```

In this case, we are dealing with a *URL dispatch* application. This specific URL has two matching routes. The matching route information is displayed first, followed by any views that are associated with that route. As you can see from the second matching route output, a route can be associated with more than one view.

For a URL that doesn't match any views, `pviews` will simply print out a *Not found* message.

20.2 The Interactive Shell

Once you've installed your program for development using `setup.py develop`, you can use an interactive Python shell to execute expressions in a Python environment exactly like the one that will be used when your application runs "for real". To do so, use the `pshell` command line utility.

The argument to `pshell` follows the format `config_file#section_name` where `config_file` is the path to your application's `.ini` file and `section_name` is the app section name inside the `.ini` file which points to your application. For example, if your application `.ini` file might have a `[app:main]` section that looks like so:

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

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

```
$ $VENV/bin/pshell starter/development.ini#main
Python 2.6.5 (r265:79063, Apr 29 2010, 00:31:32)
[GCC 4.4.3] on linux2
Type "help" for more information.

Environment:
  app           The WSGI application.
  registry      Active Pyramid registry.
  request       Active request object.
```

20. COMMAND-LINE PYRAMID

```
root          Root of the default resource tree.
root_factory  Default root factory used to create 'root'.

>>> root
<myproject.resources.MyResource object at 0x445270>
>>> registry
<Registry myproject>
>>> registry.settings['pyramid.debug_notfound']
False
>>> from myproject.views import my_view
>>> from pyramid.request import Request
>>> r = Request.blank('/')
>>> my_view(r)
{'project': 'myproject'}
```

The WSGI application that is loaded will be available in the shell as the `app` global. Also, if the application that is loaded is the Pyramid app with no surrounding *middleware*, the `root` object returned by the default *root factory*, `registry`, and `request` will be available.

You can also simply rely on the `main` default section name by omitting any hash after the filename:

```
$ $VENV/bin/pshell starter/development.ini
```

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

20.2.1 Extending the Shell

It is convenient when using the interactive shell often to have some variables significant to your application already loaded as globals when you start the `pshell`. To facilitate this, `pshell` will look for a special `[pshell]` section in your INI file and expose the subsequent key/value pairs to the shell. Each key is a variable name that will be global within the `pshell` session; each value is a *dotted Python name*. If specified, the special key `setup` should be a *dotted Python name* pointing to a callable that accepts the dictionary of globals that will be loaded into the shell. This allows for some custom initializing code to be executed each time the `pshell` is run. The `setup` callable can also be specified from the commandline using the `--setup` option which will override the key in the INI file.

For example, you want to expose your model to the shell, along with the database session so that you can mutate the model on an actual database. Here, we'll assume your model is stored in the `myapp.models` package.

```

1 [pshell]
2 setup = myapp.lib.pshell.setup
3 m = myapp.models
4 session = myapp.models.DBSession
5 t = transaction

```

By defining the `setup` callable, we will create the module `myapp.lib.pshell` containing a callable named `setup` that will receive the global environment before it is exposed to the shell. Here we mutate the environment's request as well as add a new value containing a WebTest version of the application to which we can easily submit requests.

```

1 # myapp/lib/pshell.py
2 from webtest import TestApp
3
4 def setup(env):
5     env['request'].host = 'www.example.com'
6     env['request'].scheme = 'https'
7     env['testapp'] = TestApp(env['app'])

```

When this INI file is loaded, the extra variables `m`, `session` and `t` will be available for use immediately. Since a `setup` callable was also specified, it is executed and a new variable `testapp` is exposed, and the request is configured to generate urls from the host `http://www.example.com`. For example:

```

$ $VENV/bin/pshell starter/development.ini
Python 2.6.5 (r265:79063, Apr 29 2010, 00:31:32)
[GCC 4.4.3] on linux2
Type "help" for more information.

Environment:
  app           The WSGI application.
  registry      Active Pyramid registry.
  request       Active request object.
  root          Root of the default resource tree.
  root_factory  Default root factory used to create 'root'.
  testapp       <webtest.TestApp object at ...>

Custom Variables:
  m             myapp.models
  session       myapp.models.DBSession
  t             transaction

>>> testapp.get('/')
<200 OK text/html body='<!DOCTYPE...>\n'/3337>

```

```
>>> request.route_url('home')
'https://www.example.com/'
```

20.2.2 IPython or bpython

If you have IPython and/or bpython in the interpreter you use to invoke the `pshell` command, `pshell` will autodiscover and use the first one found, in this order: IPython, bpython, standard Python interpreter. However you could specifically invoke one of your choice with the `-p` choice or `--python-shell` choice option.

```
$ $VENV/bin/pshell -p ipython | bpython | python development.ini#MyProject
```

20.3 Displaying All Application Routes

You can use the `proutes` command in a terminal window to print a summary of routes related to your application. Much like the `pshell` command (see *The Interactive Shell*), the `proutes` command accepts one argument with the format `config_file#section_name`. The `config_file` is the path to your application's `.ini` file, and `section_name` is the app section name inside the `.ini` file which points to your application. By default, the `section_name` is `main` and can be omitted.

For example:

```
1 $ $VENV/bin/proutes development.ini
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>
```

`proutes` generates a table with three columns: *Name*, *Pattern*, and *View*. 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 `proutes` is executed.

20.4 Displaying “Tweens”

A *tween* is a bit of code that sits between the main Pyramid application request handler and the WSGI application which calls it. A user can get a representation of both the implicit tween ordering (the ordering specified by calls to `pyramid.config.Configurator.add_tween()`) and the explicit tween ordering (specified by the `pyramid.tweens` configuration setting) orderings using the `ptweens` command. Tween factories will show up represented by their standard Python dotted name in the `ptweens` output.

For example, here’s the `ptweens` command run against a system configured without any explicit tweens:

```

1 $ $VENV/bin/ptweens development.ini
2 "pyramid.tweens" config value NOT set (implicitly ordered tweens used)
3
4 Implicit Tween Chain
5
6 Position      Name                                     Alias
7 -----      -
8 -             -                                     INGRESS
9 0             pyramid_debugtoolbar.toolbar.toolbar_tween_factory  pdbt
10 1            pyramid.tweens.excview_tween_factory      excview
11 -             -                                     MAIN

```

Here’s the `ptweens` command run against a system configured *with* explicit tweens defined in its `development.ini` file:

```

1 $ ptweens development.ini
2 "pyramid.tweens" config value set (explicitly ordered tweens used)
3
4 Explicit Tween Chain (used)
5
6 Position      Name
7 -----      -
8 -             INGRESS
9 0             starter.tween_factory2
10 1            starter.tween_factory1
11 2            pyramid.tweens.excview_tween_factory
12 -             MAIN
13
14 Implicit Tween Chain (not used)
15
16 Position      Name
17 -----      -
18 -             INGRESS

```

20. COMMAND-LINE PYRAMID

```
19 | 0          pyramid_debugtoolbar.toolbar.toolbar_tween_factory
20 | 1          pyramid.tweens.excview_tween_factory
21 | -          MAIN
```

Here's the application configuration section of the `development.ini` used by the above `ptweens` command which reports that the explicit tween chain is used:

```
1  [app:main]
2  use = egg:starter
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  pyramid.include = pyramid_debugtoolbar
10 pyramid.tweens = starter.tween_factory2
11                  starter.tween_factory1
12                  pyramid.tweens.excview_tween_factory
```

See *Registering Tweens* for more information about tweens.

20.5 Invoking a Request

You can use the `prequest` command-line utility to send a request to your application and see the response body without starting a server.

There are two required arguments to `prequest`:

- The config file/section: follows the format `config_file#section_name` where `config_file` is the path to your application's `.ini` file and `section_name` is the app section name inside the `.ini` file. The `section_name` is optional, it defaults to `main`. For example: `development.ini`.
- The path: this should be the non-url-quoted path element of the URL to the resource you'd like to be rendered on the server. For example, `/`.

For example:

```
$ $VENV/bin/prequest development.ini /
```

This will print the body of the response to the console on which it was invoked.

Several options are supported by `prequest`. These should precede any config file name or URL.

`prequest` has a `-d` (aka `--display-headers`) option which prints the status and headers returned by the server before the output:

```
$ $VENV/bin/prequest -d development.ini /
```

This will print the status, then the headers, then the body of the response to the console.

You can add request header values by using the `--header` option:

```
$ $VENV/bin/prequest --header=Host:example.com development.ini /
```

Headers are added to the WSGI environment by converting them to their CGI/WSGI equivalents (e.g. `Host=example.com` will insert the `HTTP_HOST` header variable as the value `example.com`). Multiple `--header` options can be supplied. The special header value `content-type` sets the `CONTENT_TYPE` in the WSGI environment.

By default, `prequest` sends a GET request. You can change this by using the `-m` (aka `--method`) option. GET, HEAD, POST and DELETE are currently supported. When you use POST, the standard input of the `prequest` process is used as the POST body:

```
$ $VENV/bin/prequest -mPOST development.ini / < somefile
```

20.6 Using Custom Arguments to Python when Running p* Scripts

New in version 1.5.

Each of Pyramid's console scripts (`pserve`, `pviews`, etc) can be run directly using `python -m`, allowing custom arguments to be sent to the python interpreter at runtime. For example:

```
python -3 -m pyramid.scripts.pserve development.ini
```

20.7 Showing All Installed Distributions and their Versions

New in version 1.5.

You can use the `pdistreport` command to show the Pyramid version in use, the Python version in use, and all installed versions of Python distributions in your Python environment:

```
$ $VENV/bin/pdistreport
Pyramid version: 1.5dev
Platform Linux-3.2.0-51-generic-x86_64-with-debian-wheezy-sid
Packages:
  authapp 0.0
    /home/chris/projects/foo/src/authapp
  beautifulsoup4 4.1.3
    /home/chris/projects/foo/lib/python2.7/site-packages/beautifulsoup4-4.1.3-py2.7.egg
... more output ...
```

`pdistreport` takes no options. Its output is useful to paste into a pastebin when you are having problems and need someone with more familiarity with Python packaging and distribution than you have to look at your environment.

20.8 Writing a Script

All web applications are, at their hearts, systems which accept a request and return a response. When a request is accepted by a Pyramid application, the system receives state from the request which is later relied on by your application code. For example, one *view callable* may assume it's working against a request that has a `request.matchdict` of a particular composition, while another assumes a different composition of the `matchdict`.

In the meantime, it's convenient to be able to write a Python script that can work “in a Pyramid environment”, for instance to update database tables used by your Pyramid application. But a “real” Pyramid environment doesn't have a completely static state independent of a request; your application (and Pyramid itself) is almost always reliant on being able to obtain information from a request. When you run

a Python script that simply imports code from your application and tries to run it, there just is no request data, because there isn't any real web request. Therefore some parts of your application and some Pyramid APIs will not work.

For this reason, Pyramid makes it possible to run a script in an environment much like the environment produced when a particular *request* reaches your Pyramid application. This is achieved by using the `pyramid.paster.bootstrap()` command in the body of your script.

New in version 1.1: `pyramid.paster.bootstrap()`

In the simplest case, `pyramid.paster.bootstrap()` can be used with a single argument, which accepts the *PasteDeploy* `.ini` file representing Pyramid your application configuration as a single argument:

```
from pyramid.paster import bootstrap
env = bootstrap('/path/to/my/development.ini')
print(env['request'].route_url('home'))
```

`pyramid.paster.bootstrap()` returns a dictionary containing framework-related information. This dictionary will always contain a *request* object as its *request* key.

The following keys are available in the `env` dictionary returned by `pyramid.paster.bootstrap()`:

request

A `pyramid.request.Request` object implying the current request state for your script.

app

The *WSGI* application object generated by bootstrapping.

root

The *resource* root of your Pyramid application. This is an object generated by the *root factory* configured in your application.

registry

The *application registry* of your Pyramid application.

closer

A parameterless callable that can be used to pop an internal Pyramid thread-local stack (used by `pyramid.threadlocal.get_current_registry()` and `pyramid.threadlocal.get_current_request()`) when your scripting job is finished.

Let's assume that the `/path/to/my/development.ini` file used in the example above looks like so:

```
[pipeline:main]
pipeline = translogger
          another

[filter:translogger]
filter_app_factory = egg:Paste#translogger
setup_console_handler = False
logger_name = wsgi

[app:another]
use = egg:MyProject
```

The configuration loaded by the above bootstrap example will use the configuration implied by the `[pipeline:main]` section of your configuration file by default. Specifying `/path/to/my/development.ini` is logically equivalent to specifying `/path/to/my/development.ini#main`. In this case, we'll be using a configuration that includes an app object which is wrapped in the Paste “translogger” *middleware* (which logs requests to the console).

You can also specify a particular *section* of the PasteDeploy `.ini` file to load instead of `main`:

```
from pyramid.paster import bootstrap
env = bootstrap('/path/to/my/development.ini#another')
print(env['request'].route_url('home'))
```

The above example specifies the `another` app, pipeline, or composite section of your PasteDeploy configuration file. The app object present in the `env` dictionary returned by `pyramid.paster.bootstrap()` will be a Pyramid *router*.

20.8.1 Changing the Request

By default, Pyramid will generate a request object in the `env` dictionary for the URL `http://localhost:80/`. This means that any URLs generated by Pyramid during the execution of your script will be anchored here. This is generally not what you want.

So how do we make Pyramid generate the correct URLs?

Assuming that you have a route configured in your application like so:

```
config.add_route('verify', '/verify/{code}')
```

You need to inform the Pyramid environment that the WSGI application is handling requests from a certain base. For example, we want to simulate mounting our application at *https://example.com/prefix*, to ensure that the generated URLs are correct for our deployment. This can be done by either mutating the resulting request object, or more simply by constructing the desired request and passing it into `bootstrap()`:

```
from pyramid.paster import bootstrap
from pyramid.request import Request

request = Request.blank('/', base_url='https://example.com/prefix')
env = bootstrap('/path/to/my/development.ini#another', request=request)
print(env['request'].application_url)
# will print 'https://example.com/prefix'
```

Now you can readily use Pyramid's APIs for generating URLs:

```
env['request'].route_url('verify', code='1337')
# will return 'https://example.com/prefix/verify/1337'
```

20.8.2 Cleanup

When your scripting logic finishes, it's good manners to call the `closer` callback:

```
from pyramid.paster import bootstrap
env = bootstrap('/path/to/my/development.ini')

# .. do stuff ...

env['closer']()
```

20.8.3 Setting Up Logging

By default, `pyramid.paster.bootstrap()` does not configure logging parameters present in the configuration file. If you'd like to configure logging based on `[logger]` and related sections in the configuration file, use the following command:

```
import pyramid.paster
pyramid.paster.setup_logging('/path/to/my/development.ini')
```

See *Logging* for more information on logging within Pyramid.

20.9 Making Your Script into a Console Script

A “console script” is *setuptools* terminology for a script that gets installed into the `bin` directory of a Python *virtualenv* (or “base” Python environment) when a *distribution* which houses that script is installed. Because it’s installed into the `bin` directory of a *virtualenv* when the distribution is installed, it’s a convenient way to package and distribute functionality that you can call from the command-line. It’s often more convenient to create a console script than it is to create a `.py` script and instruct people to call it with the “right” Python interpreter. A console script generates a file that lives in `bin`, and when it’s invoked it will always use the “right” Python environment, which means it will always be invoked in an environment where all the libraries it needs (such as Pyramid) are available.

In general, you can make your script into a console script by doing the following:

- Use an existing distribution (such as one you’ve already created via `pcreate`) or create a new distribution that possesses at least one package or module. It should, within any module within the distribution, house a callable (usually a function) that takes no arguments and which runs any of the code you wish to run.
- Add a `[console_scripts]` section to the `entry_points` argument of the distribution which creates a mapping between a script name and a dotted name representing the callable you added to your distribution.
- Run `setup.py develop`, `setup.py install`, or `easy_install` to get your distribution reinstalled. When you reinstall your distribution, a file representing the script that you named in the last step will be in the `bin` directory of the *virtualenv* in which you installed the distribution. It will be executable. Invoking it from a terminal will execute your callable.

As an example, let’s create some code that can be invoked by a console script that prints the deployment settings of a Pyramid application. To do so, we’ll pretend you have a distribution with a package in it named `myproject`. Within this package, we’ll pretend you’ve added a `scripts.py` module which contains the following code:

```

1  # myproject.scripts module
2
3  import optparse
4  import sys
5  import textwrap
6
7  from pyramid.paster import bootstrap
8
9  def settings_show():
10     description = """\
11     Print the deployment settings for a Pyramid application.  Example:
12     'show_settings deployment.ini'
13     """
14     usage = "usage: %prog config_uri"
15     parser = optparse.OptionParser(
16         usage=usage,
17         description=textwrap.dedent(description)
18     )
19     parser.add_option(
20         '-o', '--omit',
21         dest='omit',
22         metavar='PREFIX',
23         type='string',
24         action='append',
25         help=("Omit settings which start with PREFIX (you can use this "
26              "option multiple times)")
27     )
28
29     options, args = parser.parse_args(sys.argv[1:])
30     if not len(args) >= 1:
31         print('You must provide at least one argument')
32         return 2
33     config_uri = args[0]
34     omit = options.omit
35     if omit is None:
36         omit = []
37     env = bootstrap(config_uri)
38     settings, closer = env['registry'].settings, env['closer']
39     try:
40         for k, v in settings.items():
41             if any([k.startswith(x) for x in omit]):
42                 continue
43             print('%-40s    %-20s' % (k, v))
44     finally:
45         closer()

```

20. COMMAND-LINE PYRAMID

This script uses the Python `optparse` module to allow us to make sense out of extra arguments passed to the script. It uses the `pyramid.paster.bootstrap()` function to get information about the application defined by a config file, and prints the deployment settings defined in that config file.

After adding this script to the package, you'll need to tell your distribution's `setup.py` about its existence. Within your distribution's top-level directory your `setup.py` file will look something like this:

```
1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 with open(os.path.join(here, 'README.txt')) as f:
7     README = f.read()
8 with open(os.path.join(here, 'CHANGES.txt')) as f:
9     CHANGES = f.read()
10
11 requires = ['pyramid', 'pyramid_debugtoolbar']
12
13 setup(name='MyProject',
14       version='0.0',
15       description='My project',
16       long_description=README + '\n\n' + CHANGES,
17       classifiers=[
18         "Programming Language :: Python",
19         "Framework :: Pylons",
20         "Topic :: Internet :: WWW/HTTP",
21         "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
22     ],
23     author='',
24     author_email='',
25     url='',
26     keywords='web pyramid pylons',
27     packages=find_packages(),
28     include_package_data=True,
29     zip_safe=False,
30     install_requires=requires,
31     tests_require=requires,
32     test_suite="myproject",
33     entry_points = """\
34     [paste.app_factory]
35     main = myproject:main
36     """,
37 )
```

We're going to change the `setup.py` file to add an `[console_scripts]` section with in

the `entry_points` string. Within this section, you should specify a `scriptname = dotted.path.to:yourfunction` line. For example:

```
[console_scripts]
show_settings = myproject.scripts:settings_show
```

The `show_settings` name will be the name of the script that is installed into `bin`. The colon (`:`) between `myproject.scripts` and `settings_show` above indicates that `myproject.scripts` is a Python module, and `settings_show` is the function in that module which contains the code you'd like to run as the result of someone invoking the `show_settings` script from their command line.

The result will be something like:

```
1  import os
2
3  from setuptools import setup, find_packages
4
5  here = os.path.abspath(os.path.dirname(__file__))
6  with open(os.path.join(here, 'README.txt')) as f:
7      README = f.read()
8  with open(os.path.join(here, 'CHANGES.txt')) as f:
9      CHANGES = f.read()
10
11  requires = ['pyramid', 'pyramid_debugtoolbar']
12
13  setup(name='MyProject',
14        version='0.0',
15        description='My project',
16        long_description=README + '\n\n' + CHANGES,
17        classifiers=[
18            "Programming Language :: Python",
19            "Framework :: Pylons",
20            "Topic :: Internet :: WWW/HTTP",
21            "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
22        ],
23        author='',
24        author_email='',
25        url='',
26        keywords='web pyramid pylons',
27        packages=find_packages(),
28        include_package_data=True,
29        zip_safe=False,
30        install_requires=requires,
31        tests_require=requires,
32        test_suite="myproject",
```

20. COMMAND-LINE PYRAMID

```
33     entry_points = """\
34     [paste.app_factory]
35     main = myproject:main
36     [console_scripts]
37     show_settings = myproject.scripts:settings_show
38     """,
39 )
```

Once you’ve done this, invoking `$$VENV/bin/python setup.py develop` will install a file named `show_settings` into the `$somevirtualenv/bin` directory with a small bit of Python code that points to your entry point. It will be executable. Running it without any arguments will print an error and exit. Running it with a single argument that is the path of a config file will print the settings. Running it with an `--omit=foo` argument will omit the settings that have keys that start with `foo`. Running it with two “omit” options (e.g. `--omit=foo --omit=bar`) will omit all settings that have keys that start with either `foo` or `bar`:

```
$ $VENV/bin/show_settings development.ini --omit=pyramid --omit=debugtoolbar
debug_routematch                False
debug_templates                 True
reload_templates                True
mako.directories                []
debug_notfound                  False
default_locale_name             en
reload_resources                 False
debug_authorization             False
reload_assets                   False
prevent_http_cache              False
```

Pyramid’s `pserve`, `pcreate`, `pshell`, `prequest`, `ptweens` and other `p*` scripts are implemented as console scripts. When you invoke one of those, you are using a console script.

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.

21.1 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.

21.1.1 Using The `TranslationString` Class

The most primitive way to create a translation string is to use the `pyramid.i18n.TranslationString` callable:

21. 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.

21.1.2 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', default='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:

21. INTERNATIONALIZATION AND LOCALIZATION

```
1 from pyramid.i18n import TranslationString as _
2 ts = _('add-number', default='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', default='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.

21.2 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 start faster.

The tools for working with *gettext* translation files related to a Pyramid application are *Lingua* and *Gettext*. *Lingua* can scrape i18n references out of Python and Chameleon files and create the `.pot` file. *Gettext* includes `msgmerge` tool to update a `.po` file from an updated `.pot` file and `msgfmt` to compile `.po` files to `.mo` files.

21.2.1 Installing Lingua and Gettext

In order for the commands related to working with *gettext* translation files to work properly, you will need to have *Lingua* and *Gettext* installed into the same environment in which Pyramid is installed.

Installation on UNIX

Gettext is often already installed on UNIX systems. You can check if it is installed by testing if the `msgfmt` command is available. If it is not available you can install it through the packaging system from your OS; the package name is almost always `gettext`. For example on a Debian or Ubuntu system run this command:

```
$ sudo apt-get install gettext
```

Installing *Lingua* is done with the Python packaging tools. If the *virtualenv* into which you've installed your Pyramid application lives in `/my/virtualenv`, you can install *Lingua* like so:

21. INTERNATIONALIZATION AND LOCALIZATION

```
$ cd /my/virtualenv
$ $VENV/bin/easy_install lingua
```

Installation on Windows

There are several ways to install Gettext on Windows: it is included in the Cygwin collection, or you can use the installer from the GnuWin32 or compile it yourself. Make sure the installation path is added to your `$PATH`.

Installing Lingua is done with the Python packaging tools. If the *virtualenv* into which you've installed your Pyramid application lives in `C:\my\virtualenv`, you can install Lingua like so:

```
C> %VENV%\Scripts\easy_install lingua
```

21.2.2 Extracting Messages from Code and Templates

Once Lingua is installed you may extract a message catalog template from the code and *Chameleon* templates which reside in your Pyramid application. You run a `pot-create` command to extract the messages:

```
$ cd /place/where/myapplication/setup.py/lives
$ mkdir -p myapplication/locale
$ $VENV/bin/pot-create -o myapplication/locale/myapplication.pot src
```

The message catalog `.pot` template will end up in:

`myapplication/locale/myapplication.pot`.

21.2.3 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 `msginit` command from Gettext:

```
$ cd /place/where/myapplication/setup.py/lives
$ cd myapplication/locale
$ mkdir -p es/LC_MESSAGES
$ msginit -l es -o es/LC_MESSAGES/myapplication.po
```

This will create a new the message catalog `.po` file will 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*.

21.2.4 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 `msgmerge` command from Gettext.

```
$ cd /place/where/myapplication/setup.py/lives
$ cd myapplication/locale
$ msgmerge --update es/LC_MESSAGES/myapplication.po myapplication.pot
```

21.2.5 Compiling a Message Catalog File

Finally, to prepare an application for performing actual runtime translations, compile `.po` files to `.mo` files use the `msgfmt` command from Gettext:

```
$ cd /place/where/myapplication/setup.py/lives
$ msgfmt myapplication/locale/*/LC_MESSAGES/*.po
```

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.

21.3 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.request.Request.localizer` attribute to obtain a *localizer*. The localizer object will be configured to produce translations implied by the active *locale negotiator* or a default localizer object if no explicit locale negotiator is registered.

```
1 def aview(request):
2     localizer = request.localizer
```



If you need to create a localizer for a locale use the `pyramid.i18n.make_localizer()` function.

21.3.1 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 TranslationString
2
3 ts = TranslationString('Add ${number}', mapping={'number':1},
4                       domain='pyramid')
5
6 def aview(request):
7     localizer = request.localizer
8     translated = localizer.translate(ts) # translation string
9     # ... use translated ...
```

The `request.localizer` attribute will be 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*.

21.3.2 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 simplest case is the `singular` and `plural` arguments being passed as unicode literals. This returns the appropriate literal according to the locale pluralization rules for the number `n`, and interpolates `mapping`.

```
1 def aview(request):
2     localizer = request.localizer
3     translated = localizer.pluralize('Item', 'Items', 1, 'mydomain')
4     # ... use translated ...
```

However, for support of other languages, the `singular` argument should be a Unicode value representing a *message identifier*. In this case the `plural` value is ignored. `domain` should be a *translation domain*, and `mapping` should be a dictionary that is used for *replacement value* interpolation of the translated string.

The value of `n` will be used to find the appropriate plural form for the current language and `pluralize` will return a Unicode translation for the message id `singular`. The message file must have defined `singular` as a translation with plural forms.

The argument provided as `singular` may be a *translation string* object, but the `domain` and `mapping` information attached is ignored.

```
1 def aview(request):
2     localizer = request.localizer
3     num = 1
4     translated = localizer.pluralize('item_plural', '${number} items',
5                                     num, 'mydomain', mapping={'number':num})
```

The corresponding message catalog must have language plural definitions and plural alternatives set.

```
1 "Plural-Forms: nplurals=3; plural=n==0 ? 0 : n==1 ? 1 : 2;"
2
3 msgid "item_plural"
4 msgid_plural ""
5 msgstr[0] "No items"
6 msgstr[1] "${number} item"
7 msgstr[2] "${number} items"
```

More information on complex plurals can be found in the gettext documentation.

21.4 Obtaining the Locale Name for a Request

You can obtain the locale name related to a request by using the `pyramid.request.Request.locale_name()` attribute of the request.

```
1 def aview(request):
2     locale_name = request.locale_name
```

The locale name of a request is dynamically computed; it will be 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 `pyramid.default_locale_name` setting; see *Default Locale Name*.

Once `locale_name()` is first run, the locale name is stored on the request object. Subsequent calls to `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 def aview(request):
2     localizer = request.localizer
3     locale_name = localizer.locale_name
```

Obtaining the locale name as an attribute of a localizer is equivalent to obtaining a locale name by asking for the `locale_name()` attribute.

21.5 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 minimal information about how to perform date and currency related locale operations. See *Installing Lingua and Gettext* 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
3 def aview(request):
4     locale_name = request.locale_name
5     locale = Locale(locale_name)
```

21.6 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 `tal` namespace of Chameleon will also consult the Pyramid translations. See <http://chameleon.readthedocs.org/en/latest/reference.html#id50>.



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*.

21.7 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.

21.8 Localization-Related Deployment Settings

A Pyramid application will have a `pyramid.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={'pyramid.default_locale_name': 'de'})
```

You may alternately supply a `pyramid.default_locale_name` via an application’s `.ini` file:

```
1 [app:main]
2 use = egg:MyProject
3 pyramid.reload_templates = true
4 pyramid.debug_authorization = false
5 pyramid.debug_notfound = false
6 pyramid.default_locale_name = de
```

If this value is not supplied via the `Configurator` constructor or via a 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['pyramid.default_locale_name']
```

21.9 “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 `.ini` file:

```
1 [app:main]
2 use = egg:MyProject
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.

21.10 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.

21.10.1 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*.

21.10.2 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*.

21.11 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 `locale_name()` is accessed or when `negotiate_locale_name()` is invoked.

21.11.1 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`.

21.11.2 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:

21. INTERNATIONALIZATION AND LOCALIZATION

```
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:

```
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.

22.1 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 can be 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
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/chrism/projects/modwsgi/env/pyramid.wsgi
```

In the above configuration, we root a Pyramid application at `/pyramidapp` within the Apache configuration.

22.2 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.request.Request.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*.

22.3 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.

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 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` API 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 Test setup using a context manager

An alternative style of setting up a test configuration is to use the *with* statement and `pyramid.testing.testConfig()` to create a context manager. The context manager will call `pyramid.testing.setUp()` before the code under test and `pyramid.testing.tearDown()` afterwards.

This style is useful for small self-contained tests. For example:

```
1 import unittest
2
3 class MyTest(unittest.TestCase):
4
5     def test_my_function(self):
6         from pyramid import testing
7         with testing.testConfig() as config:
8             config.add_route('bar', '/bar/{id}')
9             my_function_which_needs_route_bar()
```

23.1.2 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.httpexceptions import HTTPForbidden
2
3 def view_fn(request):
4     if request.has_permission('edit'):
5         raise HTTPForbidden
6     return {'greeting': 'hello'}

```



This code implies that you have defined a renderer imperatively in a relevant `pyramid.config.Configurator` instance, otherwise it would fail when run normally.

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 `HTTPForbidden`?

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.httpexceptions import HTTPForbidden
13         from my.package import view_fn
14         self.config.testing_securitypolicy(userid='hank',
15                                           permissive=False)
16         request = testing.DummyRequest()
17         request.context = testing.DummyResource()
18         self.assertRaises(HTTPForbidden, view_fn, request)
19

```

```
20 def test_view_fn_allowed(self):
21     from my.package import view_fn
22     self.config.testing_securitypolicy(userid='hank',
23                                       permissive=True)
24     request = testing.DummyRequest()
25     request.context = testing.DummyResource()
26     response = view_fn(request)
27     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.request.Request.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 `HTTPForbidden` 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:

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 its `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.assertTrue('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 a 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.assertTrue('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.app.TestApp` instance.

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).

24.1 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 the ‘c’ leaf resource 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.

24.2 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).

For your convenience

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.

24.3 Generating The URL Of A Resource

If your resources are *location* aware, you can use the `pyramid.request.Request.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 url = request.resource_url(resource)
```

The `request` 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()`:

```
url = request.resource_url(resource, '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:

```
url = request.resource_url(resource, 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.

For more information about generating resource URLs, see the documentation for `pyramid.request.Request.resource_url()`.

24.3.1 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 the following keys:

physical_path A string representing the “physical path” computed for the resource, as defined by `pyramid.traversal.resource_path(resource)`. It will begin and end with a slash.

virtual_path A string representing 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. It will begin and end with a slash.

app_url A string representing the application URL generated during `request.resource_url`. It will not end with a slash. It represents a potentially customized URL prefix, containing potentially custom scheme, host and port information passed by the user to `request.resource_url`. It should be preferred over use of `request.application_url`.

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 info['app_url'] + info['virtual_path']
```

The above example actually just generates and returns the default URL, which would have been what was generated by the default `resource_url` machinery, 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 with `resource_url()`.

24.4 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()`.

24.5 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.

24.6 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:

```
1 list(lineage(thing2))
2 [ <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 has a `__parent__` attribute of `None`.

See the documentation for `pyramid.location.lineage()` for more information.

24.7 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.

24.8 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.

24.9 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*.

```
1 import datetime
2 from zope.interface import implementer
3 from zope.interface import Interface
4
5 class IBlogEntry(Interface):
6     pass
7
8 @implementer(IBlogEntry)
9 class BlogEntry(object):
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 `implementer` class decorator 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.implementer()` function as a class decorator. 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 import datetime
2 from zope.interface import directlyProvides
3 from zope.interface import Interface
4
5 class IBlogEntry(Interface):
6     pass
7
8 class BlogEntry(object):
9     def __init__(self, title, body, author):
10         self.title = title
11         self.body = body
12         self.author = author
13         self.created = datetime.datetime.now()
```

```
14
15 entry = BlogEntry('title', 'body', 'author')
16 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  import datetime
2  from zope.interface import alsoProvides
3  from zope.interface import directlyProvides
4  from zope.interface import Interface
5
6  class IBlogEntry1(Interface):
7      pass
8
9  class IBlogEntry2(Interface):
10     pass
11
12  class BlogEntry(object):
13     def __init__(self, title, body, author):
14         self.title = title
15         self.body = body
16         self.author = author
17         self.created = datetime.datetime.now()
18
19  entry = BlogEntry('title', 'body', 'author')
20  directlyProvides(entry, IBlogEntry1)
21  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*.

24.10 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:

```
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:

See also *Resources Which Implement Interfaces*.

24.11 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 on the `pyramid.request.Request` 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 security related APIs on the `pyramid.request.Request` class also accept *context* as an argument, and a context is always a resource.

Hello Traversal World

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

If code speaks louder than words, maybe this will help. Here is a single-file Pyramid application that uses traversal:

```
1 from wsgiref.simple_server import make_server
2 from pyramid.config import Configurator
3 from pyramid.response import Response
4
5 class Resource(dict):
6     pass
7
8 def get_root(request):
9     return Resource({'a': Resource({'b': Resource({'c': Resource()})})})
10
11 def hello_world_of_resources(context, request):
12     output = "Here's a resource and its children: %s" % context
13     return Response(output)
14
15 if __name__ == '__main__':
16     config = Configurator(root_factory=get_root)
17     config.add_view(hello_world_of_resources, context=Resource)
18     app = config.make_wsgi_app()
19     server = make_server('0.0.0.0', 8080, app)
20     server.serve_forever()
```

25. HELLO TRAVERSAL WORLD

You may notice that this application is intentionally very similar to the “hello world” app from *Creating Your First Pyramid Application*.

On lines 5-6, we create a trivial *resource* class that’s just a dictionary subclass.

On lines 8-9, we hard-code a *resource tree* in our *root factory* function.

On lines 11-13 we define a single *view callable* that can display a single instance of our Resource class, passed as the `context` argument.

The rest of the file sets up and serves our pyramid WSGI app. Line 18 is where our view gets configured for use whenever the traversal ends with an instance of our Resource class.

Interestingly, there are no URLs explicitly configured in this application. Instead, the URL space is defined entirely by the keys in the resource tree.

25.1 Example requests

If this example is running on `http://localhost:8080`, and the user browses to `http://localhost:8080/a/b`, Pyramid will call `get_root(request)` to get the root resource, then traverse the tree from there by key; starting from the root, it will find the child with key "a", then its child with key "b"; then use that as the `context` argument for calling `hello_world_of_resources`.

Or, if the user browses to `http://localhost:8080/`, Pyramid will stop at the root - the outermost Resource instance, in this case - and use that as the `context` argument to the same view.

Or, if the user browses to a key that doesn’t exist in this resource tree, like `http://localhost:8080/xyz` or `http://localhost:8080/a/b/c/d`, the traversal will end by raising a `KeyError`, and Pyramid will turn that into a 404 HTTP response.

A more complicated application could have many types of resources, with different view callables defined for each type, and even multiple views for each type.

See also:

Full technical details may be found in *Traversal*.

For more about *why* you might use traversal, see *Much Ado About Traversal*.

Much Ado About Traversal

(Or, why you should care about it)



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 who 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.

26.1 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/ {photo1d}`, 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.

26.2 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.

26.3 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.)

26.4 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:

```
>>> adict = {}
>>> adict['a'] = 1
>>> adict['a']
1
>>> adict.__getitem__('a')
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.

26.5 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.

26.6 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

This chapter explains the technical details of how traversal works in Pyramid.

For a quick example, see *Hello Traversal World*.

For more about *why* you might use traversal, see *Much Ado About 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 to “publish” this resource by generating a *response*.

 Using *Traversal* to map a URL to code is optional. If you’re creating your first Pyramid application it probably makes more sense to use *URL dispatch* to map URLs to code instead of traversal, as new Pyramid developers tend to find URL dispatch slightly easier to understand. If you use URL dispatch, you needn’t read this chapter.

27.1 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.

27.2 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. The above `Root` class is analogous to the default root factory present in Pyramid. The default root factory is very simple and not very useful.



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.

27.3 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.

27.3.1 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 in 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 "B" 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 `AttributeError`) 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. 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:

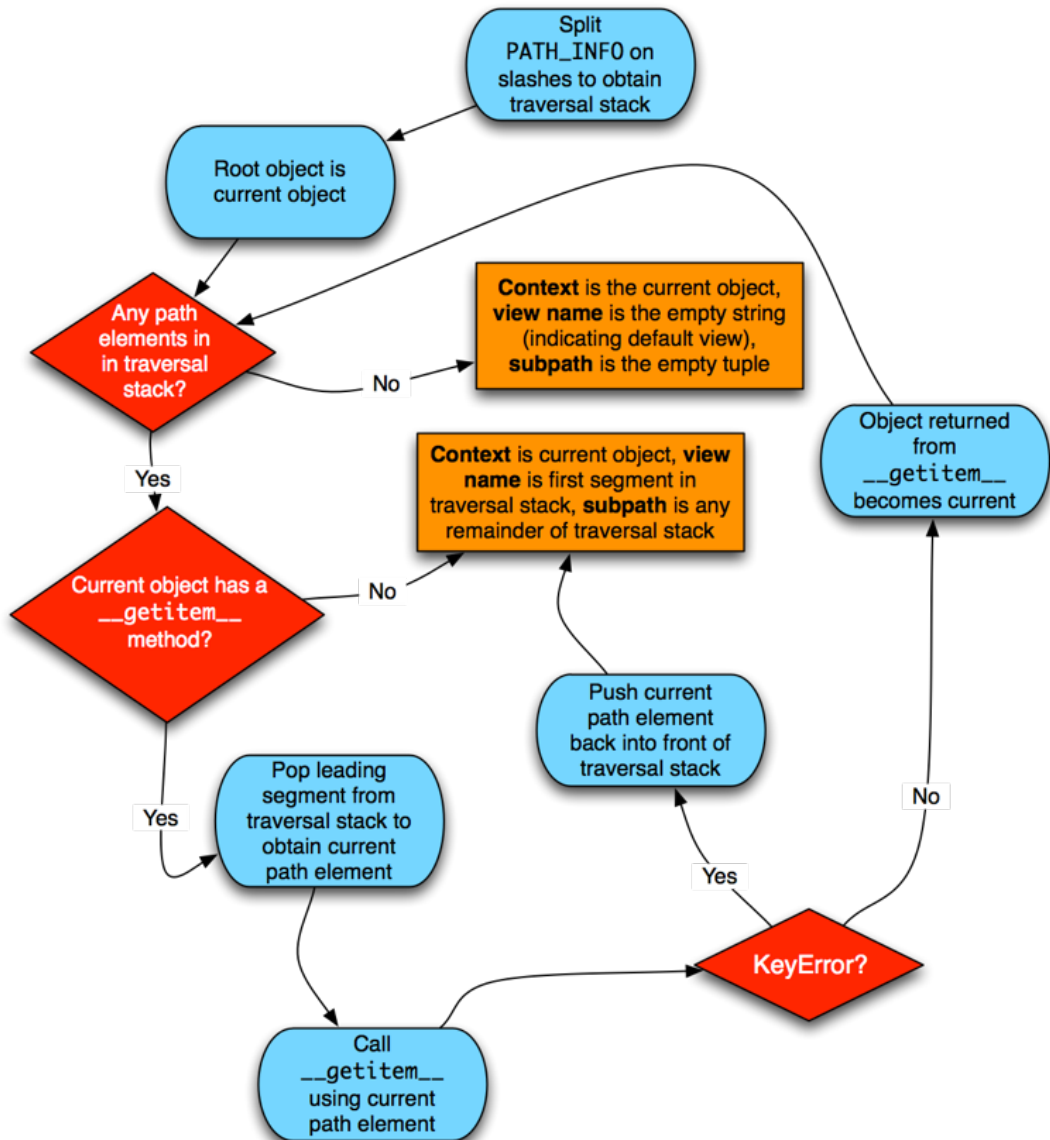
- You will often end up with a *view name* that is the empty string as the result of a particular traversal. This indicates that the view lookup machinery should look up the *default view*. The default view is a view that is registered with no name or a view which is registered with a name that equals 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



27.3.2 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:

```

/--
 |
 |-- foo
    |
    ----bar
        |
        ----baz
            |
            biz

```

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*.

27.3.3 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.implementer()` class decorator to associate the interface with the class.

```
1 from zope.interface import Interface
2 from zope.interface import implementer
3
4 class IHello(Interface):
5     """ A marker interface """
6
7 @implementer(IHello)
8 class Hello(object):
9     pass
```

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*.

27.4 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.request.Request.resource_url()` method generates a URL when given a resource retrieved from a resource tree.

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.

28.1 Enabling an Authorization Policy

Pyramid does not enable any authorization policy by default. 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.

28.1.1 Enabling an Authorization Policy Imperatively

Use the `set_authorization_policy()` method of the `Configurator` to enable 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 `set_authentication_policy()` method during application setup to specify the authentication policy.

For example:

```
1 from pyramid.config import Configurator
2 from pyramid.authentication import AuthTktAuthenticationPolicy
3 from pyramid.authorization import ACLAuthorizationPolicy
4 authn_policy = AuthTktAuthenticationPolicy('seekrit', hashalg='sha512')
5 authz_policy = ACLAuthorizationPolicy()
6 config = Configurator()
7 config.set_authentication_policy(authn_policy)
8 config.set_authorization_policy(authz_policy)
```



The `authentication_policy` and `authorization_policy` arguments may also be passed to their respective methods mentioned above 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 configure a Pyramid application with an authentication policy but without the authorization policy or vice versa. If you do this, you’ll receive an error at application startup time.

See also:

See also the `pyramid.authorization` and `pyramid.authentication` modules for alternate implementations of authorization and authentication policies.

28.2 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.

28.2.1 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.

The `pyramid.config.Configurator.set_default_permission()` method supports configuring a default permission for an application.

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 the permission `pyramid.security.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 `pyramid.security.NO_PERMISSION_REQUIRED` permission.

28.3 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 Allow
2 from pyramid.security import Everyone
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 Allow
2 from pyramid.security import Everyone
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.

Dynamic ACLs are also possible by turning the ACL into a callable on the resource. This may allow the ACL to dynamically generate rules based on properties of the instance.

```
1 from pyramid.security import Allow
2 from pyramid.security import Everyone
3
4 class Blog(object):
5     def __acl__(self):
6         return [
7             (Allow, Everyone, 'view'),
8             (Allow, self.owner, 'edit'),
9             (Allow, 'group:editors', 'edit'),
10        ]
11
12     def __init__(self, owner):
13         self.owner = owner
```

28.4 Elements of an ACL

Here’s an example ACL:

```
1 from pyramid.security import Allow
2 from pyramid.security import Everyone
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')`.

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* 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 Allow
2 from pyramid.security import Deny
3 from pyramid.security import Everyone
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 Allow
2 from pyramid.security import Everyone
3
4 __acl__ = [
5     (Allow, Everyone, 'view'),
6     (Allow, 'group:editors', ('add', 'edit')),
7 ]
```

28.5 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`).

28.6 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.

28.7 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) ]
```

28.8 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 also:

See also *pyramid.location* for documentations of functions which use location-awareness.

See also:

See also *Location-Aware Resources*.

28.9 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.

28.10 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 $VENV/bin/pserve 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 `pyramid.debug_authorization` key to `true` within the application's configuration section, e.g.:

```
1 [app:main]
2 use = egg:MyProject
3 pyramid.debug_authorization = true
```

With this debug flag turned on, the response sent to the browser will also contain security debugging information in its body.

28.11 Debugging Imperative Authorization Failures

The `pyramid.request.Request.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.

28.12 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 IAuthenticationPolicy(object):
2     """ An object representing a Pyramid authentication policy. """
3
4     def authenticated_userid(self, request):
5         """ Return the authenticated userid or ``None`` if no
6             authenticated userid can be found. This method of the policy
7             should ensure that a record exists in whatever persistent store is
8             used related to the user (the user should not have been deleted);
9             if a record associated with the current id does not exist in a
10             persistent store, it should return ``None``. """
11
```

```
12     def unauthenticated_userid(self, request):
13         """ Return the *unauthenticated* userid. This method performs the
14         same duty as ``authenticated_userid`` but is permitted to return the
15         userid based only on data present in the request; it needn't (and
16         shouldn't) check any persistent store to ensure that the user record
17         related to the request userid exists. """
18
19     def effective_principals(self, request):
20         """ Return a sequence representing the effective principals
21         including the userid and any groups belonged to by the current
22         user, including 'system' groups such as
23         'pyramid.security.Everyone' and
24         'pyramid.security.Authenticated'. """
25
26     def remember(self, request, principal, **kw):
27         """ Return a set of headers suitable for 'remembering' the
28         principal named 'principal' when set in a response. An
29         individual authentication policy and its consumers can decide
30         on the composition and meaning of **kw. """
31
32     def forget(self, request):
33         """ Return a set of headers suitable for 'forgetting' the
34         current user on subsequent requests. """
```

After you do so, you can pass an instance of such a class into the `set_authentication_policy` method configuration time to use it.

28.13 Creating Your Own Authorization Policy

An authorization policy is a policy that allows or denies access after a user has been authenticated. Most Pyramid applications will use the default `pyramid.authorization.ACLAuthorizationPolicy`.

However, 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 `set_authorization_policy` method at configuration time to use it.

28.14 Admonishment Against Secret-Sharing

A “secret” is required by various components of Pyramid. For example, the *authentication policy* below uses a secret value `seekrit`:

```
authn_policy = AuthTktAuthenticationPolicy('seekrit', hashalg='sha512')
```

A *session factory* also requires a secret:

```
my_session_factory = SignedCookieSessionFactory('itsaseekreet')
```

It is tempting to use the same secret for multiple Pyramid subsystems. For example, you might be tempted to use the value `seekrit` as the secret for both the authentication policy and the session factory defined above. This is a bad idea, because in both cases, these secrets are used to sign the payload of the data.

If you use the same secret for two different parts of your application for signing purposes, it may allow an attacker to get his chosen plaintext signed, which would allow the attacker to control the content of the payload. Re-using a secret across two different subsystems might drop the security of signing to zero. Keys should not be re-used across different contexts where an attacker has the possibility of providing a chosen plaintext.

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.

29.1 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.

29.1.1 URL Dispatch Only

An application that uses *url dispatch* exclusively to map URLs to code will often have statements like this within application startup configuration:

29. COMBINING TRAVERSAL AND URL DISPATCH

```
1 # config is an instance of pyramid.config.Configurator
2
3 config.add_route('foobar', '{foo}/{bar}')
4 config.add_route('bazbuz', '{baz}/{buz}')
5
6 config.add_view('myproject.views.foobar', route_name='foobar')
7 config.add_view('myproject.views.bazbuz', route_name='bazbuz')
```

Each *route* corresponds to one or more view callables. Each view callable is associated with a route by passing a `route_name` parameter that matches its name during a call to `add_view()`. When a route is matched during a request, *view lookup* is used to match the request to its associated view callable. The presence of calls to `add_route()` signify that an application is using URL dispatch.

29.1.2 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.

29.2 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`.

29.2.1 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*.

29.2.2 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*.

A hybrid mode application relies more heavily on *traversal* to do *resource location* and *view lookup* than most examples indicate within *URL Dispatch*.

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 = Resource(
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 `Resource` 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` function 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.

In our above example, this particular set of traversal steps will mean that the *context* resource of the view would be the `Resource` 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', route_name='home')
4 config.add_view('mypackage.views.another_view', route_name='home',
5                  name='another')

```

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).

29.2.3 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='{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')
```

29.2.4 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.static.static_view` helper attempt to compute `PATH_INFO` from the request's subpath when its `use_subpath` argument is `True`, 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 from pyramid.static import static_view
2
3 www = static_view('mypackage:static', use_subpath=True)
4
5 config.add_route('static', '/static/*subpath')
6 config.add_view(www, route_name='static')

```

`mypackage.views.www` is an instance of `pyramid.static.static_view`. This effectively tells the static helper to traverse everything in the subpath as a filename.

29.3 Corner Cases

A number of corner case “gotchas” exist when using a hybrid application. We’ll detail them here.

29.3.1 Registering a Default View for a Route That Has a `view` Attribute



As of Pyramid 1.1 this section is slated to be removed in a later documentation release because the ability to add views directly to the *route configuration* by passing a `view` argument to `add_route` has been deprecated.

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:

29. COMBINING TRAVERSAL AND URL DISPATCH

```
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.

29.3.2 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.bazbuz', name='bazbuz',
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.bazbuz', name='bazbuz',
3               route_name='abc')
```

With the above configuration, the `myproject.views.bazbuz` view will be invoked when the request URI is `/abc/bazbuz`, assuming there is no object contained by the root object with the key `bazbuz`. A different request URI, such as `/abc/foo/bar`, would invoke the default `myproject.views.abc` view.

29.4 Generating Hybrid URLs

New in version 1.5.

The `pyramid.request.Request.resource_url()` method and the `pyramid.request.Request.resource_path()` method both accept optional keyword

arguments that make it easier to generate route-prefixed URLs that contain paths to traversal resources: `route_name`, `route_kw`, and `route_remainder_name`.

Any route that has a pattern that contains a `*remainder` pattern (any stararg remainder pattern, such as `*traverse` or `*subpath` or `*fred`) can be used as the target name for `request.resource_url(..., route_name=)` and `request.resource_path(..., route_name=)`.

For example, let's imagine you have a route defined in your Pyramid application like so:

```
config.add_route('mysection', '/mysection*traverse')
```

If you'd like to generate the URL `http://example.com/mysection/a/`, you can use the following incantation, assuming that the variable `a` below points to a resource that is a child of the root with a `__name__` of `a`:

```
request.resource_url(a, route_name='mysection')
```

You can generate only the path portion `/mysection/a/` assuming the same:

```
request.resource_path(a, route_name='mysection')
```

The path is virtual host aware, so if the `X-Vhm-Root` environ variable is present in the request, and it's set to `/a`, the above call to `request.resource_url` would generate `http://example.com/mysection/` and the above call to `request.resource_path` would generate `/mysection/`. See *Virtual Root Support* for more information.

If the route you're trying to use needs simple dynamic part values to be filled in to successfully generate the URL, you can pass these as the `route_kw` argument to `resource_url` and `resource_path`. For example, assuming that the route definition is like so:

```
config.add_route('mysection', '/{id}/mysection*traverse')
```

You can pass `route_kw` in to fill in `{id}` above:

```
request.resource_url(a, route_name='mysection', route_kw={'id': '1'})
```

If you pass `route_kw` but do not pass `route_name`, `route_kw` will be ignored.

By default this feature works by calling `route_url` under the hood, and passing the value of the resource path to that function as `traverse`. If your route has a different `*stararg` remainder name (such as `*subpath`), you can tell `resource_url` or `resource_path` to use that instead of `traverse` by passing `route_remainder_name`. For example, if you have the following route:

29. COMBINING TRAVERSAL AND URL DISPATCH

```
config.add_route('mysection', '/mysection*subpath')
```

You can fill in the `*subpath` value using `resource_url` by doing:

```
request.resource_path(a, route_name='mysection',  
                      route_remainder_name='subpath')
```

If you pass `route_remainder_name` but do not pass `route_name`, `route_remainder_name` will be ignored.

If you try to use `resource_path` or `resource_url` when the `route_name` argument points at a route that does not have a remainder stararg, an error will not be raised, but the generated URL will not contain any remainder information either.

All other values that are normally passable to `resource_path` and `resource_url` (such as `query`, `anchor`, `host`, `port`, and positional elements) work as you might expect in this configuration.

Note that this feature is incompatible with the `__resource_url__` feature (see *Overriding Resource URL Generation*) implemented on resource objects. Any `__resource_url__` supplied by your resource will be ignored when you pass `route_name`.

Invoking a Subrequest

New in version 1.4.

Pyramid allows you to invoke a subrequest at any point during the processing of a request. Invoking a subrequest allows you to obtain a *response* object from a view callable within your Pyramid application while you're executing a different view callable within the same application.

Here's an example application which uses a subrequest:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.request import Request

def view_one(request):
    subreq = Request.blank('/view_two')
    response = request.invoke_subrequest(subreq)
    return response

def view_two(request):
    request.response.body = 'This came from view_two'
    return request.response

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

30. INVOKING A SUBREQUEST

When `/view_one` is visited in a browser, the text printed in the browser pane will be `This came from view_two`. The `view_one` view used the `pyramid.request.Request.invoke_subrequest()` API to obtain a response from another view (`view_two`) within the same application when it executed. It did so by constructing a new request that had a URL that it knew would match the `view_two` view registration, and passed that new request along to `pyramid.request.Request.invoke_subrequest()`. The `view_two` view callable was invoked, and it returned a response. The `view_one` view callable then simply returned the response it obtained from the `view_two` view callable.

Note that it doesn't matter if the view callable invoked via a subrequest actually returns a *literal* `Response` object. Any view callable that uses a renderer or which returns an object that can be interpreted by a response adapter when found and invoked via `pyramid.request.Request.invoke_subrequest()` will return a `Response` object:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.request import Request

def view_one(request):
    subreq = Request.blank('/view_two')
    response = request.invoke_subrequest(subreq)
    return response

def view_two(request):
    return 'This came from view_two'

if __name__ == '__main__':
    config = Configurator()
    config.add_route('one', '/view_one')
    config.add_route('two', '/view_two')
    config.add_view(view_one, route_name='one')
    config.add_view(view_two, route_name='two', renderer='string')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

Even though the `view_two` view callable returned a string, it was invoked in such a way that the `string` renderer associated with the view registration that was found turned it into a “real” response object for consumption by `view_one`.

Being able to unconditionally obtain a response object by invoking a view callable indirectly is the main advantage to using `pyramid.request.Request.invoke_subrequest()` instead of simply importing a view callable and executing it directly. Note that there's not much advantage to invoking a view

using a subrequest if you *can* invoke a view callable directly. Subrequests are slower and are less convenient if you actually do want just the literal information returned by a function that happens to be a view callable.

Note that, by default, if a view callable invoked by a subrequest raises an exception, the exception will be raised to the caller of `invoke_subrequest()` even if you have a *exception view* configured:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.request import Request

def view_one(request):
    subreq = Request.blank('/view_two')
    response = request.invoke_subrequest(subreq)
    return response

def view_two(request):
    raise ValueError('foo')

def excview(request):
    request.response.body = b'An exception was raised'
    request.response.status_int = 500
    return request.response

if __name__ == '__main__':
    config = Configurator()
    config.add_route('one', '/view_one')
    config.add_route('two', '/view_two')
    config.add_view(view_one, route_name='one')
    config.add_view(view_two, route_name='two', renderer='string')
    config.add_view(excview, context=Exception)
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

When we run the above code and visit `/view_one` in a browser, the `excview` *exception view* will *not* be executed. Instead, the call to `invoke_subrequest()` will cause a `ValueError` exception to be raised and a response will never be generated. We can change this behavior; how to do so is described below in our discussion of the `use_tweens` argument.

The `pyramid.request.Request.invoke_subrequest()` API accepts two arguments: a positional argument `request` that must be provided, and `use_tweens` keyword argument that is optional; it defaults to `False`.

The `request` object passed to the API must be an object that implements the Pyramid request interface (such as a `pyramid.request.Request` instance). If `use_tweens` is `True`, the request will be

30. INVOKING A SUBREQUEST

sent to the *tween* in the tween stack closest to the request ingress. If `use_tweens` is `False`, the request will be sent to the main router handler, and no tweens will be invoked.

In the example above, the call to `invoke_subrequest()` will always raise an exception. This is because it's using the default value for `use_tweens`, which is `False`. You can pass `use_tweens=True` instead to ensure that it will convert an exception to a `Response` if an *exception view* is configured instead of raising the exception. This is because exception views are called by the exception view *tween* as described in *Custom Exception Views* when any view raises an exception.

We can cause the subrequest to be run through the tween stack by passing `use_tweens=True` to the call to `invoke_subrequest()`, like this:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.request import Request

def view_one(request):
    subreq = Request.blank('/view_two')
    response = request.invoke_subrequest(subreq, use_tweens=True)
    return response

def view_two(request):
    raise ValueError('foo')

def excview(request):
    request.response.body = b'An exception was raised'
    request.response.status_int = 500
    return request.response

if __name__ == '__main__':
    config = Configurator()
    config.add_route('one', '/view_one')
    config.add_route('two', '/view_two')
    config.add_view(view_one, route_name='one')
    config.add_view(view_two, route_name='two', renderer='string')
    config.add_view(excview, context=Exception)
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

In the above case, the call to `request.invoke_subrequest(subreq)` will not raise an exception. Instead, it will retrieve a “500” response from the attempted invocation of `view_two`, because the tween which invokes an exception view to generate a response is run, and therefore `excview` is executed.

This is one of the major differences between specifying the `use_tweens=True` and `use_tweens=False` arguments to `invoke_subrequest()`. `use_tweens=True` may

also imply invoking transaction commit/abort for the logic executed in the subrequest if you've got `pyramid_tm` in the tween list, injecting debug HTML if you've got `pyramid_debugtoolbar` in the tween list, and other tween-related side effects as defined by your particular tween list.

The `invoke_subrequest()` function also unconditionally:

- manages the threadlocal stack so that `get_current_request()` and `get_current_registry()` work during a request (they will return the subrequest instead of the original request)
- Adds a `registry` attribute and a `invoke_subrequest` attribute (a callable) to the request object it's handed.
- sets request extensions (such as those added via `add_request_method()` or `set_request_property()`) on the subrequest object passed as `request`
- causes a `NewRequest` event to be sent at the beginning of request processing.
- causes a `ContextFound` event to be sent when a context resource is found.
- Ensures that the user implied by the request passed has the necessary authorization to invoke view callable before calling it.
- Calls any *response callback* functions defined within the subrequest's lifetime if a response is obtained from the Pyramid application.
- causes a `NewResponse` event to be sent if a response is obtained.
- Calls any *finished callback* functions defined within the subrequest's lifetime.

The invocation of a subrequest has more or less exactly the same effect as the invocation of a request received by the Pyramid router from a web client when `use_tweens=True`. When `use_tweens=False`, the tweens are skipped but all the other steps take place.

It's a poor idea to use the original `request` object as an argument to `invoke_subrequest()`. You should construct a new request instead as demonstrated in the above example, using `pyramid.request.Request.blank()`. Once you've constructed a request object, you'll need to massage it to match the view callable you'd like to be executed during the subrequest. This can be done by adjusting the subrequest's URL, its headers, its request method, and other attributes. The documentation for `pyramid.request.Request` exposes the methods you should call and attributes you should set on the request you create to massage it into something that will actually match the view you'd like to call via a subrequest.

We've demonstrated use of a subrequest from within a view callable, but you can use the `invoke_subrequest()` API from within a tween or an event handler as well. It's usually a poor idea to invoke `invoke_subrequest()` from within a tween, because tweens already by definition have access to a function that will cause a subrequest (they are passed a `handle` function), but you can do it. It's fine to invoke `invoke_subrequest()` from within an event handler, however.

Using Hooks

“Hooks” can be used to influence the behavior of the Pyramid framework in various ways.

31.1 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*. The default Not Found View can be overridden through application configuration.

If your application uses *imperative configuration*, you can replace the Not Found View by using the `pyramid.config.Configurator.add_notfound_view()` method:

```
1 def notfound(request):
2     return Response('Not Found, dude', status='404 Not Found')
3
4 def main(globals, **settings):
5     config = Configurator()
6     config.add_notfound_view(notfound)
```

The *Not Found View* callable is a view callable like any other.

If your application instead uses `pyramid.view.view_config` decorators and a *scan*, you can replace the Not Found View by using the `pyramid.view.notfound_view_config` decorator:

31. USING HOOKS

```
1 from pyramid.view import notfound_view_config
2
3 @notfound_view_config()
4 def notfound(request):
5     return Response('Not Found, dude', status='404 Not Found')
6
7 def main(globals, **settings):
8     config = Configurator()
9     config.scan()
```

This does exactly what the imperative example above showed.

Your application can define *multiple* Not Found Views if necessary. Both `pyramid.config.Configurator.add_notfound_view()` and `pyramid.view.notfound_view_config` take most of the same arguments as `pyramid.config.Configurator.add_view` and `pyramid.view.view_config`, respectively. This means that Not Found Views can carry predicates limiting their applicability. For example:

```
1 from pyramid.view import notfound_view_config
2
3 @notfound_view_config(request_method='GET')
4 def notfound_get(request):
5     return Response('Not Found during GET, dude', status='404 Not Found')
6
7 @notfound_view_config(request_method='POST')
8 def notfound_post(request):
9     return Response('Not Found during POST, dude', status='404 Not Found')
10
11 def main(globals, **settings):
12     config = Configurator()
13     config.scan()
```

The `notfound_get` view will be called when a view could not be found and the request method was GET. The `notfound_post` view will be called when a view could not be found and the request method was POST.

Like any other view, the Not Found 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 `HTTPNotFound` exception that caused the view to be called.

Both `pyramid.config.Configurator.add_notfound_view()` and `pyramid.view.notfound_view_config` can be used to automatically redirect requests to slash-appended routes. See *Redirecting to Slash-Appended Routes* for examples.

Here's some sample code that implements a minimal *Not Found View* callable:

```
1 from pyramid.httpexceptions import HTTPNotFound
2
3 def notfound(request):
4     return HTTPNotFound()
```



When a Not Found View callable is invoked, it is passed a *request*. The `exception` attribute of the request will be an instance of the `HTTPNotFound` exception that caused the Not Found View to be called. The value of `request.exception.message` will be a value explaining why the Not Found error was raised. This message has different values depending whether the `pyramid.debug_notfound` environment setting is true or false.



Both `pyramid.config.Configurator.add_notfound_view()` and `pyramid.view.notfound_view_config` are new as of Pyramid 1.3. Older Pyramid documentation instructed users to use `add_view` instead, with a context of `HTTPNotFound`. This still works; the convenience method and decorator are just wrappers around this functionality.



When a Not Found 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 `HTTPNotFound` exception instance. If available, the resource context will still be available as `request.context`.

31.2 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 “forbidden” view consists of using the `pyramid.config.Configurator.add_forbidden_view()` API or the `pyramid.view.forbidden_view_config` decorator.

For example, you can add a forbidden view by using the `pyramid.config.Configurator.add_forbidden_view()` method to register a forbidden view:

31. USING HOOKS

```
1 def forbidden(request):
2     return Response('forbidden')
3
4 def main(globals, **settings):
5     config = Configurator()
6     config.add_forbidden_view(forbidden_view)
```

If instead you prefer to use decorators and a *scan*, you can use the `pyramid.view.forbidden_view_config` decorator to mark a view callable as a forbidden view:

```
1 from pyramid.view import forbidden_view_config
2
3 @forbidden_view_config()
4 def forbidden(request):
5     return Response('forbidden')
6
7 def main(globals, **settings):
8     config = Configurator()
9     config.scan()
```

Like any other view, the forbidden view must accept at least a `request` parameter, or both `context` and `request`. If a forbidden view callable accepts both `context` and `request`, the `HTTPException` is passed as `context`. The context as found by the router when view was denied (that you normally would expect) is available as `request.context`. The request is the current *request* representing the denied action.

Here's some sample code that implements a minimal forbidden view:

```
1 from pyramid.view 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 `HTTPForbidden` exception that caused the forbidden view to be called. The value of `request.exception.message` will be a value explaining why the forbidden was raised and `request.exception.result` will be extended information about the forbidden exception. These messages have different values depending whether the `pyramid.debug_authorization` environment setting is true or false.

31.3 Changing the Request Factory

Whenever Pyramid handles a request from a *WSGI* server, 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)
```

31.4 Adding Methods or Properties to Request Object

New in version 1.4..

Since each Pyramid application can only have one *request* factory, *changing the request factory* is not that extensible, especially if you want to build composable features (e.g., Pyramid add-ons and plugins).

A lazy property can be registered to the request object via the `pyramid.config.Configurator.add_request_method()` API. This allows you to

31. USING HOOKS

specify a callable that will be available on the request object, but will not actually execute the function until accessed.



This will silently override methods and properties from *request factory* that have the same name.

```
1 from pyramid.config import Configurator
2
3 def total(request, *args):
4     return sum(args)
5
6 def prop(request):
7     print("getting the property")
8     return "the property"
9
10 config = Configurator()
11 config.add_request_method(total)
12 config.add_request_method(prop, reify=True)
```

In the above example, `total` is added as a method. However, `prop` is added as a property and its result is cached per-request by setting `reify=True`. This way, we eliminate the overhead of running the function multiple times.

```
>>> request.total(1, 2, 3)
6
>>> request.prop
getting the property
the property
>>> request.prop
the property
```

To not cache the result of `request.prop`, set `property=True` instead of `reify=True`.

Here is an example of passing a class to `Configurator.add_request_method`:

```
1 from pyramid.config import Configurator
2 from pyramid.decorator import reify
3
4 class ExtraStuff(object):
5
6     def __init__(self, request):
7         self.request = request
8
```

```

9     def total(self, *args):
10         return sum(args)
11
12     # use @property if you don't want to cache the result
13     @reify
14     def prop(self):
15         print("getting the property")
16         return "the property"
17
18 config = Configurator()
19 config.add_request_method(ExtraStuff, 'extra', reify=True)

```

We attach and cache an object named `extra` to the request object.

```

>>> request.extra.total(1, 2, 3)
6
>>> request.extra.prop
getting the property
the property
>>> request.extra.prop
the property

```

31.5 Using The Before Render Event

Subscribers to the `pyramid.events.BeforeRender` event may introspect 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.

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.

31. USING HOOKS

The set of keys added to the renderer globals dictionary by all `pyramid.events.BeforeRender` subscribers and renderer globals factories must be unique.

The dictionary returned from the view is accessible through the `rendering_val` attribute of a `BeforeRender` event.

Suppose you return `{'mykey': 'somevalue', 'mykey2': 'somevalue2'}` from your view callable, like so:

```
1 from pyramid.view import view_config
2
3 @view_config(renderer='some_renderer')
4 def myview(request):
5     return {'mykey': 'somevalue', 'mykey2': 'somevalue2'}
```

`rendering_val` can be used to access these values from the `BeforeRender` object:

```
1 from pyramid.events import subscriber
2 from pyramid.events import BeforeRender
3
4 @subscriber(BeforeRender)
5 def read_return(event):
6     # {'mykey': 'somevalue'} is returned from the view
7     print(event.rendering_val['mykey'])
```

See the API documentation for the `BeforeRender` event interface at `pyramid.interfaces.IBeforeRender`.

31.6 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 whatever response object is returned by a view, usually in order to mutate the response.

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 *before* 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).

31.7 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 logging
2
3 log = logging.getLogger(__name__)
4
5 def log_callback(request):
6     """Log information at the end of request"""
7     log.debug('Request is finished.')
8     request.add_finished_callback(log_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.

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).

31.8 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.

```
1 from pyramid.config import Configurator
2 from myapp.traversal import Traverser
3 config = Configurator()
4 config.add_traverser(Traverser)
```

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 myapp.traversal import Traverser
2  from myapp.resources import MyRoot
3  from pyramid.config import Configurator
4  config = Configurator()
5  config.add_traverser(Traverser, MyRoot)

```

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.

31.9 Changing How `pyramid.request.Request.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.request.Request.resource_url()` API. However, since the way traversal

31. USING HOOKS

is done will have been modified, the URLs it generates by default may be incorrect when used against resources derived from your custom traverser.

If you've added a traverser, you can change how `resource_url()` generates a URL for a specific type of resource by adding a call to `pyramid.config.Configurator.add_resource_url_adapter()`.

For example:

```
1 from myapp.traversal import ResourceURLAdapter
2 from myapp.resources import MyRoot
3
4 config.add_resource_url_adapter(ResourceURLAdapter, MyRoot)
```

In the above example, the `myapp.traversal.ResourceURLAdapter` class will be used to provide services to `resource_url()` any time the *resource* passed to `resource_url` is of the class `myapp.resources.MyRoot`. The `resource_iface` argument `MyRoot` represents the type of interface that must be possessed by the resource for this resource url factory to be found. If the `resource_iface` argument is omitted, this resource url adapter will be used for *all* resources.

The API that must be implemented by a class that provides `IResourceURL` is as follows:

```
1 class MyResourceURL(object):
2     """ An adapter which provides the virtual and physical paths of a
3         resource
4     """
5     def __init__(self, resource, request):
6         """ Accept the resource and request and set self.physical_path and
7             self.virtual_path"""
8         self.virtual_path = some_function_of(resource, request)
9         self.physical_path = some_other_function_of(resource, request)
```

The default context URL generator is available for perusal as the class `pyramid.traversal.ResourceURL` in the traversal module of the *Pylons* GitHub Pyramid repository.

See `pyramid.config.add_resource_url_adapter()` for more information.

31.10 Changing How Pyramid Treats View Responses

New in version 1.1.

It is possible to control how Pyramid treats the result of calling a view callable on a per-type basis by using a hook involving `pyramid.config.Configurator.add_response_adapter()` or the `response_adapter` decorator.

Pyramid, in various places, adapts the result of calling a view callable to the `IResponse` interface to ensure that the object returned by the view callable is a “true” response object. The vast majority of time, the result of this adaptation is the result object itself, as view callables written by “civilians” who read the narrative documentation contained in this manual will always return something that implements the `IResponse` interface. Most typically, this will be an instance of the `pyramid.response.Response` class or a subclass. If a civilian returns a non-`Response` object from a view callable that isn’t configured to use a *renderer*, he will typically expect the router to raise an error. However, you can hook Pyramid in such a way that users can return arbitrary values from a view callable by providing an adapter which converts the arbitrary return value into something that implements `IResponse`.

For example, if you’d like to allow view callables to return bare string objects (without requiring a *renderer* to convert a string to a response object), you can register an adapter which converts the string to a `Response`:

```
1 from pyramid.response import Response
2
3 def string_response_adapter(s):
4     response = Response(s)
5     return response
6
7 # config is an instance of pyramid.config.Configurator
8
9 config.add_response_adapter(string_response_adapter, str)
```

Likewise, if you want to be able to return a simplified kind of response object from view callables, you can use the `IResponse` hook to register an adapter to the more complex `IResponse` interface:

```
1 from pyramid.response import Response
2
3 class SimpleResponse(object):
4     def __init__(self, body):
5         self.body = body
6
```

31. USING HOOKS

```
7 def simple_response_adapter(simple_response):
8     response = Response(simple_response.body)
9     return response
10
11 # config is an instance of pyramid.config.Configurator
12
13 config.add_response_adapter(simple_response_adapter, SimpleResponse)
```

If you want to implement your own `Response` object instead of using the `pyramid.response.Response` object in any capacity at all, you'll have to make sure the object implements every attribute and method outlined in `pyramid.interfaces.IResponse` and you'll have to ensure that it uses `zope.interface.implementer(IResponse)` as a class decorator.

```
1 from pyramid.interfaces import IResponse
2 from zope.interface import implementer
3
4 @implementer(IResponse)
5 class MyResponse(object):
6     # ... an implementation of every method and attribute
7     # documented in IResponse should follow ...
```

When an alternate response object implementation is returned by a view callable, if that object asserts that it implements `IResponse` (via `zope.interface.implementer(IResponse)`), an adapter needn't be registered for the object; Pyramid will use it directly.

An `IResponse` adapter for `webob.Response` (as opposed to `pyramid.response.Response`) is registered by Pyramid by default at startup time, as by their nature, instances of this class (and instances of subclasses of the class) will natively provide `IResponse`. The adapter registered for `webob.Response` simply returns the response object.

Instead of using `pyramid.config.Configurator.add_response_adapter()`, you can use the `pyramid.response.response_adapter` decorator:

```
1 from pyramid.response import Response
2 from pyramid.response import response_adapter
3
4 @response_adapter(str)
5 def string_response_adapter(s):
6     response = Response(s)
7     return response
```

The above example, when scanned, has the same effect as:

```
config.add_response_adapter(string_response_adapter, str)
```

The `response_adapter` decorator will have no effect until activated by a *scan*.

31.11 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):
```

31. USING HOOKS

```
10         matchdict = request.matchdict.copy()
11         matchdict.pop('action', None)
12         inst = view(request)
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 pyramid.response import Response
4  from pyramid.config import Configurator
5  import pyramid_handlers
6  from wsgiref.simple_server import make_server
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      server.make_server('0.0.0.0', 8080, config.make_wsgi_app())
18      server.serve_forever()
```

The `pyramid.config.Configurator.set_view_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()`.

31.12 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 mypackage.interfaces import IMyUtility
3
4 class registerFunction(object):
5
6     def __init__(self, path):
7         self.path = path
8
9     def register(self, scanner, name, wrapped):
10         registry = scanner.config.registry
11         registry.getUtility(IMyUtility).register(
12             self.path, wrapped)
13
14     def __call__(self, wrapped):
15         venusian.attach(wrapped, self.register)
16         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 zope.interface import implementer
2
3 from wsgiref.simple_server import make_server
4 from pyramid.config import Configurator
5 from mypackage.interfaces import IMyUtility
6
7 @implementer(IMyUtility)
```

```
8 class UtilityImplementation:
9
10     def __init__(self):
11         self.registrations = {}
12
13     def register(self, path, callable_):
14         self.registrations[path] = callable_
15
16 if __name__ == '__main__':
17     config = Configurator()
18     config.registry.registerUtility(UtilityImplementation())
19     config.scan()
20     app = config.make_wsgi_app()
21     server = make_server('0.0.0.0', 8080, app)
22     server.serve_forever()
```

For full details, please read the Venusian documentation.

31.13 Registering Tweens

New in version 1.2: Tweens

A *tween* (a contraction of the word “between”) is a bit of code that sits between the Pyramid router’s main request handling function and the upstream WSGI component that uses Pyramid as its “app”. This is a feature that may be used by Pyramid framework extensions, to provide, for example, Pyramid-specific view timing support bookkeeping code that examines exceptions before they are returned to the upstream WSGI application. Tweens behave a bit like *WSGI middleware* but they have the benefit of running in a context in which they have access to the Pyramid *request*, *response* and *application registry* as well as the Pyramid rendering machinery.

31.13.1 Creating a Tween

To create a tween, you must write a “tween factory”. A tween factory must be a globally importable callable which accepts two arguments: *handler* and *registry*. *handler* will be either the main Pyramid request handling function or another tween. *registry* will be the Pyramid *application registry* represented by this Configurator. A tween factory must return the tween (a callable object) when it is called.

A tween is called with a single argument, *request*, which is the *request* created by Pyramid’s router when it receives a WSGI request. A tween should return a *response*, usually the one generated by the downstream Pyramid application.

You can write the tween factory as a simple closure-returning function:

```
1 def simple_tween_factory(handler, registry):
2     # one-time configuration code goes here
3
4     def simple_tween(request):
5         # code to be executed for each request before
6         # the actual application code goes here
7
8         response = handler(request)
9
10        # code to be executed for each request after
11        # the actual application code goes here
12
13        return response
14
15    return simple_tween
```

Alternatively, the tween factory can be a class with the `__call__` magic method:

```
1 class simple_tween_factory(object):
2     def __init__(self, handler, registry):
3         self.handler = handler
4         self.registry = registry
5
6         # one-time configuration code goes here
7
8     def __call__(self, request):
9         # code to be executed for each request before
10        # the actual application code goes here
11
12        response = self.handler(request)
13
14        # code to be executed for each request after
15        # the actual application code goes here
16
17        return response
```

The closure style performs slightly better and enables you to conditionally omit the tween from the request processing pipeline (see the following timing tween example), whereas the class style makes it easier to have shared mutable state, and it allows subclassing.

Here's a complete example of a tween that logs the time spent processing each request:

```
1 # in a module named myapp.twens
2
3 import time
4 from pyramid.settings import asbool
5 import logging
6
7 log = logging.getLogger(__name__)
8
9 def timing_tween_factory(handler, registry):
10     if asbool(registry.settings.get('do_timing')):
11         # if timing support is enabled, return a wrapper
12         def timing_tween(request):
13             start = time.time()
14             try:
15                 response = handler(request)
16             finally:
17                 end = time.time()
18                 log.debug('The request took %s seconds' %
19                           (end - start))
20             return response
21         return timing_tween
22     # if timing support is not enabled, return the original
23     # handler
24     return handler
```

In the above example, the tween factory defines a `timing_tween` tween and returns it if `asbool(registry.settings.get('do_timing'))` is true. It otherwise simply returns the handler it was given. The `registry.settings` attribute is a handle to the deployment settings provided by the user (usually in an `.ini` file). In this case, if the user has defined a `do_timing` setting, and that setting is `True`, the user has said she wants to do timing, so the tween factory returns the timing tween; it otherwise just returns the handler it has been provided, preventing any timing.

The example timing tween simply records the start time, calls the downstream handler, logs the number of seconds consumed by the downstream handler, and returns the response.

31.13.2 Registering an Implicit Tween Factory

Once you've created a tween factory, you can register it into the implicit tween chain using the `pyramid.config.Configurator.add_tween()` method using its *dotted Python name*.

Here's an example of registering a tween factory as an “implicit” tween in a Pyramid application:

```
1 from pyramid.config import Configurator
2 config = Configurator()
3 config.add_tween('myapp.tweens.timing_tween_factory')
```

Note that you must use a *dotted Python name* as the first argument to `pyramid.config.Configurator.add_tween()`; this must point at a tween factory. You cannot pass the tween factory object itself to the method: it must be *dotted Python name* that points to a globally importable object. In the above example, we assume that a `timing_tween_factory` tween factory was defined in a module named `myapp.tweens`, so the tween factory is importable as `myapp.tweens.timing_tween_factory`.

When you use `pyramid.config.Configurator.add_tween()`, you’re instructing the system to use your tween factory at startup time unless the user has provided an explicit tween list in his configuration. This is what’s meant by an “implicit” tween. A user can always elect to supply an explicit tween list, reordering or disincluding implicitly added tweens. See *Explicit Tween Ordering* for more information about explicit tween ordering.

If more than one call to `pyramid.config.Configurator.add_tween()` is made within a single application configuration, the tweens will be chained together at application startup time. The *first* tween factory added via `add_tween` will be called with the Pyramid exception view tween factory as its `handler` argument, then the tween factory added directly after that one will be called with the result of the first tween factory as its `handler` argument, and so on, ad infinitum until all tween factories have been called. The Pyramid router will use the outermost tween produced by this chain (the tween generated by the very last tween factory added) as its request handler function. For example:

```
1 from pyramid.config import Configurator
2
3 config = Configurator()
4 config.add_tween('myapp.tween_factory1')
5 config.add_tween('myapp.tween_factory2')
```

The above example will generate an implicit tween chain that looks like this:

```
INGRESS (implicit)
myapp.tween_factory2
myapp.tween_factory1
pyramid.tweens.excview_tween_factory (implicit)
MAIN (implicit)
```

31.13.3 Suggesting Implicit Tween Ordering

By default, as described above, the ordering of the chain is controlled entirely by the relative ordering of calls to `pyramid.config.Configurator.add_tween()`. However, the caller of `add_tween` can provide an optional hint that can influence the implicit tween chain ordering by supplying `under` or `over` (or both) arguments to `add_tween()`. These hints are only used when an explicit tween ordering is not used. See *Explicit Tween Ordering* for a description of how to set an explicit tween ordering.

Allowable values for `under` or `over` (or both) are:

- `None` (the default).
- A *dotted Python name* to a tween factory: a string representing the predicted dotted name of a tween factory added in a call to `add_tween` in the same configuration session.
- One of the constants `pyramid.tweens.MAIN`, `pyramid.tweens.INGRESS`, or `pyramid.tweens.EXCVIEW`.
- An iterable of any combination of the above. This allows the user to specify fallbacks if the desired tween is not included, as well as compatibility with multiple other tweens.

Effectively, `over` means “closer to the request ingress than” and `under` means “closer to the main Pyramid application than”. You can think of an onion with outer layers over the inner layers, the application being under all the layers at the center.

For example, the following call to `add_tween()` will attempt to place the tween factory represented by `myapp.tween_factory` directly ‘above’ (in `ptweens` order) the main Pyramid request handler.

```
1 import pyramid.tweens
2
3 config.add_tween('myapp.tween_factory', over=pyramid.tweens.MAIN)
```

The above example will generate an implicit tween chain that looks like this:

```
INGRESS (implicit)
pyramid.tweens.excview_tween_factory (implicit)
myapp.tween_factory
MAIN (implicit)
```

Likewise, calling the following call to `add_tween()` will attempt to place this tween factory ‘above’ the main handler but ‘below’ a separately added tween factory:

```

1 import pyramid.tweens
2
3 config.add_tween('myapp.tween_factory1',
4                 over=pyramid.tweens.MAIN)
5 config.add_tween('myapp.tween_factory2',
6                 over=pyramid.tweens.MAIN,
7                 under='myapp.tween_factory1')

```

The above example will generate an implicit tween chain that looks like this:

```

INGRESS (implicit)
pyramid.tweens.excview_tween_factory (implicit)
myapp.tween_factory1
myapp.tween_factory2
MAIN (implicit)

```

Specifying neither `over` nor `under` is equivalent to specifying `under=INGRESS`.

If all options for `under` (or `over`) cannot be found in the current configuration, it is an error. If some options are specified purely for compatibility with other tweens, just add a fallback of `MAIN` or `INGRESS`. For example, `under=('someothertween', 'someothertween2', INGRESS)`. This constraint will require the tween to be located under both the 'someothertween' tween, the 'someothertween2' tween, and `INGRESS`. If any of these is not in the current configuration, this constraint will only organize itself based on the tweens that are present.

31.13.4 Explicit Tween Ordering

Implicit tween ordering is obviously only best-effort. Pyramid will attempt to provide an implicit order of tweens as best it can using hints provided by calls to `add_tween()`, but because it's only best-effort, if very precise tween ordering is required, the only surefire way to get it is to use an explicit tween order. The deploying user can override the implicit tween inclusion and ordering implied by calls to `add_tween()` entirely by using the `pyramid.tweens` settings value. When used, this settings value must be a list of Python dotted names which will override the ordering (and inclusion) of tween factories in the implicit tween chain. For example:

```

1 [app:main]
2 use = egg:MyApp
3 pyramid.reload_templates = true
4 pyramid.debug_authorization = false
5 pyramid.debug_notfound = false

```

31. USING HOOKS

```
6 pyramid.debug_routematch = false
7 pyramid.debug_templates = true
8 pyramid.tweens = myapp.my_cool_tween_factory
9                 pyramid.tweens.excview_tween_factory
```

In the above configuration, calls made during configuration to `pyramid.config.Configurator.add_tween()` are ignored, and the user is telling the system to use the tween factories he has listed in the `pyramid.tweens` configuration setting (each is a *dotted Python name* which points to a tween factory) instead of any tween factories added via `pyramid.config.Configurator.add_tween()`. The *first* tween factory in the `pyramid.tweens` list will be used as the producer of the effective Pyramid request handling function; it will wrap the tween factory declared directly “below” it, ad infinitum. The “main” Pyramid request handler is implicit, and always “at the bottom”.



Pyramid’s own *exception view* handling logic is implemented as a tween factory function: `pyramid.tweens.excview_tween_factory()`. If Pyramid exception view handling is desired, and tween factories are specified via the `pyramid.tweens` configuration setting, the `pyramid.tweens.excview_tween_factory()` function must be added to the `pyramid.tweens` configuration setting list explicitly. If it is not present, Pyramid will not perform exception view handling.

31.13.5 Tween Conflicts and Ordering Cycles

Pyramid will prevent the same tween factory from being added to the tween chain more than once using configuration conflict detection. If you wish to add the same tween factory more than once in a configuration, you should either: a) use a tween factory that is a separate globally importable instance object from the factory that it conflicts with b) use a function or class as a tween factory with the same logic as the other tween factory it conflicts with but with a different `__name__` attribute or c) call `pyramid.config.Configurator.commit()` between calls to `pyramid.config.Configurator.add_tween()`.

If a cycle is detected in implicit tween ordering when `over` and `under` are used in any call to “`add_tween`”, an exception will be raised at startup time.

31.13.6 Displaying Tween Ordering

The `ptweens` command-line utility can be used to report the current implicit and explicit tween chains used by an application. See *Displaying “Tweens”*.

31.14 Adding A Third Party View, Route, or Subscriber Predicate

New in version 1.4.

31.14.1 View and Route Predicates

View and route predicates used during configuration allow you to narrow the set of circumstances under which a view or route will match. For example, the `request_method` view predicate can be used to ensure a view callable is only invoked when the request's method is `POST`:

```
@view_config(request_method='POST')
def someview(request):
    ...
```

Likewise, a similar predicate can be used as a *route* predicate:

```
config.add_route('name', '/foo', request_method='POST')
```

Many other built-in predicates exist (request_param, and others). You can add third-party predicates to the list of available predicates by using one of `pyramid.config.Configurator.add_view_predicate()` or `pyramid.config.Configurator.add_route_predicate()`. The former adds a view predicate, the latter a route predicate.

When using one of those APIs, you pass a *name* and a *factory* to add a predicate during Pyramid's configuration stage. For example:

```
config.add_view_predicate('content_type', ContentTypePredicate)
```

The above example adds a new predicate named `content_type` to the list of available predicates for views. This will allow the following view configuration statement to work:

```
1 @view_config(content_type='File')
2 def aview(request): ...
```

31. USING HOOKS

The first argument to `pyramid.config.Configurator.add_view_predicate()`, the name, is a string representing the name that is expected to be passed to `view_config` (or its imperative analogue `add_view`).

The second argument is a view or route predicate factory, or a *dotted Python name* which refers to a view or route predicate factory. A view or route predicate factory is most often a class with a constructor (`__init__`), a `text` method, a `phash` method and a `__call__` method. For example:

```
1 class ContentTypePredicate(object):
2     def __init__(self, val, config):
3         self.val = val
4
5     def text(self):
6         return 'content_type = %s' % (self.val,)
7
8     phash = text
9
10    def __call__(self, context, request):
11        return getattr(context, 'content_type', None) == self.val
```

The constructor of a predicate factory takes two arguments: `val` and `config`. The `val` argument will be the argument passed to `view_config` (or `add_view`). In the example above, it will be the string `File`. The second arg, `config` will be the `Configurator` instance at the time of configuration.

The `text` method must return a string. It should be useful to describe the behavior of the predicate in error messages.

The `phash` method must return a string or a sequence of strings. It's most often the same as `text`, as long as `text` uniquely describes the predicate's name and the value passed to the constructor. If `text` is more general, or doesn't describe things that way, `phash` should return a string with the name and the value serialized. The result of `phash` is not seen in output anywhere, it just informs the uniqueness constraints for view configuration.

The `__call__` method of a predicate factory must accept a resource (`context`) and a request, and must return `True` or `False`. It is the “meat” of the predicate.

You can use the same predicate factory as both a view predicate and as a route predicate, but you'll need to call `add_view_predicate` and `add_route_predicate` separately with the same factory.

31.14.2 Subscriber Predicates

Subscriber predicates work almost exactly like view and route predicates. They narrow the set of circumstances in which a subscriber will be called. There are several minor differences between a subscriber predicate and a view/route predicate:

- There are no default subscriber predicates. You must register one to use one.
- The `__call__` method of a subscriber predicate accepts a single `event` object instead of a `context` and a `request`.
- Not every subscriber predicate can be used with every event type. Some subscriber predicates will assume a certain event type.

Here's an example of a subscriber predicate that can be used in conjunction with a subscriber that subscribes to the `pyramid.events.NewRequest` event type.

```
1 class RequestPathStartsWith(object):
2     def __init__(self, val, config):
3         self.val = val
4
5     def text(self):
6         return 'path_startswith = %s' % (self.val,)
7
8     phash = text
9
10    def __call__(self, event):
11        return event.request.path.startswith(self.val)
```

Once you've created a subscriber predicate, it may be registered via `pyramid.config.Configurator.add_subscriber_predicate()`. For example:

```
config.add_subscriber_predicate(
    'request_path_startswith', RequestPathStartsWith)
```

Once a subscriber predicate is registered, you can use it in a call to `pyramid.config.Configurator.add_subscriber()` or to `pyramid.events.subscriber`. Here's an example of using the previously registered `request_path_startswith` predicate in a call to `add_subscriber()`:

31. USING HOOKS

```
1 # define a subscriber in your code
2
3 def yosubscriber(event):
4     event.request.yo = 'YO!'
5
6 # and at configuration time
7
8 config.add_subscriber(yosubscriber, NewRequest,
9     request_path_startswith='/add_yo')
```

Here's the same subscriber/predicate/event-type combination used via subscriber.

```
1 from pyramid.events import subscriber
2
3 @subscriber(NewRequest, request_path_startswith='/add_yo')
4 def yosubscriber(event):
5     event.request.yo = 'YO!'
```

In either of the above configurations, the `yosubscriber` callable will only be called if the request path starts with `/add_yo`. Otherwise the event subscriber will not be called.

Note that the `request_path_startswith` subscriber you defined can be used with events that have a `request` attribute, but not ones that do not. So, for example, the predicate can be used with subscribers registered for `pyramid.events.NewRequest` and `pyramid.events.ContextFound` events, but it cannot be used with subscribers registered for `pyramid.events.ApplicationCreated` because the latter type of event has no `request` attribute. The point being: unlike route and view predicates, not every type of subscriber predicate will necessarily be applicable for use in every subscriber registration. It is not the responsibility of the predicate author to make every predicate make sense for every event type; it is the responsibility of the predicate consumer to use predicates that make sense for a particular event type registration.

Pyramid Configuration Introspection

New in version 1.3.

When Pyramid starts up, each call to a *configuration directive* causes one or more *introspectable* objects to be registered with an *introspector*. The introspector can be queried by application code to obtain information about the configuration of the running application. This feature is useful for debug toolbars, command-line scripts which show some aspect of configuration, and for runtime reporting of startup-time configuration settings.

32.1 Using the Introspector

Here's an example of using Pyramid's introspector from within a view callable:

```
1 from pyramid.view import view_config
2 from pyramid.response import Response
3
4 @view_config(route_name='bar')
5 def show_current_route_pattern(request):
6     introspector = request.registry.introspector
7     route_name = request.matched_route.name
8     route_intr = introspector.get('routes', route_name)
9     return Response(str(route_intr['pattern']))
```

This view will return a response that contains the “pattern” argument provided to the `add_route` method of the route which matched when the view was called. It uses the `pyramid.interfaces.IIntrospector.get()` method to return an introspectable in the category routes with a *discriminator* equal to the matched route name. It then uses the returned introspectable to obtain a “pattern” value.

The introspectable returned by the query methods of the introspector has methods and attributes described by `pyramid.interfaces.IIntrospectable`. In particular, the `get()`, `get_category()`, `categories()`, `categorized()`, and `related()` methods of an introspector can be used to query for introspectables.

32.2 Introspectable Objects

Introspectable objects are returned from query methods of an introspector. Each introspectable object implements the attributes and methods documented at `pyramid.interfaces.IIntrospectable`.

The important attributes shared by all introspectables are the following:

`title`

A human-readable text title describing the introspectable

`category_name`

A text category name describing the introspection category to which this introspectable belongs. It is often a plural if there are expected to be more than one introspectable registered within the category.

`discriminator`

A hashable object representing the unique value of this introspectable within its category.

`discriminator_hash`

The integer hash of the discriminator (useful for using in HTML links).

`type_name`

The text name of a subtype within this introspectable’s category. If there is only one type name in this introspectable’s category, this value will often be a singular version of the category name but it can be an arbitrary value.

`action_info`

An object describing the directive call site which caused this introspectable to be registered; contains attributes described in `pyramid.interfaces.IActionInfo`.

Besides having the attributes described above, an introspectable is a dictionary-like object. An introspectable can be queried for data values via its `__getitem__`, `get`, `keys`, `values`, or `items` methods. For example:

```
1 route_intr = introspector.get('routes', 'edit_user')
2 pattern = route_intr['pattern']
```

32.3 Pyramid Introspection Categories

The list of concrete introspection categories provided by built-in Pyramid configuration directives follows. Add-on packages may supply other introspectables in categories not described here.

`subscribers`

Each introspectable in the `subscribers` category represents a call to `pyramid.config.Configurator.add_subscriber()` (or the decorator equivalent); each will have the following data.

`subscriber`

The subscriber callable object (the resolution of the `subscriber` argument passed to `add_subscriber`).

`interfaces`

A sequence of interfaces (or classes) that are subscribed to (the resolution of the `ifaces` argument passed to `add_subscriber`).

`derived_subscriber`

A wrapper around the subscriber used internally by the system so it can call it with more than one argument if your original subscriber accepts only one.

`predicates`

32. PYRAMID CONFIGURATION INTROSPECTION

The predicate objects created as the result of passing predicate arguments to `add_subscriber`

`derived_predicates`

Wrappers around the predicate objects created as the result of passing predicate arguments to `add_subscriber` (to be used when predicates take only one value but must be passed more than one).

`response adapters`

Each introspectable in the `response adapters` category represents a call to `pyramid.config.Configurator.add_response_adapter()` (or a decorator equivalent); each will have the following data.

`adapter`

The adapter object (the resolved adapter argument to `add_response_adapter`).

`type`

The resolved `type_or_iface` argument passed to `add_response_adapter`.

`root factories`

Each introspectable in the `root factories` category represents a call to `pyramid.config.Configurator.set_root_factory()` (or the `Configurator` constructor equivalent) *or* a factory argument passed to `pyramid.config.Configurator.add_route()`; each will have the following data.

`factory`

The factory object (the resolved factory argument to `set_root_factory`).

`route_name`

The name of the route which will use this factory. If this is the *default* root factory (if it's registered during a call to `set_root_factory`), this value will be `None`.

`session factory`

Only one introspectable will exist in the `session_factory` category. It represents a call to `pyramid.config.Configurator.set_session_factory()` (or the `Configurator` constructor equivalent); it will have the following data.

`factory`

The factory object (the resolved factory argument to `set_session_factory`).

`request_factory`

Only one introspectable will exist in the `request_factory` category. It represents a call to `pyramid.config.Configurator.set_request_factory()` (or the `Configurator` constructor equivalent); it will have the following data.

`factory`

The factory object (the resolved factory argument to `set_request_factory`).

`locale_negotiator`

Only one introspectable will exist in the `locale_negotiator` category. It represents a call to `pyramid.config.Configurator.set_locale_negotiator()` (or the `Configurator` constructor equivalent); it will have the following data.

`negotiator`

The factory object (the resolved negotiator argument to `set_locale_negotiator`).

`renderer_factories`

Each introspectable in the `renderer_factories` category represents a call to `pyramid.config.Configurator.add_renderer()` (or the `Configurator` constructor equivalent); each will have the following data.

`name`

The name of the renderer (the value of the `name` argument to `add_renderer`).

`factory`

The factory object (the resolved factory argument to `add_renderer`).

32. PYRAMID CONFIGURATION INTROSPECTION

routes

Each introspectable in the routes category represents a call to `pyramid.config.Configurator.add_route()`; each will have the following data.

name

The name argument passed to `add_route`.

pattern

The pattern argument passed to `add_route`.

factory

The (resolved) factory argument passed to `add_route`.

xhr

The `xhr` argument passed to `add_route`.

request_method

The `request_method` argument passed to `add_route`.

request_methods

A sequence of request method names implied by the `request_method` argument passed to `add_route` or the value `None` if a `request_method` argument was not supplied.

path_info

The `path_info` argument passed to `add_route`.

request_param

The `request_param` argument passed to `add_route`.

header

The header argument passed to `add_route`.

`accept`

The `accept` argument passed to `add_route`.

`traverse`

The `traverse` argument passed to `add_route`.

`custom_predicates`

The `custom_predicates` argument passed to `add_route`.

`pregenerator`

The `pregenerator` argument passed to `add_route`.

`static`

The `static` argument passed to `add_route`.

`use_global_views`

The `use_global_views` argument passed to `add_route`.

`object`

The `pyramid.interfaces.IRoute` object that is used to perform matching and generation for this route.

authentication policy

There will be one and only one introspectable in the authentication policy category. It represents a call to the `pyramid.config.Configurator.set_authentication_policy()` method (or its `Configurator` constructor equivalent); it will have the following data.

`policy`

The `policy` object (the resolved `policy` argument to `set_authentication_policy`).

authorization policy

32. PYRAMID CONFIGURATION INTROSPECTION

There will be one and only one introspectable in the authorization policy category. It represents a call to the `pyramid.config.Configurator.set_authorization_policy()` method (or its `Configurator` constructor equivalent); it will have the following data.

policy

The policy object (the resolved policy argument to `set_authorization_policy()`).

default permission

There will be one and only one introspectable in the default permission category. It represents a call to the `pyramid.config.Configurator.set_default_permission()` method (or its `Configurator` constructor equivalent); it will have the following data.

value

The permission name passed to `set_default_permission()`.

views

Each introspectable in the views category represents a call to `pyramid.config.Configurator.add_view()`; each will have the following data.

name

The name argument passed to `add_view()`.

context

The (resolved) context argument passed to `add_view()`.

containment

The (resolved) containment argument passed to `add_view()`.

request_param

The `request_param` argument passed to `add_view()`.

request_methods

A sequence of request method names implied by the `request_method` argument passed to `add_view` or the value `None` if a `request_method` argument was not supplied.

`route_name`

The `route_name` argument passed to `add_view`.

`attr`

The `attr` argument passed to `add_view`.

`xhr`

The `xhr` argument passed to `add_view`.

`accept`

The `accept` argument passed to `add_view`.

`header`

The `header` argument passed to `add_view`.

`path_info`

The `path_info` argument passed to `add_view`.

`match_param`

The `match_param` argument passed to `add_view`.

`csrf_token`

The `csrf_token` argument passed to `add_view`.

`callable`

The (resolved) view argument passed to `add_view`. Represents the “raw” view callable.

`derived_callable`

32. PYRAMID CONFIGURATION INTROSPECTION

The view callable derived from the `view` argument passed to `add_view`. Represents the view callable which Pyramid itself calls (wrapped in security and other wrappers).

`mapper`

The (resolved) `mapper` argument passed to `add_view`.

`decorator`

The (resolved) `decorator` argument passed to `add_view`.

`permissions`

Each introspectable in the `permissions` category represents a call to `pyramid.config.Configurator.add_view()` that has an explicit `permission` argument to *or* a call to `pyramid.config.Configurator.set_default_permission()`; each will have the following data.

`value`

The permission name passed to `add_view` or `set_default_permission`.

`templates`

Each introspectable in the `templates` category represents a call to `pyramid.config.Configurator.add_view()` that has a `renderer` argument which points to a template; each will have the following data.

`name`

The renderer's name (a string).

`type`

The renderer's type (a string).

`renderer`

The `pyramid.interfaces.IRendererInfo` object which represents this template's renderer.

`view_mapper`

Each introspectable in the `permissions` category represents a call to `pyramid.config.Configurator.add_view()` that has an explicit `mapper` argument to *or* a call to `pyramid.config.Configurator.set_view_mapper()`; each will have the following data.

`mapper`

The (resolved) `mapper` argument passed to `add_view` or `set_view_mapper`.

`asset overrides`

Each introspectable in the `asset overrides` category represents a call to `pyramid.config.Configurator.override_asset()`; each will have the following data.

`to_override`

The `to_override` argument (an asset spec) passed to `override_asset`.

`override_with`

The `override_with` argument (an asset spec) passed to `override_asset`.

`translation directories`

Each introspectable in the `asset overrides` category represents an individual element in a `specs` argument passed to `pyramid.config.Configurator.add_translation_dirs()`; each will have the following data.

`directory`

The absolute path of the translation directory.

`spec`

The asset specification passed to `add_translation_dirs`.

`tweens`

Each introspectable in the `tweens` category represents a call to `pyramid.config.Configurator.add_tween()`; each will have the following data.

`name`

32. PYRAMID CONFIGURATION INTROSPECTION

The dotted name to the tween factory as a string (passed as the `tween_factory` argument to `add_tween`).

factory

The (resolved) tween factory object.

type

implicit or explicit as a string.

under

The under argument passed to `add_tween` (a string).

over

The over argument passed to `add_tween` (a string).

static views

Each introspectable in the static views category represents a call to `pyramid.config.Configurator.add_static_view()`; each will have the following data.

name

The name argument provided to `add_static_view`.

spec

A normalized version of the spec argument provided to `add_static_view`.

traversers

Each introspectable in the traversers category represents a call to `pyramid.config.Configurator.add_traverser()`; each will have the following data.

iface

The (resolved) interface or class object that represents the return value of a root factory that this traverser will be used for.

adapter

The (resolved) traverser class.

resource url adapters

Each introspectable in the resource url adapters category represents a call to `pyramid.config.Configurator.add_resource_url_adapter()`; each will have the following data.

adapter

The (resolved) resource URL adapter class.

resource_iface

The (resolved) interface or class object that represents the resource interface that this url adapter is registered for.

request_iface

The (resolved) interface or class object that represents the request interface that this url adapter is registered for.

32.4 Introspection in the Toolbar

The Pyramid debug toolbar (part of the `pyramid_debugtoolbar` package) provides a canned view of all registered introspectables and their relationships. It looks something like this:

Introspection

Permissions

| permission | no_permission_required |
|------------|----------------------------|
| value | '_no_permission_required_' |

Source

Line 12 of file /home/chrism/projects/pyramid/env26/wiggystatic/wiggystatic/__init__.py:
 config.add_static_view('static', 'static', cache_max_age=3600)

References

- view object <pyramid.static.static_view object at 0x282fdd0>
- view method 'exception' of class pyramid_debugtoolbar.views.ExceptionDebugView
- view method 'source' of class pyramid_debugtoolbar.views.ExceptionDebugView
- view method 'execute' of class pyramid_debugtoolbar.views.ExceptionDebugView
- view method 'sql_select' of class pyramid_debugtoolbar.views.SQLAlchemyViews
- view method 'sql_explain' of class pyramid_debugtoolbar.views.SQLAlchemyViews
- view object <pyramid.static.static_view object at 0x289b210>

Renderer factories

| renderer factory function | pyramid.chameleon_text.renderer_factory |
|---------------------------|--|
| factory | <function renderer_factory at 0x1f06b18> |
| name | 'txt' |

Source

Line 357 of file /home/chrism/projects/pyramid/pyramid/config/__init__.py:

Hide »

Versions

Settings

HTTP Headers

Request Vars

Renderings

1 RENDERING

Logging

0 MESSAGES

Performance ✓

CPU: 100.00ms (107.60ms)

Routes

SQLAlchemy

Tweens

Introspection

32.5 Disabling Introspection

You can disable Pyramid introspection by passing the flag `introspection=False` to the *Configurator* constructor in your application setup:

```
from pyramid.config import Configurator
config = Configurator(..., introspection=False)
```

When `introspection` is `False`, all introspectables generated by configuration directives are thrown away.

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.

33.1 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 into 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*.

33.2 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 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*.

33.2.1 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, the `add_handler` directive of the `pyramid_handlers` package adds a single route, and some number of views.

33.3 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.

33.3.1 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.

33. EXTENDING AN EXISTING PYRAMID APPLICATION

```
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.

```
1 if __name__ == '__main__':
2     config.scan('someotherpackage')
3     config.commit()
4     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*.

33.3.2 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 scaffold 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. `$VENV/bin/python setup.py develop` or `$VENV/bin/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. This 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*.

33.3.3 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.

33.3.4 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.

33.3.5 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.

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.

34.1 Conflict Detection

Here's a familiar example of one of the simplest Pyramid applications, configured imperatively:

```
1 from wsgiref.simple_server import make_server
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    server = make_server('0.0.0.0', 8080, app)
13    server.serve_forever()
```

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?

34. ADVANCED CONFIGURATION

```
1 from wsgiref.simple_server import make_server
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     server = make_server('0.0.0.0', 8080, app)
21     server.serve_forever()
```

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     ... more code ...
9 pyramid.exceptions.ConfigurationConflictError:
10     Conflicting configuration actions
11 For: ('view', None, '', None, <InterfaceClass pyramid.interfaces.IView>,
12     None, None, None, None, None, False, None, None, None)
13 Line 14 of file app.py in <module>: 'config.add_view(hello_world)'
14 Line 17 of file app.py in <module>: 'config.add_view(goodbye_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*.

34.1.1 Manually Resolving Conflicts

There are a number of ways to manually resolve conflicts: by changing registrations to not conflict, 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.Configurator.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:

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 wsgiref.simple_server import make_server
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     server = make_server('0.0.0.0', 8080, app)
21     server.serve_forever()
```

We can prevent the two `add_view` calls from conflicting by issuing a call to `commit()` between them:

```

1 from wsgiref.simple_server import make_server
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     server = make_server('0.0.0.0', 8080, app)
23     server.serve_forever()

```

In the above example we’ve issued a call to `commit()` between the two `add_view` calls. `commit()` will execute 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)

```

34. ADVANCED CONFIGURATION

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 auto-committing Configurator, because you are usually unconcerned about conflict detection or two-phase configuration in test code.

34.1.2 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 that does the including, even if configuration statements in the included code would conflict if it was moved “up” to the calling code.

34.1.3 Methods Which Provide Conflict Detection

These are the methods of the configurator which provide conflict detection:

```
add_view(),      add_route(),      add_renderer(),      add_request_method(),
set_request_factory(), set_session_factory(), set_request_property(),
set_root_factory(), set_view_mapper(), set_authentication_policy(),
set_authorization_policy(),          set_locale_negotiator(),
set_default_permission(), add_traverser(), add_resource_url_adapter(), and
add_response_adapter().
```

`add_static_view()` also indirectly provides conflict detection, because it's implemented in terms of the conflict-aware `add_route` and `add_view` methods.

34.2 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.Configurator.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.

34.3 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.

34.4 More Information

For more information, see the article, “A Whirlwind Tour of Advanced Configuration Tactics”, in the Pyramid Cookbook.

Extending Pyramid Configuration

Pyramid allows you to extend its `Configurator` with custom directives. Custom directives can use other directives, they can add a custom *action*, they can participate in *conflict resolution*, and they can provide some number of *introspectable* objects.

35.1 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. Using `add_directive()` 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:

```
1 from pyramid.events import NewRequest
2 from pyramid.config import Configurator
3
4 def add_newrequest_subscriber(config, subscriber):
5     config.add_subscriber(subscriber, NewRequest)
6
7 if __name__ == '__main__':
8     config = Configurator()
9     config.add_directive('add_newrequest_subscriber',
10                          add_newrequest_subscriber)
```

35. EXTENDING PYRAMID CONFIGURATION

Once `add_directive()` is called, a user can then call the added directive 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`:

```
1 def includeme(config):
2     config.add_directive('add_newrequest_subscriber',
3                           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)
```

35.2 Using `config.action` in a Directive

If a custom directive can’t do its work exclusively in terms of existing configurator methods (such as `pyramid.config.Configurator.add_subscriber()`, as above), the directive may need to make use of the `pyramid.config.Configurator.action()` method. This method adds an entry to the list of “actions” that Pyramid will attempt to process when `pyramid.config.Configurator.commit()` is called. An action is simply a dictionary that includes a *discriminator*, possibly a callback function, and possibly other metadata used by Pyramid’s action system.

Here’s an example directive which uses the “action” method:

```
1 def add_jammyjam(config, jammyjam):
2     def register():
3         config.registry.jammyjam = jammyjam
4         config.action('jammyjam', register)
5
6 if __name__ == '__main__':
7     config = Configurator()
8     config.add_directive('add_jammyjam', add_jammyjam)
```

Fancy, but what does it do? The action method accepts a number of arguments. In the above directive named `add_jammyjam`, we call `action()` with two arguments: the string `jammyjam` is passed as the first argument named discriminator, and the closure function named `register` is passed as the second argument named callable.

When the `action()` method is called, it appends an action to the list of pending configuration actions. All pending actions with the same discriminator value are potentially in conflict with one another (see *Conflict Detection*). When the `commit()` method of the `Configurator` is called (either explicitly or as the result of calling `make_wsgi_app()`), conflicting actions are potentially automatically resolved as per *Automatic Conflict Resolution*. If a conflict cannot be automatically resolved, a `pyramid.exceptions.ConfigurationConflictError` is raised and application startup is prevented.

In our above example, therefore, if a consumer of our `add_jammyjam` directive did this:

```
config.add_jammyjam('first')
config.add_jammyjam('second')
```

When the action list was committed resulting from the set of calls above, our user's application would not start, because the discriminators of the actions generated by the two calls are in direct conflict. Automatic conflict resolution cannot resolve the conflict (because no `config.include` is involved), and the user provided no intermediate `pyramid.config.Configurator.commit()` call between the calls to `add_jammyjam` to ensure that the successive calls did not conflict with each other.

This demonstrates the purpose of the discriminator argument to the action method: it's used to indicate a uniqueness constraint for an action. Two actions with the same discriminator will conflict unless the conflict is automatically or manually resolved. A discriminator can be any hashable object, but it is generally a string or a tuple. *You use a discriminator to declaratively ensure that the user doesn't provide ambiguous configuration statements.*

But let's imagine that a consumer of `add_jammyjam` used it in such a way that no configuration conflicts are generated.

```
config.add_jammyjam('first')
```

What happens now? When the `add_jammyjam` method is called, an action is appended to the pending actions list. When the pending configuration actions are processed during `commit()`, and no conflicts occur, the *callable* provided as the second argument to the `action()` method within `add_jammyjam` is called with no arguments. The callable in `add_jammyjam` is the `register` closure function. It simply sets the value `config.registry.jammyjam` to whatever the user passed in as the `jammyjam` argument to the `add_jammyjam` function. Therefore, the result of the user's call to our directive will set the `jammyjam` attribute of the registry to the string `first`. *A callable is used by a directive to defer the result of a user's call to the directive until conflict detection has had a chance to do its job.*

Other arguments exist to the `action()` method, including `args`, `kw`, `order`, and `introspectables`.

`args` and `kw` exist as values, which, if passed, will be used as arguments to the callable function when it is called back. For example our directive might use them like so:

```
1 def add_jammyjam(config, jammyjam):
2     def register(*arg, **kw):
3         config.registry.jammyjam_args = arg
4         config.registry.jammyjam_kw = kw
5         config.registry.jammyjam = jammyjam
6         config.action('jammyjam', register, args=('one',), kw={'two':'two'})
```

In the above example, when this directive is used to generate an action, and that action is committed, `config.registry.jammyjam_args` will be set to `('one',)` and `config.registry.jammyjam_kw` will be set to `{'two':'two'}`. `args` and `kw` are honestly not very useful when your callable is a closure function, because you already usually have access to every local in the directive without needing them to be passed back. They can be useful, however, if you don't use a closure as a callable.

`order` is a crude order control mechanism. `order` defaults to the integer 0; it can be set to any other integer. All actions that share an order will be called before other actions that share a higher order. This makes it possible to write a directive with callable logic that relies on the execution of the callable of another directive being done first. For example, Pyramid's `pyramid.config.Configurator.add_view()` directive registers an action with a higher order than the `pyramid.config.Configurator.add_route()` method. Due to this, the `add_view` method's callable can assume that, if a `route_name` was passed to it, that a route by this name was already registered by `add_route`, and if such a route has not already been registered, it's a configuration error (a view that names a nonexistent route via its `route_name` parameter will never be called).

`introspectables` is a sequence of *introspectable* objects. You can pass a sequence of `introspectables` to the `action()` method, which allows you to augment Pyramid's configuration introspection system.

35.3 Adding Configuration Introspection

New in version 1.3.

Pyramid provides a configuration introspection system that can be used by debugging tools to provide visibility into the configuration of a running application.

All built-in Pyramid directives (such as `pyramid.config.Configurator.add_view()` and `pyramid.config.Configurator.add_route()`) register a set of introspectables when called. For example, when you register a view via `add_view`, the directive registers at least one introspectable: an introspectable about the view registration itself, providing human-consumable values for the arguments it was passed. You can later use the introspection query system to determine whether a particular view uses a renderer, or whether a particular view is limited to a particular request method, or which routes a particular view is registered against. The Pyramid “debug toolbar” makes use of the introspection system in various ways to display information to Pyramid developers.

Introspection values are set when a sequence of *introspectable* objects is passed to the `action()` method. Here’s an example of a directive which uses introspectables:

```

1 def add_jammyjam(config, value):
2     def register():
3         config.registry.jammyjam = value
4         intr = config.introspectable(category_name='jammyjams',
5                                     discriminator='jammyjam',
6                                     title='a jammyjam',
7                                     type_name=None)
8         intr['value'] = value
9         config.action('jammyjam', register, introspectables=(intr,))
10
11 if __name__ == '__main__':
12     config = Configurator()
13     config.add_directive('add_jammyjam', add_jammyjam)

```

If you notice, the above directive uses the `introspectable` attribute of a `Configurator` (`pyramid.config.Configurator.introspectable`) to create an introspectable object. The introspectable object’s constructor requires at least four arguments: the `category_name`, the `discriminator`, the `title`, and the `type_name`.

The `category_name` is a string representing the logical category for this introspectable. Usually the `category_name` is a pluralization of the type of object being added via the action.

The `discriminator` is a value unique **within the category** (unlike the action discriminator, which must be unique within the entire set of actions). It is typically a string or tuple representing the values

unique to this introspectable within the category. It is used to generate links and as part of a relationship-forming target for other introspectables.

The `title` is a human-consumable string that can be used by introspection system frontends to show a friendly summary of this introspectable.

The `type_name` is a value that can be used to subtype this introspectable within its category for sorting and presentation purposes. It can be any value.

An introspectable is also dictionary-like. It can contain any set of key/value pairs, typically related to the arguments passed to its related directive. While the `category_name`, `discriminator`, `title` and `type_name` are *metadata* about the introspectable, the values provided as key/value pairs are the actual data provided by the introspectable. In the above example, we set the `value` key to the value of the `value` argument passed to the directive.

Our directive above mutates the introspectable, and passes it in to the `action` method as the first element of a tuple as the value of the `introspectable` keyword argument. This associates this introspectable with the action. Introspection tools will then display this introspectable in their index.

35.3.1 Introspectable Relationships

Two introspectables may have relationships between each other.

```
1 def add_jammyjam(config, value, template):
2     def register():
3         config.registry.jammyjam = (value, template)
4         intr = config.introspectable(category_name='jammyjams',
5                                     discriminator='jammyjam',
6                                     title='a jammyjam',
7                                     type_name=None)
8         intr['value'] = value
9         tpl_intr = config.introspectable(category_name='jammyjam templates',
10                                         discriminator=template,
11                                         title=template,
12                                         type_name=None)
13         tpl_intr['value'] = template
14         intr.relate('jammyjam templates', template)
15         config.action('jammyjam', register, introspectables=(intr, tpl_intr))
16
17 if __name__ == '__main__':
18     config = Configurator()
19     config.add_directive('add_jammyjam', add_jammyjam)
```

In the above example, the `add_jammyjam` directive registers two introspectables. The first is related to the `value` passed to the directive; the second is related to the `template` passed to the directive. If you believe a concept within a directive is important enough to have its own introspectable, you can cause the same directive to register more than one introspectable, registering one introspectable for the “main idea” and another for a related concept.

The call to `intr.relate` above (`pyramid.interfaces.IIntrospectable.relate()`) is passed two arguments: a category name and a directive. The example above effectively indicates that the directive wishes to form a relationship between the `intr` introspectable and the `tmpl_intr` introspectable; the arguments passed to `relate` are the category name and discriminator of the `tmpl_intr` introspectable.

Relationships need not be made between two introspectables created by the same directive. Instead, a relationship can be formed between an introspectable created in one directive and another introspectable created in another by calling `relate` on either side with the other directive’s category name and discriminator. An error will be raised at configuration commit time if you attempt to relate an introspectable with another nonexistent introspectable, however.

Introspectable relationships will show up in frontend system renderings of introspection values. For example, if a view registration names a route name, the introspectable related to the view callable will show a reference to the route to which it relates to and vice versa.

Creating Pyramid Scaffolds

You can extend Pyramid by creating a *scaffold* template. A scaffold template is useful if you'd like to distribute a customizable configuration of Pyramid to other users. Once you've created a scaffold, and someone has installed the distribution that houses the scaffold, they can use the `pcreate` script to create a custom version of your scaffold's template. Pyramid itself uses scaffolds to allow people to bootstrap new projects. For example, `pcreate -s alchemy MyStuff` causes Pyramid to render the `alchemy` scaffold template to the `MyStuff` directory.

36.1 Basics

A scaffold template is just a bunch of source files and directories on disk. A small definition class points at this directory; it is in turn pointed at by a *setuptools* “entry point” which registers the scaffold so it can be found by the `pcreate` command.

To create a scaffold template, create a Python *distribution* to house the scaffold which includes a `setup.py` that relies on the `setuptools` package. See [Creating a Package](#) for more information about how to do this. For the sake of example, we'll pretend the distribution you create is named `CoolExtension`, and it has a package directory within it named `coolextension`.

Once you've created the distribution put a “scaffolds” directory within your distribution's package directory, and create a file within that directory named `__init__.py` with something like the following:

36. CREATING PYRAMID SCAFFOLDS

```
1 # CoolExtension/coolextension/scaffolds/__init__.py
2
3 from pyramid.scaffolds import PyramidTemplate
4
5 class CoolExtensionTemplate(PyramidTemplate):
6     _template_dir = 'coolextension_scaffold'
7     summary = 'My cool extension'
```

Once this is done, within the `scaffolds` directory, create a template directory. Our example used a template directory named `coolextension_scaffold`.

As you create files and directories within the template directory, note that:

- Files which have a name which are suffixed with the value `_tmpl` will be rendered, and replacing any instance of the literal string `{{var}}` with the string value of the variable named `var` provided to the scaffold.
- Files and directories with filenames that contain the string `+var+` will have that string replaced with the value of the `var` variable provided to the scaffold.

Otherwise, files and directories which live in the template directory will be copied directly without modification to the `pcreate` output location.

The variables provided by the default `PyramidTemplate` include `project` (the project name provided by the user as an argument to `pcreate`), `package` (a lowercasing and normalizing of the project name provided by the user), `random_string` (a long random string), and `package_logger` (the name of the package's logger).

See Pyramid's "scaffolds" package (<https://github.com/Pylons/pyramid/tree/master/pyramid/scaffolds>) for concrete examples of scaffold directories (`zodb`, `alchemy`, and `starter`, for example).

After you've created the template directory, add the following to the `entry_points` value of your distribution's `setup.py`:

```
[pyramid.scaffold]
coolextension=coolextension.scaffolds:CoolExtensionTemplate
```

For example:

```
def setup(
    ...,
    entry_points = """\
        [pyramid.scaffold]
        cooextension=cooextension.scaffolds:CoolExtensionTemplate
    """
)
```

Run your distribution's `setup.py develop` or `setup.py install` command. After that, you should be able to see your scaffolding template listed when you run `pcreate -l`. It will be named `cooextension` because that's the name we gave it in the entry point setup. Running `pcreate -s cooextension MyStuff` will then render your scaffold to an output directory named `MyStuff`.

See the module documentation for `pyramid.scaffolds` for information about the API of the `pyramid.scaffolds.Template` class and related classes. You can override methods of this class to get special behavior.

36.2 Supporting Older Pyramid Versions

Because different versions of Pyramid handled scaffolding differently, if you want to have extension scaffolds that can work across Pyramid 1.0.X, 1.1.X, 1.2.X and 1.3.X, you'll need to use something like this bit of horror while defining your scaffold template:

```
1  try: # pyramid 1.0.X
2      # "pyramid.paster.paste_script..." doesn't exist past 1.0.X
3      from pyramid.paster import paste_script_template_renderer
4      from pyramid.paster import PyramidTemplate
5  except ImportError:
6      try: # pyramid 1.1.X, 1.2.X
7          # trying to import "paste_script_template_renderer" fails on 1.3.X
8          from pyramid.scaffolds import paste_script_template_renderer
9          from pyramid.scaffolds import PyramidTemplate
10     except ImportError: # pyramid >=1.3a2
11         paste_script_template_renderer = None
12         from pyramid.scaffolds import PyramidTemplate
13
14 class CoolExtensionTemplate(PyramidTemplate):
15     _template_dir = 'cooextension_scaffold'
16     summary = 'My cool extension'
17     template_renderer = staticmethod(paste_script_template_renderer)
```

36. CREATING PYRAMID SCAFFOLDS

And then in the `setup.py` of the package that contains your scaffold, define the template as a target of both `paste.paster_create_template` (for `paster create`) and `pyramid.scaffold` (for `pcreate`):

```
[paste.paster_create_template]
coolextension=coolextension.scaffolds:CoolExtensionTemplate
[pyramid.scaffold]
coolextension=coolextension.scaffolds:CoolExtensionTemplate
```

Doing this hideousness will allow your scaffold to work as a `paster create` target (under 1.0, 1.1, or 1.2) or as a `pcreate` target (under 1.3). If an invoker tries to run `paster create` against a scaffold defined this way under 1.3, an error is raised instructing them to use `pcreate` instead.

If you want only to support Pyramid 1.3 only, it's much cleaner, and the API is stable:

```
1 from pyramid.scaffolds import PyramidTemplate
2
3 class CoolExtensionTemplate(PyramidTemplate):
4     _template_dir = 'coolextension_scaffold'
5     summary = 'My cool_extension'
```

You only need to specify a `paste.paster_create_template` entry point target in your `setup.py` if you want your scaffold to be consumable by users of Pyramid 1.0, 1.1, or 1.2. To support only 1.3, specifying only the `pyramid.scaffold` entry point is good enough. If you want to support both `paster create` and `pcreate` (meaning you want to support Pyramid 1.2 and some older version), you'll need to define both.

36.3 Examples

Existing third-party distributions which house scaffolding are available via *PyPI*. The `pyramid_jqm`, `pyramid_zcml` and `pyramid_jinja2` packages house scaffolds. You can install and examine these packages to see how they work in the quest to develop your own scaffolding.

Upgrading Pyramid

When a new version of Pyramid is released, it will sometimes deprecate a feature or remove a feature that was deprecated in an older release. When features are removed from Pyramid, applications that depend on those features will begin to break. This chapter explains how to ensure your Pyramid applications keep working when you upgrade the Pyramid version you’re using.

About Release Numbering

Conventionally, application version numbering in Python is described as `major.minor.micro`. If your Pyramid version is “1.2.3”, it means you’re running a version of Pyramid with the major version “1”, the minor version “2” and the micro version “3”. A “major” release is one that increments the first-dot number; 2.X.X might follow 1.X.X. A “minor” release is one that increments the second-dot number; 1.3.X might follow 1.2.X. A “micro” release is one that increments the third-dot number; 1.2.3 might follow 1.2.2. In general, micro releases are “bugfix-only”, and contain no new features, minor releases contain new features but are largely backwards compatible with older versions, and a major release indicates a large set of backwards incompatibilities.

The Pyramid core team is conservative when it comes to removing features. We don’t remove features unnecessarily, but we’re human, and we make mistakes which cause some features to be evolutionary dead ends. Though we are willing to support dead-end features for some amount of time, some eventually have to be removed when the cost of supporting them outweighs the benefit of keeping them around, because each feature in Pyramid represents a certain documentation and maintenance burden.

37.1 Deprecation and Removal Policy

When a feature is scheduled for removal from Pyramid or any of its official add-ons, the core development team takes these steps:

- Using the feature will begin to generate a *DeprecationWarning*, indicating the version in which the feature became deprecated.
- A note is added to the documentation indicating that the feature is deprecated.
- A note is added to the *changelog* about the deprecation.

When a deprecated feature is eventually removed:

- The feature is removed.
- A note is added to the *changelog* about the removal.

Features are never removed in *micro* releases. They are only removed in minor and major releases. Deprecated features are kept around for at least *three* minor releases from the time the feature became deprecated. Therefore, if a feature is added in Pyramid 1.0, but it's deprecated in Pyramid 1.1, it will be kept around through all 1.1.X releases, all 1.2.X releases and all 1.3.X releases. It will finally be removed in the first 1.4.X release.

Sometimes features are “docs-deprecated” instead of formally deprecated. This means that the feature will be kept around indefinitely, but it will be removed from the documentation or a note will be added to the documentation telling folks to use some other newer feature. This happens when the cost of keeping an old feature around is very minimal and the support and documentation burden is very low. For example, we might rename a function that is an API without changing the arguments it accepts. In this case, we'll often rename the function, and change the docs to point at the new function name, but leave around a backwards compatibility alias to the old function name so older code doesn't break.

“Docs deprecated” features tend to work “forever”, meaning that they won't be removed, and they'll never generate a deprecation warning. However, such changes are noted in the *changelog*, so it's possible to know that you should change older spellings to newer ones to ensure that people reading your code can find the APIs you're using in the Pyramid docs.

37.2 Consulting the Change History

Your first line of defense against application failures caused by upgrading to a newer Pyramid release is always to read the *changelog*, to find the deprecations and removals for each release between the release you're currently running and the one you wish to upgrade to. The change history notes every deprecation within a `Deprecation` section and every removal within a `Backwards Incompatibilities` section for each release.

The change history often contains instructions for changing your code to avoid deprecation warnings and how to change docs-deprecated spellings to newer ones. You can follow along with each deprecation explanation in the change history, simply doing a `grep` or other code search to your application, using the change log examples to remediate each potential problem.

37.3 Testing Your Application Under a New Pyramid Release

Once you've upgraded your application to a new Pyramid release and you've remediated as much as possible by using the change history notes, you'll want to run your application's tests (see *Run the Tests*) in such a way that you can see `DeprecationWarnings` printed to the console when the tests run.

```
$ python -Wd setup.py test -q
```

The `-Wd` argument is an argument that tells Python to print deprecation warnings to the console. Note that the `-Wd` flag is only required for Python 2.7 and better: Python versions 2.6 and older print deprecation warnings to the console by default. See the `Python -W` flag documentation for more information.

As your tests run, deprecation warnings will be printed to the console explaining the deprecation and providing instructions about how to prevent the deprecation warning from being issued. For example:

```
$ python -Wd setup.py test -q
# .. elided ...
running build_ext
/home/chris/projects/pyramid/env27/myproj/myproj/views.py:3:
DeprecationWarning: static: The "pyramid.view.static" class is deprecated
as of Pyramid 1.1; use the "pyramid.static.static_view" class instead with
the "use_subpath" argument set to True.
    from pyramid.view import static
.
-----
Ran 1 test in 0.014s

OK
```

37. UPGRADING PYRAMID

In the above case, it's line #3 in the `myproj.views` module (`from pyramid.view import static`) that is causing the problem:

```
1 from pyramid.view import view_config
2
3 from pyramid.view import static
4 myview = static('static', 'static')
```

The deprecation warning tells me how to fix it, so I can change the code to do things the newer way:

```
1 from pyramid.view import view_config
2
3 from pyramid.static import static_view
4 myview = static_view('static', 'static', use_subpath=True)
```

When I run the tests again, the deprecation warning is no longer printed to my console:

```
$ python -Wd setup.py test -q
# .. elided ...
running build_ext
.
-----
Ran 1 test in 0.014s

OK
```

37.4 My Application Doesn't Have Any Tests or Has Few Tests

If your application has no tests, or has only moderate test coverage, running tests won't tell you very much, because the Pyramid codepaths that generate deprecation warnings won't be executed.

In this circumstance, you can start your application interactively under a server run with the `PYTHONWARNINGS` environment variable set to default. On UNIX, you can do that via:

```
$ PYTHONWARNINGS=default $VENV/bin/pserve development.ini
```

On Windows, you need to issue two commands:

```
C:\> set PYTHONWARNINGS=default
C:\> Scripts/pserve.exe development.ini
```

At this point, it's ensured that deprecation warnings will be printed to the console whenever a codepath is hit that generates one. You can then click around in your application interactively to try to generate them, and remediate as explained in *Testing Your Application Under a New Pyramid Release*.

See the PYTHONWARNINGS environment variable documentation or the Python -W flag documentation for more information.

37.5 Upgrading to the Very Latest Pyramid Release

When you upgrade your application to the most recent Pyramid release, it's advisable to upgrade stepwise through each most recent minor release, beginning with the one that you know your application currently runs under, and ending on the most recent release. For example, if your application is running in production on Pyramid 1.2.1, and the most recent Pyramid 1.3 release is Pyramid 1.3.3, and the most recent Pyramid release is 1.4.4, it's advisable to do this:

- Upgrade your environment to the most recent 1.2 release. For example, the most recent 1.2 release might be 1.2.3, so upgrade to it. Then run your application's tests under 1.2.3 as described in *Testing Your Application Under a New Pyramid Release*. Note any deprecation warnings and remediate.
- Upgrade to the most recent 1.3 release, 1.3.3. Run your application's tests, note any deprecation warnings and remediate.
- Upgrade to 1.4.4. Run your application's tests, note any deprecation warnings and remediate.

If you skip testing your application under each minor release (for example if you upgrade directly from 1.2.1 to 1.4.4), you might miss a deprecation warning and waste more time trying to figure out an error caused by a feature removal than it would take to upgrade stepwise through each minor release.

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”.

38.1 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()` (deprecated as of 1.5) 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.

38.2 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 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.

39.1 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 *PasteDeploy* “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 Pyramid application at `/pyramid`, 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 Pyramid 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.

39.1.1 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)`:

```
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()`.

39.1.2 Enabling the ZCA Global API by Using `hook_zca`

Consider the following bit of idiomatic Pyramid startup code:

```
1 from pyramid.config import Configurator
2
3 def app(global_settings, **settings):
4     config = Configurator(settings=settings)
5     config.include('some.other.package')
6     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.

39. 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” Pyramid registry, you need to call `hook_zca()` within your setup code. For example:

```
1 from pyramid.config import Configurator
2
3 def app(global_settings, **settings):
4     config = Configurator(settings=settings)
5     config.hook_zca()
6     config.include('some.other.application')
7     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*.

39.1.3 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.include('some.other.application')
9     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 Pyramid-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

SQLAlchemy + URL Dispatch Wiki Tutorial

This tutorial introduces a *SQLAlchemy* and *url dispatch* -based Pyramid application to a developer familiar with Python. 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 on GitHub at [docs/tutorials/wiki2/src](https://github.com/PyramidFoss/docs/tutorials/wiki2/src), which corresponds to the same location if you have Pyramid sources.

40.1 Background

This tutorial presents a Pyramid application that uses technologies which will be familiar to someone with SQL database experience. It uses *SQLAlchemy* as a persistence mechanism and *url dispatch* to map URLs to code.

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* a Windows system of any kind.



This tutorial runs on both Python 2 and 3 without modification.

Have fun!

40.2 Design

Following is a quick overview of the design of our wiki application, to help us understand the changes that we will be making as we work through the tutorial.

40.2.1 Overall

We choose to use *reStructuredText* markup in the wiki text. Translation from reStructuredText to HTML is provided by the widely used `docutils` Python module. We will add this module in the dependency list on the project `setup.py` file.

40.2.2 Models

We'll be using a SQLite database to hold our wiki data, and we'll be using *SQLAlchemy* to access the data in this database.

Within the database, we define a single table named *pages*, whose elements will store the wiki pages. There are two columns: *name* and *data*.

URLs like `/PageName` will try to find an element in the table that has a corresponding name.

To add a page to the wiki, a new row is created and the text is stored in *data*.

A page named `FrontPage` containing the text *This is the front page*, will be created when the storage is initialized, and will be used as the wiki home page.

40.2.3 Views

There will be three views to handle the normal operations of adding, editing and viewing wiki pages, plus one view for the wiki front page. Two templates will be used, one for viewing, and one for both for adding and editing wiki pages.

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.

40.2.4 Security

We'll eventually be adding security to our application. The components we'll use to do this are below.

- `USERS`, a dictionary mapping users names to their corresponding passwords.
- `GROUPS`, a dictionary mapping user names to a list of groups they belong to.
- `groupfinder`, an *authorization callback* that looks up `USERS` and `GROUPS`. It will be provided in a new *security.py* file.
- An *ACL* is attached to the root *resource*. Each row below details an *ACE*:

| Action | Principal | Permission |
|--------|---------------|------------|
| Allow | Everyone | View |
| Allow | group:editors | Edit |

- Permission declarations are added to the views to assert the security policies as each request is handled.

Two additional views and one template will handle the login and logout tasks.

40.2.5 Summary

The URL, actions, template and permission associated to each view are listed in the following table:

| URL | Action | View | Template | Permission |
|---------------------|--|------------------------|----------|------------|
| / | Redirect to /Front-Page | view_wiki | | |
| /PageName | Display existing page ¹ | view_page ² | view.pt | view |
| /PageName/edit_page | Display edit form with existing content. If the form is submitted, redirect to /PageName | edit_page | edit.pt | edit |
| /add_page/PageName | Create the page <i>PageName</i> in storage, display the edit form without content. If the form is submitted, redirect to /PageName | add_page | edit.pt | edit |
| /login | Display login form, Forbidden ³ If the form is submitted, authenticate. <ul style="list-style-type: none">• If authentication successful, redirect to the page that we came from.• If authentication fails, display login form with “login failed” message. | login | login.pt | |
| /logout | Redirect to /Front-Page | logout | | |

40.3 Installation

40.3.1 Before You Begin

This tutorial assumes that you have already followed the steps in *Installing Pyramid*, thereby satisfying the following requirements.

- Python interpreter is installed on your operating system
- *setuptools* or *distribute* is installed
- *virtualenv* is installed

Create and Use a Virtual Python Environment

Next let's create a *virtualenv* workspace for our project. We will use the *VENV* environment variable instead of absolute path of the virtual environment.

On UNIX

```
$ export VENV=~/.pyramiddtut
$ virtualenv $VENV
New python executable in /home/foo/env/bin/python
Installing setuptools.....done.
```

On Windows

Set the *VENV* environment variable.

```
c:\> set VENV=c:\pyramiddtut
```

Versions of Python use different paths, so you will need to adjust the path to the command for your Python version.

Python 2.7:

40. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
c:\> c:\Python27\Scripts\virtualenv %VENV%
```

Python 3.2:

```
c:\> c:\Python32\Scripts\virtualenv %VENV%
```

Install Pyramid Into the Virtual Python Environment

On UNIX

```
$ $VENV/bin/easy_install pyramid
```

On Windows

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

Install SQLite3 and Its Development Packages

If you used a package manager to install your Python or if you compiled your Python from source, then you must install SQLite3 and its development packages. If you downloaded your Python as an installer from python.org, then you already have it installed and can proceed to the next section *Making a Project..*

If you need to install the SQLite3 packages, then, for example, using the Debian system and apt-get, the command would be the following:

```
$ sudo apt-get install libsqlite3-dev
```

Change Directory to Your Virtual Python Environment

Change directory to the `pyramidtut` directory.

On UNIX

```
$ cd pyramidtut
```

On Windows

```
c:\> cd pyramidtut
```

40.3.2 Making a Project

Your next step is to create a project. For this tutorial we will use the *scaffold* named *alchemy* which generates an application that uses *SQLAlchemy* and *URL dispatch*.

Pyramid supplies a variety of scaffolds to generate sample projects. We will use *pcreate*—a script that comes with Pyramid to quickly and easily generate scaffolds usually with a single command—to create the scaffold for our project.

By passing in *alchemy* into the *pcreate* command, the script creates the files needed to use *SQLAlchemy*. By passing in our application name *tutorial*, the script inserts that application name into all the required files. For example, *pcreate* creates the *initialize_tutorial_db* in the *pyramidtut/bin* directory.

The below instructions assume your current working directory is the “virtualenv” named “pyramidtut”.

On UNIX

```
$ $VENV/bin/pcreate -s alchemy tutorial
```

On Windows

```
c:\pyramidtut> %VENV%\pcreate -s alchemy tutorial
```

i If you are using Windows, the *alchemy* scaffold 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.

40.3.3 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 the virtualenv Python interpreter.

On UNIX

```
$ cd tutorial
$ $VENV/bin/python setup.py develop
```

On Windows

```
c:\pyramidtut> cd tutorial
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py develop
```

The console will show *setup.py* checking for packages and installing missing packages. Success executing this command will show a line like the following:

```
Finished processing dependencies for tutorial==0.0
```

40.3.4 Running the Tests

After you’ve installed the project in development mode, you may run the tests for the project.

On UNIX

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

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py test -q
```

For a successful test run, you should see output that ends like this:

```
.
-----
Ran 1 test in 0.094s
OK
```

40.3.5 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 the `nose` and `coverage` packages into our `virtualenv`:

On UNIX

```
$ $VENV/bin/easy_install nose coverage
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\easy_install nose coverage
```

Once `nose` and `coverage` are installed, we can actually run the coverage tests.

On UNIX

40. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
$ $VENV/bin/nosetests --cover-package=tutorial --cover-erase --with-coverage
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\nosetests --cover-package=tutorial \
    --cover-erase --with-coverage
```

If successful, you will see output something like this:

```
.
Name                Stmts   Miss  Cover   Missing
-----
tutorial             11      7    36%    9-15
tutorial.models      17      0   100%
tutorial.scripts      0      0   100%
tutorial.tests       24      0   100%
tutorial.views        6      0   100%
-----
TOTAL                58      7    88%
-----
Ran 1 test in 0.459s

OK
```

Looks like our package doesn't quite have 100% test coverage.

40.3.6 Initializing the Database

We need to use the `initialize_tutorial_db console script` to initialize our database.

Type the following command, make sure you are still in the `tutorial` directory (the directory with a `development.ini` in it):

On UNIX

```
$ $VENV/bin/initialize_tutorial_db development.ini
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\initialize_tutorial_db development.ini
```

The output to your console should be something like this:

```
2011-11-26 14:42:25,012 INFO [sqlalchemy.engine.base.Engine] [MainThread]
PRAGMA table_info("models")
2011-11-26 14:42:25,013 INFO [sqlalchemy.engine.base.Engine] [MainThread] (
2011-11-26 14:42:25,013 INFO [sqlalchemy.engine.base.Engine] [MainThread]
CREATE TABLE models (
    id INTEGER NOT NULL,
    name VARCHAR(255),
    value INTEGER,
    PRIMARY KEY (id),
    UNIQUE (name)
)
2011-11-26 14:42:25,013 INFO [sqlalchemy.engine.base.Engine] [MainThread] (
2011-11-26 14:42:25,135 INFO [sqlalchemy.engine.base.Engine] [MainThread]
COMMIT
2011-11-26 14:42:25,137 INFO [sqlalchemy.engine.base.Engine] [MainThread]
BEGIN (implicit)
2011-11-26 14:42:25,138 INFO [sqlalchemy.engine.base.Engine] [MainThread]
INSERT INTO models (name, value) VALUES (?, ?)
2011-11-26 14:42:25,139 INFO [sqlalchemy.engine.base.Engine] [MainThread]
(u'one', 1)
2011-11-26 14:42:25,140 INFO [sqlalchemy.engine.base.Engine] [MainThread]
COMMIT
```

Success! You should now have a `tutorial.sqlite` file in your current working directory. This will be a SQLite database with a single table defined in it (`models`).

40.3.7 Starting the Application

Start the application.

On UNIX

```
$ $VENV/bin/pserve development.ini --reload
```

On Windows

```
c:\pyramidtut\tutorial> %VENV%\Scripts\pserve development.ini --reload
```

If successful, you will see something like this on your console:

```
Starting subprocess with file monitor
Starting server in PID 8966.
Starting HTTP server on http://0.0.0.0:6543
```

This means the server is ready to accept requests.

At this point, when you visit `http://localhost:6543/` in your web browser, you will see the generated application's default page.

One thing you'll notice is the “debug toolbar” icon on right hand side of the page. You can read more about the purpose of the icon at *The Debug Toolbar*. It allows you to get information about your application while you develop.

40.3.8 Decisions the alchemy Scaffold Has Made For You

Creating a project using the alchemy scaffold 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 use *ZopeTransactionExtension* and *pyramid_tm* to scope sessions to requests



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.

40.4 Basic Layout

The starter files generated by the `alchemy` scaffold are very basic, but they provide a good orientation for the high-level patterns common to most *url dispatch*-based Pyramid projects.

40.4.1 Application Configuration 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 marker indicating the directory it's contained within is a package, and to contain configuration code.

Open `tutorial/tutorial/__init__.py`. It should already contain the following:

```

1  from pyramid.config import Configurator
2  from sqlalchemy import engine_from_config
3
4  from .models import (
5      DBSession,
6      Base,
7  )
8
9
10 def main(global_config, **settings):
11     """ This function returns a Pyramid WSGI application.
12     """
13     engine = engine_from_config(settings, 'sqlalchemy.')
14     DBSession.configure(bind=engine)
15     Base.metadata.bind = engine
16     config = Configurator(settings=settings)
17     config.include('pyramid_chameleon')
18     config.add_static_view('static', 'static', cache_max_age=3600)
19     config.add_route('home', '/')
20     config.scan()
21     return config.make_wsgi_app()
```

Let's go over this piece-by-piece. First, we need some imports to support later code:

```

1  from pyramid.config import Configurator
2  from sqlalchemy import engine_from_config
3
4  from .models import (
5      DBSession,
6      Base,
7  )
```

`__init__.py` defines a function named `main`. Here is the entirety of the `main` function we've defined in our `__init__.py`:

```
1 def main(global_config, **settings):
2     """ This function returns a Pyramid WSGI application.
3     """
4     engine = engine_from_config(settings, 'sqlalchemy.')
5     DBSession.configure(bind=engine)
6     Base.metadata.bind = engine
7     config = Configurator(settings=settings)
8     config.include('pyramid_chameleon')
9     config.add_static_view('static', 'static', cache_max_age=3600)
10    config.add_route('home', '/')
11    config.scan()
12    return config.make_wsgi_app()
```

When you invoke the `pserve development.ini` command, the `main` function above is executed. It accepts some settings and returns a *WSGI* application. (See *Startup* for more about `pserve`.)

The `main` function first creates a *SQLAlchemy* database engine using `sqlalchemy.engine_from_config()` from the `sqlalchemy.` prefixed settings in the `development.ini` file's `[app:main]` section. This will be a URI (something like `sqlite://`):

```
engine = engine_from_config(settings, 'sqlalchemy.')
```

`main` then initializes our *SQLAlchemy* session object, passing it the engine:

```
DBSession.configure(bind=engine)
```

`main` subsequently initializes our *SQLAlchemy* declarative `Base` object, assigning the engine we created to the `bind` attribute of it's metadata object. This allows table definitions done imperatively (instead of declaratively, via a class statement) to work. We won't use any such tables in our application, but if you add one later, long after you've forgotten about this tutorial, you won't be left scratching your head when it doesn't work.

```
Base.metadata.bind = engine
```

The next step of `main` is to construct a *Configurator* object:

```
config = Configurator(settings=settings)
```

`settings` is passed to the Configurator as a keyword argument with the dictionary values passed as the `**settings` argument. This will be a dictionary of settings parsed from the `.ini` file, which contains deployment-related values such as `pyramid.reload_templates`, `db_string`, etc.

Next, include *Chameleon* templating bindings so that we can use renderers with the `.pt` extension within our project.

```
config.include('pyramid_chameleon')
```

`main` now calls `pyramid.config.Configurator.add_static_view()` with two arguments: `static` (the name), and `static` (the path):

```
config.add_static_view('static', 'static', cache_max_age=3600)
```

This registers a static resource view which will match any URL that starts with the prefix `/static` (by virtue of the first argument to `add_static_view`). 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 (by virtue of the second argument to `add_static_view`). 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.

Using the configurator `main` also registers a *route configuration* via the `pyramid.config.Configurator.add_route()` method that will be used when the URL is `/`:

```
config.add_route('home', '/')
```

Since this route has a pattern equalling `/` it is the route that will be matched when the URL `/` is visited, e.g. `http://localhost:6543/`.

`main` next calls the `scan` method of the configurator (`pyramid.config.Configurator.scan()`), which will recursively scan our tutorial package, looking for `@view_config` (and other special) decorators. When it finds a `@view_config` decorator, a view configuration will be registered, which will allow one of our application URLs to be mapped to some code.

```
config.scan()
```

Finally, `main` is finished configuring things, so it uses the `pyramid.config.Configurator.make_wsgi_app()` method to return a *WSGI* application:

```
return config.make_wsgi_app()
```

40.4.2 View Declarations via `views.py`

The main function of a web framework is mapping each URL pattern to code (a *view callable*) that is executed when the requested URL matches the corresponding *route*. Our application uses the `pyramid.view.view_config()` decorator to perform this mapping.

Open `tutorial/tutorial/views.py`. It should already contain the following:

```
1 from pyramid.response import Response
2 from pyramid.view import view_config
3
4 from sqlalchemy.exc import DBAPIError
5
6 from .models import (
7     DBSession,
8     MyModel,
9 )
10
11
12 @view_config(route_name='home', renderer='templates/mytemplate.pt')
13 def my_view(request):
14     try:
15         one = DBSession.query(MyModel).filter(MyModel.name == 'one').first()
16     except DBAPIError:
17         return Response(conn_err_msg, content_type='text/plain', status_int=500)
18     return {'one': one, 'project': 'tutorial'}
19
20 conn_err_msg = """\
21 Pyramid is having a problem using your SQL database. The problem
22 might be caused by one of the following things:
23
24 1. You may need to run the "initialize_tutorial_db" script
25    to initialize your database tables. Check your virtual
26    environment's "bin" directory for this script and try to run it.
27
28 2. Your database server may not be running. Check that the
29    database server referred to by the "sqlalchemy.url" setting in
30    your "development.ini" file is running.
31
32 After you fix the problem, please restart the Pyramid application to
```

```

33 | try it again.
34 | """

```

The important part here is that the `@view_config` decorator associates the function it decorates (`my_view`) with a *view configuration*, consisting of:

- a `route_name` (`home`)
- a `renderer`, which is a template from the `templates` subdirectory of the package.

When the pattern associated with the `home` view is matched during a request, `my_view()` will be executed. `my_view()` returns a dictionary; the `renderer` will use the `templates/mytemplate.pt` template to create a response based on the values in the dictionary.

Note that `my_view()` accepts a single argument named `request`. This is the standard call signature for a Pyramid *view callable*.

Remember in our `__init__.py` when we executed the `pyramid.config.Configurator.scan()` method, i.e. `config.scan()`? The purpose of calling the `scan` method was to find and process this `@view_config` decorator in order to create a view configuration within our application. Without being processed by `scan`, the decorator effectively does nothing. `@view_config` is inert without being detected via a *scan*.

The sample `my_view()` created by the scaffold uses a `try:` and `except:` clause, to detect if there is a problem accessing the project database and provide an alternate error response. That response will include the text shown at the end of the file, which will be displayed in the browser to inform the user about possible actions to take to solve the problem.

40.4.3 Content Models with `models.py`

In a SQLAlchemy-based application, a *model* object is an object composed by querying the SQL database. The `models.py` file is where the `alchemy` scaffold put the classes that implement our models.

Open `tutorial/tutorial/models.py`. It should already contain the following:

```
1 from sqlalchemy import (
2     Column,
3     Integer,
4     Text,
5 )
6
7 from sqlalchemy.ext.declarative import declarative_base
8
9 from sqlalchemy.orm import (
10     scoped_session,
11     sessionmaker,
12 )
13
14 from zope.sqlalchemy import ZopeTransactionExtension
15
16 DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
17 Base = declarative_base()
18
19
20 class MyModel(Base):
21     __tablename__ = 'models'
22     id = Column(Integer, primary_key=True)
23     name = Column(Text, unique=True)
24     value = Column(Integer)
```

Let's examine this in detail. First, we need some imports to support later code:

```
1 from sqlalchemy import (
2     Column,
3     Integer,
4     Text,
5 )
6
7 from sqlalchemy.ext.declarative import declarative_base
8
9 from sqlalchemy.orm import (
10     scoped_session,
11     sessionmaker,
12 )
13
14 from zope.sqlalchemy import ZopeTransactionExtension
```

Next we set up a SQLAlchemy DBSession object:

```
DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
```

`scoped_session` and `sessionmaker` are standard SQLAlchemy helpers. `scoped_session` allows us to access our database connection globally. `sessionmaker` creates a database session object. We pass to `sessionmaker` the `extension=ZopeTransactionExtension()` extension option in order to allow the system to automatically manage database transactions. With `ZopeTransactionExtension` activated, our application will automatically issue a transaction commit after every request unless an exception is raised, in which case the transaction will be aborted.

We also need to create a declarative Base object to use as a base class for our model:

```
Base = declarative_base()
```

Our model classes will inherit from this Base class so they can be associated with our particular database connection.

To give a simple example of a model class, we define one named `MyModel`:

```
1 class MyModel(Base):
2     __tablename__ = 'models'
3     id = Column(Integer, primary_key=True)
4     name = Column(Text, unique=True)
5     value = Column(Integer)
```

Our example model does not require an `__init__` method because SQLAlchemy supplies for us a default constructor if one is not already present, which accepts keyword arguments of the same name as that of the mapped attributes.



Example usage of `MyModel`:

```
johnny = MyModel(name="John Doe", value=10)
```

The `MyModel` class has a `__tablename__` attribute. This informs SQLAlchemy which table to use to store the data representing instances of this class.

That's about all there is to it regarding models, views, and initialization code in our stock application.

40.5 Defining the Domain Model

The first change we'll make to our stock `pcreate`-generated application will be to define a *domain model* constructor representing a wiki page. We'll do this inside our `models.py` file.

40.5.1 Making Edits to `models.py`



There is nothing 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.

Open `tutorial/tutorial/models.py` file and edit it to look like the following:

```
1 from sqlalchemy import (
2     Column,
3     Integer,
4     Text,
5 )
6
7 from sqlalchemy.ext.declarative import declarative_base
8
9 from sqlalchemy.orm import (
10     scoped_session,
11     sessionmaker,
12 )
13
14 from zope.sqlalchemy import ZopeTransactionExtension
15
16 DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
17 Base = declarative_base()
18
19
20 class Page(Base):
21     """ The SQLAlchemy declarative model class for a Page object. """
22     __tablename__ = 'pages'
23     id = Column(Integer, primary_key=True)
24     name = Column(Text, unique=True)
25     data = Column(Text)
```

(The highlighted lines are the ones that need to be changed.)

The first thing we've done 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 added a `Page` class. Because this is a SQLAlchemy application, this class inherits from an instance of `sqlalchemy.ext.declarative.declarative_base()`.

```
1 class Page(Base):
2     """ The SQLAlchemy declarative model class for a Page object. """
3     __tablename__ = 'pages'
4     id = Column(Integer, primary_key=True)
5     name = Column(Text, unique=True)
6     data = Column(Text)
```

As you can see, our `Page` class has 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`, `name` and `data` (all instances of `sqlalchemy.schema.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.

40.5.2 Changing `scripts/initializedb.py`

We haven't looked at the details of this file yet, but within the `scripts` directory of your tutorial package is a file named `initializedb.py`. Code in this file is executed whenever we run the `initialize_tutorial_db` command (as we did in the installation step of this tutorial).

Since we've changed our model, we need to make changes to our `initializedb.py` script. In particular, we'll replace our import of `MyModel` with one of `Page` and we'll change the very end of the script to create a `Page` rather than a `MyModel` and add it to our `DBSession`.

Open `tutorial/tutorial/scripts/initializedb.py` and edit it to look like the following:

```
1 import os
2 import sys
3 import transaction
4
5 from sqlalchemy import engine_from_config
6
7 from pyramid.paster import (
8     get_appsettings,
```

```
9     setup_logging,
10 )
11
12 from ..models import (
13     DBSession,
14     Page,
15     Base,
16 )
17
18
19 def usage(argv):
20     cmd = os.path.basename(argv[0])
21     print('usage: %s <config_uri>\n'
22           '(example: "%s development.ini")' % (cmd, cmd))
23     sys.exit(1)
24
25
26 def main(argv=sys.argv):
27     if len(argv) != 2:
28         usage(argv)
29     config_uri = argv[1]
30     setup_logging(config_uri)
31     settings = get_appsettings(config_uri)
32     engine = engine_from_config(settings, 'sqlalchemy.')
33     DBSession.configure(bind=engine)
34     Base.metadata.create_all(engine)
35     with transaction.manager:
36         model = Page(name='FrontPage', data='This is the front page')
37         DBSession.add(model)
```

(Only the highlighted lines need to be changed.)

40.5.3 Installing the Project and re-initializing the Database

Because our model has changed, in order to reinitialize the database, we need to rerun the `initialize_tutorial_db` command to pick up the changes you've made to both the `models.py` file and to the `initializedb.py` file. See *Initializing the Database* for instructions.

Success will look something like this:

```
2011-11-27 01:22:45,277 INFO [sqlalchemy.engine.base.Engine][MainThread]
PRAGMA table_info("pages")
2011-11-27 01:22:45,277 INFO [sqlalchemy.engine.base.Engine][MainThread] ()
```

```

2011-11-27 01:22:45,277 INFO [sqlalchemy.engine.base.Engine][MainThread]
CREATE TABLE pages (
    id INTEGER NOT NULL,
    name TEXT,
    data TEXT,
    PRIMARY KEY (id),
    UNIQUE (name)
)

2011-11-27 01:22:45,278 INFO [sqlalchemy.engine.base.Engine][MainThread] (
2011-11-27 01:22:45,397 INFO [sqlalchemy.engine.base.Engine][MainThread]
COMMIT
2011-11-27 01:22:45,400 INFO [sqlalchemy.engine.base.Engine][MainThread]
BEGIN (implicit)
2011-11-27 01:22:45,401 INFO [sqlalchemy.engine.base.Engine][MainThread]
INSERT INTO pages (name, data) VALUES (?, ?)
2011-11-27 01:22:45,401 INFO [sqlalchemy.engine.base.Engine][MainThread]
('FrontPage', 'This is the front page')
2011-11-27 01:22:45,402 INFO [sqlalchemy.engine.base.Engine][MainThread]
COMMIT

```

40.5.4 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 (See *Starting 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.

40.6 Defining Views

A *view callable* in a 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.

The request object has a dictionary as an attribute named *matchdict*. A *matchdict* maps the placeholders in the matching URL pattern to the substrings of the path in the *request* URL. For instance, if a

call to `pyramid.config.Configurator.add_route()` has the pattern `{one}/{two}`, and a user visits `http://example.com/foo/bar`, our pattern would be matched against `/foo/bar` and the `matchdict` would look like: `{ 'one': 'foo', 'two': 'bar' }`

40.6.1 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 `pcreate` 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 `requires` parameter in `setup()`.

Open `tutorial/setup.py` and edit it to look like the following:

```
1 import os
2
3 from setuptools import setup, find_packages
4
5 here = os.path.abspath(os.path.dirname(__file__))
6 with open(os.path.join(here, 'README.txt')) as f:
7     README = f.read()
8 with open(os.path.join(here, 'CHANGES.txt')) as f:
9     CHANGES = f.read()
10
11 requires = [
12     'pyramid',
13     'pyramid_chameleon',
14     'pyramid_debugtoolbar',
15     'pyramid_tm',
16     'SQLAlchemy',
17     'transaction',
18     'zope.sqlalchemy',
19     'waitress',
20     'docutils',
21 ]
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 :: Pyramid",
```

```

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 [console_scripts]
46 initialize_tutorial_db = tutorial.scripts.initializedb:main
47 """,
48 )

```

(Only the highlighted line needs to be added.)

40.6.2 Running `setup.py develop`

Since a new software dependency was 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 distribution.

Make sure your current working directory is the root of the project (the directory in which `setup.py` lives) and execute the following command.

On UNIX:

```

$ cd tutorial
$ $VENV/bin/python setup.py develop

```

On Windows:

```

c:\pyramidtut> cd tutorial
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py develop

```

Success executing this command will end with a line to the console something like:

Finished processing dependencies for tutorial==0.0

40.6.3 Changing the `views.py` File

It's time for a major change. Open `tutorial/tutorial/views.py` and edit it to look like the following:

```
1 import cgi
2 import re
3 from docutils.core import publish_parts
4
5 from pyramid.httpexceptions import (
6     HTTPFound,
7     HTTPNotFound,
8 )
9 from pyramid.view import view_config
10
11 from .models import (
12     DBSession,
13     Page,
14 )
15
16 # regular expression used to find WikiWords
17 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+)" )
18
19 @view_config(route_name='view_wiki')
20 def view_wiki(request):
21     return HTTPFound(location = request.route_url('view_page',
22                                                    pagename='FrontPage'))
23
24 @view_config(route_name='view_page', renderer='templates/view.pt')
25 def view_page(request):
26     pagename = request.matchdict['pagename']
27     page = DBSession.query(Page).filter_by(name=pagename).first()
28     if page is None:
29         return HTTPNotFound('No such page')
30
31     def check(match):
32         word = match.group(1)
33         exists = DBSession.query(Page).filter_by(name=word).all()
34         if exists:
35             view_url = request.route_url('view_page', pagename=word)
36             return '<a href="%s">%s</a>' % (view_url, cgi.escape(word))
```

```

37         else:
38             add_url = request.route_url('add_page', pagename=word)
39             return '<a href="%s">%s</a>' % (add_url, cgi.escape(word))
40
41         content = publish_parts(page.data, writer_name='html')['html_body']
42         content = wikiwords.sub(check, content)
43         edit_url = request.route_url('edit_page', pagename=pagename)
44         return dict(page=page, content=content, edit_url=edit_url)
45
46 @view_config(route_name='add_page', renderer='templates/edit.pt')
47 def add_page(request):
48     pagename = request.matchdict['pagename']
49     if 'form.submitted' in request.params:
50         body = request.params['body']
51         page = Page(name=pagename, data=body)
52         DBSession.add(page)
53         return HTTPFound(location = request.route_url('view_page',
54                                                         pagename=pagename))
55
56     save_url = request.route_url('add_page', pagename=pagename)
57     page = Page(name='', data='')
58     return dict(page=page, save_url=save_url)
59
60 @view_config(route_name='edit_page', renderer='templates/edit.pt')
61 def edit_page(request):
62     pagename = request.matchdict['pagename']
63     page = DBSession.query(Page).filter_by(name=pagename).one()
64     if 'form.submitted' in request.params:
65         page.data = request.params['body']
66         DBSession.add(page)
67         return HTTPFound(location = request.route_url('view_page',
68                                                         pagename=pagename))
69
70     return dict(
71         page=page,
72         save_url = request.route_url('edit_page', pagename=pagename),
73     )

```

(The highlighted lines are the ones that need to be added or edited.)


We got rid of the `my_view` view function and its decorator that was added when we originally rendered the alchemy scaffold. It was only an example and isn't relevant to our application.

Then we added four *view callable* functions to our `views.py` module:

- `view_wiki()` - Displays the wiki itself. It will answer on the root URL.
- `view_page()` - Displays an individual page.

- `add_page()` - Allows the user to add a page.
- `edit_page()` - Allows the user to edit a page.

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

`view_wiki()` is the *default view* that gets called when a request is made to the root URL of our wiki. It always redirects to a URL which represents the path to our “FrontPage”.

```
1 @view_config(route_name='view_wiki')
2 def view_wiki(request):
3     return HTTPFound(location = request.route_url('view_page',
```

`view_wiki()` returns an instance of the `pyramid.httpexceptions.HTTPFound` class (instances of which implement the `pyramid.interfaces.IResponse` interface like `pyramid.response.Response` does).

It uses the `pyramid.request.Request.route_url()` API to construct a URL to the FrontPage page (e.g. `http://localhost:6543/FrontPage`), which is used as the “location” of the `HTTPFound` response, forming an HTTP redirect.

The `view_page` view function

`view_page()` is used to display a single page of our wiki. It renders the *reStructuredText* body of a page (stored as the `data` attribute of a `Page` model object) as HTML. Then it substitutes an HTML anchor for each *WikiWord* reference in the rendered HTML using a compiled regular expression.

```

1 @view_config(route_name='view_page', renderer='templates/view.pt')
2 def view_page(request):
3     pagename = request.matchdict['pagename']
4     page = DBSession.query(Page).filter_by(name=pagename).first()
5     if page is None:
6         return HTTPNotFound('No such page')
7
8     def check(match):
9         word = match.group(1)
10        exists = DBSession.query(Page).filter_by(name=word).all()
11        if exists:
12            view_url = request.route_url('view_page', pagename=word)
13            return '<a href="%s">%s</a>' % (view_url, cgi.escape(word))
14        else:
15            add_url = request.route_url('add_page', pagename=word)
16            return '<a href="%s">%s</a>' % (add_url, cgi.escape(word))
17
18    content = publish_parts(page.data, writer_name='html')['html_body']
19    content = wikiwords.sub(check, content)
20    edit_url = request.route_url('edit_page', pagename=pagename)

```

The `check()` function 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, `check()` generates a view link to be used as the substitution value and returns it. If the wiki does not already contain a page with the matched WikiWord name, `check()` 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 `view_page()` 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 response. In our case, the renderer used will be the `templates/view.pt` template, as indicated in the `@view_config` decorator that is applied to `view_page()`.

The add_page view function

`add_page()` is 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. `add_page()` 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.

```
1 @view_config(route_name='add_page', renderer='templates/edit.pt')
2 def add_page(request):
3     pagename = request.matchdict['pagename']
4     if 'form.submitted' in request.params:
5         body = request.params['body']
6         page = Page(name=pagename, data=body)
7         DBSession.add(page)
8         return HTTPFound(location = request.route_url('view_page',
9                                                         pagename=pagename))
10    save_url = request.route_url('add_page', pagename=pagename)
11    page = Page(name='', data='')
```

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 value for `'pagename'` in the `matchdict` will be `'SomeName'`.

If the view execution *is* a result of a form submission (i.e. the expression `'form.submitted'` in `request.params` is `True`), we scrape the page body from the form data, create a `Page` object with this page body and the name taken from `matchdict['pagename']`, and save it into the database using `DBSession.add`. We then redirect back to the `view_page` view for the newly created page.

If the view execution is *not* a result of a form submission (i.e. 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 uses as the form post URL during rendering. We're lazy here, so we're going 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` object in order to satisfy the edit form's desire to have *some* page object exposed as page. Pyramid will render the template associated with this view to a response.

The `edit_page` view function

`edit_page()` is 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 `edit_page` view will have a `'pagename'` key matching the name of the page the user wants to edit.

```
1 @view_config(route_name='edit_page', renderer='templates/edit.pt')
2 def edit_page(request):
3     pagename = request.matchdict['pagename']
4     page = DBSession.query(Page).filter_by(name=pagename).one()
5     if 'form.submitted' in request.params:
6         page.data = request.params['body']
```

```

7         DBSession.add(page)
8         return HTTPFound(location = request.route_url('view_page',
9                                                         pagename=pagename))
10    return dict(
11        page=page,
12        save_url = request.route_url('edit_page', pagename=pagename),

```

If the view execution *is* a result of a form submission (i.e. the expression `'form.submitted'` in `request.params` is `True`), the view grabs the body element of the request parameters and sets it as the data attribute of the page object. It then redirects to the `view_page` view of the wiki page.

If the view execution is *not* a result of a form submission (i.e. the expression `'form.submitted'` in `request.params` is `False`), the view simply renders the edit form, passing the page object and a `save_url` which will be used as the action of the generated form.

40.6.4 Adding Templates

The `view_page`, `add_page` and `edit_page` views that we've added reference a *template*. Each template is a *Chameleon ZPT* template. These templates 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

Create `tutorial/tutorial/templates/view.pt` and add the following content:

```

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"></div>
41             </div>
42         </div>
43         <div id="bottom">
44             <div class="bottom">
45                 <div tal:replace="structure content">
46                     Page text goes here.
47                 </div>
48                 <p>
49                     <a tal:attributes="href edit_url" href="">
50                         Edit this page
51                     </a>
52                 </p>
53             </div>
54         </div>
55     </div>
56 </body>
57 </html>
```

This template is used by `view_page()` for displaying a single wiki page. It includes:

- A `div` element that is replaced with the `content` value provided by the view (rows 45-47).

content contains HTML, so the `structure` keyword is used to prevent escaping it (i.e. changing “>” to “>”, etc.)

- A link that points at the “edit” URL which invokes the `edit_page` view for the page being viewed (rows 49-51).

The `edit.pt` Template

Create `tutorial/tutorial/templates/edit.pt` and add the following content:

```

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 Goes
36             Here</span></b><br/>
37         You can return to the
38         <a href="${request.application_url}">FrontPage</a>.<br/>
39     </div>
40     <div id="right" class="app-welcome align-right"></div>
41 </div>
42 </div>
43 <div id="bottom">
44     <div class="bottom">
45         <form action="${save_url}" method="post">
46             <textarea name="body" tal:content="page.data" rows="10"
47                 cols="60"/><br/>
48             <input type="submit" name="form.submitted" value="Save"/>
49         </form>
50     </div>
51 </div>
52 </div>
53 </body>
54 </html>
```

This template is used by `add_page()` and `edit_page()` for adding and editing a wiki page. It displays a page containing a form that includes:

- A 10 row by 60 column `textarea` field named `body` that is filled with any existing page data when it is rendered (rows 46-47).
- A submit button that has the name `form.submitted` (row 48).

The form POSTs back to the “`save_url`” argument supplied by the view (row 45). The view will use the `body` and `form.submitted` values.



Our templates use a `request` object that none of our tutorial views return in their dictionary. `request` is one of several names that are available “by default” in a template when a template renderer is used. See *System Values Used During Rendering* for information about other names that are available by default when a template is used as a renderer.

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__.py` 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.

40.6.5 Adding Routes to `__init__.py`

The `__init__.py` file contains `pyramid.config.Configurator.add_route()` calls which serve to add routes to our application. 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 route named `view_wiki`. It maps to our `view_wiki` view callable by virtue of the `@view_config` attached to the `view_wiki` view function indicating `route_name='view_wiki'`.
2. Add a declaration which maps the pattern `/ {pagename}` to the route named `view_page`. This is the regular view for a page. It maps to our `view_page` view callable by virtue of the `@view_config` attached to the `view_page` view function indicating `route_name='view_page'`.
3. Add a declaration which maps the pattern `/add_page/ {pagename}` to the route named `add_page`. This is the add view for a new page. It maps to our `add_page` view callable by virtue of the `@view_config` attached to the `add_page` view function indicating `route_name='add_page'`.
4. Add a declaration which maps the pattern `/ {pagename}/edit_page` to the route named `edit_page`. This is the edit view for a page. It maps to our `edit_page` view callable by virtue of the `@view_config` attached to the `edit_page` view function indicating `route_name='edit_page'`.

As a result of our edits, the `__init__.py` file should look something like:

```
1 from pyramid.config import Configurator
2 from sqlalchemy import engine_from_config
3
4 from .models import (
5     DBSession,
6     Base,
```

```
7         )
8
9
10 def main(global_config, **settings):
11     """ This function returns a Pyramid WSGI application.
12     """
13     engine = engine_from_config(settings, 'sqlalchemy.')
14     DBSession.configure(bind=engine)
15     Base.metadata.bind = engine
16     config = Configurator(settings=settings)
17     config.include('pyramid_chameleon')
18     config.add_static_view('static', 'static', cache_max_age=3600)
19     config.add_route('view_wiki', '/')
20     config.add_route('view_page', '/{pagename}')
21     config.add_route('add_page', '/add_page/{pagename}')
22     config.add_route('edit_page', '/{pagename}/edit_page')
23     config.scan()
24     return config.make_wsgi_app()
```

(The highlighted lines are the ones that need to be added or edited.)

40.6.6 Viewing the Application in a Browser

We can finally examine our application in a browser (See *Starting the Application*). Launch a browser and visit each of the following URLs, check that the result is as expected:

- `http://localhost:6543` in a browser invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` object.
- `http://localhost:6543/FrontPage` in a browser invokes the `view_page` view of the front page object.
- `http://localhost:6543/FrontPage/edit_page` in a browser invokes the edit view for the front page object.
- `http://localhost:6543/add_page/SomePageName` in a browser invokes the add view for a page.
- To generate an error, visit `http://localhost:6543/foobars/edit_page` which will generate a `NoResultFound: No row was found for one() error`. You'll see an interactive traceback facility provided by `pyramid_debugtoolbar`.

40.7 Adding Authorization

Pyramid provides facilities for *authentication* and *authorization*. We'll make use of both features to provide security to our application. Our application currently allows anyone with access to the server to view, edit, and add pages to our wiki. We'll change that to allow only people who 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.

We will also add a login page and a logout link on all the pages. The login page will be shown when a user is denied access to any of the views that require permission, instead of a default “403 Forbidden” page.

We will implement the access control with the following steps:

- Add users and groups (`security.py`, a new module).
- Add an *ACL* (`models.py` and `__init__.py`).
- Add an *authentication policy* and an *authorization policy* (`__init__.py`).
- Add *permission* declarations to the `edit_page` and `add_page` views (`views.py`).

Then we will add the login and logout feature:

- Add routes for `/login` and `/logout` (`__init__.py`).
- Add login and logout views (`views.py`).
- Add a login template (`login.pt`).
- Make the existing views return a `logged_in` flag to the renderer (`views.py`).
- Add a “Logout” link to be shown when logged in and viewing or editing a page (`view.pt`, `edit.pt`).

40.7.1 Access Control

Add users and groups

Create a new `tutorial/tutorial/security.py` module 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 accepts a `userid` and a `request` and returns one of these values:

- If the `userid` exists in the system, it 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, it will return `None`.

For example, `groupfinder('editor', request)` returns `['group:editor']`, `groupfinder('viewer', request)` returns `[]`, and `groupfinder('admin', request)` returns `None`. We will use `groupfinder()` as an *authentication policy* “callback” that will provide the *principal* or principals for a user.

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.

Add an ACL

Open `tutorial/tutorial/models.py` and add the following import statement at the head:

```
1 from pyramid.security import (
2     Allow,
3     Everyone,
4 )
```

Add the following class definition:

```
1 class RootFactory(object):
2     __acl__ = [ (Allow, Everyone, 'view'),
3                 (Allow, 'group:editors', 'edit') ]
4     def __init__(self, request):
5         pass
```

We import `Allow`, an action that means that permission is allowed:, and `Everyone`, a special *principal* that is associated to all requests. Both are used in the *ACE* entries that make up the ACL.

The ACL is a list that needs to be named `__acl__` and be an attribute of a class. We define an ACL with two *ACE* entries: the first entry allows any user the *view* permission. The second entry allows the `group:editors` principal the *edit* permission.

The `RootFactory` class that contains the ACL is a *root factory*. We need to associate it to our Pyramid application, so the ACL is provided to each view in the *context* of the request, as the `context` attribute.

Open `tutorial/tutorial/__init__.py` and add a `root_factory` parameter to our *Configurator* constructor, that points to the class we created above:

```
1 config = Configurator(settings=settings,  
2                       root_factory='tutorial.models.RootFactory')
```

(Only the highlighted line needs to be added.)

We are now providing the ACL to the application. See *Assigning ACLs to your Resource Objects* for more information about what an ACL represents.



Although we don't use the functionality here, the factory used to create route contexts may differ per-route as opposed to globally. See the `factory` argument to `pyramid.config.Configurator.add_route()` for more info.

Add Authentication and Authorization Policies

Open `tutorial/__init__.py` and add these import statements:

```
1 from pyramid.authentication import AuthTktAuthenticationPolicy  
2 from pyramid.authorization import ACLAuthorizationPolicy  
3 from tutorial.security import groupfinder
```

Now add those policies to the configuration:

```
1     authn_policy = AuthTktAuthenticationPolicy(  
2         'sosecret', callback=groupfinder, hashalg='sha512')  
3     authz_policy = ACLAuthorizationPolicy()  
4     config = Configurator(settings=settings,  
5                             root_factory='tutorial.models.RootFactory')  
6     config.set_authentication_policy(authn_policy)  
7     config.set_authorization_policy(authz_policy)
```

(Only the highlighted lines need to be added.)

We are enabling an `AuthTktAuthenticationPolicy`, which is based in an auth ticket that may be included in the request. We are also enabling an `ACLAuthorizationPolicy`, which uses an ACL to determine the *allow* or *deny* outcome for a view.

Note that the `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 the `groupfinder()` function that we created before.

Add permission declarations

Add a `permission='edit'` parameter to the `@view_config` decorator for `add_page()` and `edit_page()`, for example:

```
1 @view_config(route_name='add_page', renderer='templates/edit.pt',  
2              permission='edit')
```

(Only the highlighted line needs to be added.)

The result is that only users who possess the `edit` permission at the time of the request may invoke those two views.

Add a `permission='view'` parameter to the `@view_config` decorator for `view_wiki()` and `view_page()`, like this:

```
1 @view_config(route_name='view_page', renderer='templates/view.pt',  
2              permission='view')
```

(Only the highlighted line needs to be added.)

This allows anyone to invoke these two views.

We are done with the changes needed to control access. The changes that follow will add the login and logout feature.

40.7.2 Login, Logout

Add routes for /login and /logout

Go back to `tutorial/tutorial/__init__.py` and add these two routes:

```
1 config.add_route('login', '/login')
2 config.add_route('logout', '/logout')
```



The preceding lines must be added *before* the following `view_page` route definition:

```
1 config.add_route('logout', '/logout')
```

This is because `view_page`'s route definition uses a catch-all “replacement marker” `{pagename}` (see *Route Pattern Syntax*) which will catch any route that was not already caught by any route listed above it in `__init__.py`. Hence, for login and logout views to have the opportunity of being matched (or “caught”), they must be above `{pagename}`.

Add 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 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.

Add the following import statements to the head of `tutorial/tutorial/views.py`:

```
1 from pyramid.view import (
2     view_config,
3     forbidden_view_config,
4 )
5
6 from pyramid.security import (
7     remember,
8     forget,
9 )
10
11 from .security import USERS
```

(Only the highlighted lines need to be added.)

`forbidden_view_config()` will be used to customize the default 403 Forbidden page.
`remember()` and `forget()` help to create and expire an auth ticket cookie.

Now add the login and logout views:

```
1 @view_config(route_name='login', renderer='templates/login.pt')
2 @forbidden_view_config(renderer='templates/login.pt')
3 def login(request):
4     login_url = request.route_url('login')
5     referrer = request.url
6     if referrer == login_url:
7         referrer = '/' # never use the login form itself as came_from
8     came_from = request.params.get('came_from', referrer)
9     message = ''
10    login = ''
11    password = ''
12    if 'form.submitted' in request.params:
13        login = request.params['login']
14        password = request.params['password']
15        if USERS.get(login) == password:
16            headers = remember(request, login)
17            return HTTPFound(location = came_from,
18                            headers = headers)
19        message = 'Failed login'
20
21    return dict(
22        message = message,
23        url = request.application_url + '/login',
24        came_from = came_from,
25        login = login,
26        password = password,
27    )
28
29 @view_config(route_name='logout')
30 def logout(request):
31     headers = forget(request)
32     return HTTPFound(location = request.route_url('view_wiki'),
33                     headers = headers)
```

`login()` is decorated with two decorators:

- a `@view_config` decorator which associates it with the `login` route and makes it visible when we visit `/login`,

- a `@forbidden_view_config` decorator which turns it into an *forbidden view*. `login()` will be invoked when a users tries to execute a view callable that they are not allowed to. For example, if a user has not logged in and tries to add or edit a Wiki page, he will be shown the login form before being allowed to continue on.

The order of these two *view configuration* decorators is unimportant.

`logout()` is decorated with a `@view_config` decorator which associates it with the `logout` route. It will be invoked when we visit `/logout`.

Add the `login.pt` Template

Create `tutorial/tutorial/templates/login.pt` with the following content:

```
<!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>
```

```
</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 id="footer">
  <div class="footer">
    &copy; Copyright 2008-2011, Agendaless Consulting.</div>
  </div>
</body>
</html>
```

The above template is referred to within the login view we just added to `views.py`.

Return a `logged_in` flag to the renderer

Add a `logged_in` parameter to the return value of `view_page()`, `edit_page()` and `add_page()`, like this:

```
1 return dict(page = page,
2             content = content,
3             edit_url = edit_url,
4             logged_in = request.authenticated_userid)
```

(Only the highlighted line needs to be added.)

The `authenticated_userid()` property will be `None` if the user is not authenticated.

Add a “Logout” link when logged in

Open `tutorial/tutorial/templates/edit.pt` and `tutorial/tutorial/templates/view.pt` and add this 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>
```

The attribute `tal:condition="logged_in"` will make the element be included when `logged_in` is any user id. The link will invoke the `logout` view. The above element will not be included if `logged_in` is `None`, such as when a user is not authenticated.

40.7.3 Seeing Our Changes

Our `tutorial/tutorial/__init__.py` will look something like this when we’re done:

```
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.security import groupfinder
8
9 from .models import (
10     DBSession,
11     Base,
12 )
13
14
15 def main(global_config, **settings):
16     """ This function returns a Pyramid WSGI application.
17     """
18     engine = engine_from_config(settings, 'sqlalchemy.')
19     DBSession.configure(bind=engine)
20     Base.metadata.bind = engine
21     authn_policy = AuthTktAuthenticationPolicy(
22         'sosecret', callback=groupfinder, hashalg='sha512')
23     authz_policy = ACLAuthorizationPolicy()
24     config = Configurator(settings=settings,
25                           root_factory='tutorial.models.RootFactory')
```

40. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
26 config.set_authentication_policy(authn_policy)
27 config.set_authorization_policy(authz_policy)
28 config.include('pyramid_chameleon')
29 config.add_static_view('static', 'static', cache_max_age=3600)
30 config.add_route('view_wiki', '/')
31 config.add_route('login', '/login')
32 config.add_route('logout', '/logout')
33 config.add_route('view_page', '/{pagename}')
34 config.add_route('add_page', '/add_page/{pagename}')
35 config.add_route('edit_page', '/{pagename}/edit_page')
36 config.scan()
37 return config.make_wsgi_app()
```

(Only the highlighted lines need to be added.)

Our tutorial/tutorial/models.py will look something like this when we're done:

```
1 from pyramid.security import (
2     Allow,
3     Everyone,
4 )
5
6 from sqlalchemy import (
7     Column,
8     Integer,
9     Text,
10 )
11
12 from sqlalchemy.ext.declarative import declarative_base
13
14 from sqlalchemy.orm import (
15     scoped_session,
16     sessionmaker,
17 )
18
19 from zope.sqlalchemy import ZopeTransactionExtension
20
21 DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
22 Base = declarative_base()
23
24
25 class Page(Base):
26     """ The SQLAlchemy declarative model class for a Page object. """
27     __tablename__ = 'pages'
28     id = Column(Integer, primary_key=True)
```

```

29     name = Column(Text, unique=True)
30     data = Column(Text)
31
32
33 class RootFactory(object):
34     __acl__ = [ (Allow, Everyone, 'view'),
35                (Allow, 'group:editors', 'edit') ]
36     def __init__(self, request):
37         pass

```

(Only the highlighted lines need to be added.)

Our `tutorial/tutorial/views.py` will look something like this when we're done:

```

1  import re
2  from docutils.core import publish_parts
3
4  from pyramid.httpexceptions import (
5      HTTPFound,
6      HTTPNotFound,
7  )
8
9  from pyramid.view import (
10     view_config,
11     forbidden_view_config,
12 )
13
14 from pyramid.security import (
15     remember,
16     forget,
17 )
18
19 from .security import USERS
20
21 from .models import (
22     DBSession,
23     Page,
24 )
25
26
27 # regular expression used to find WikiWords
28 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+) ")
29
30 @view_config(route_name='view_wiki',
31             permission='view')

```

```
32 def view_wiki(request):
33     return HTTPFound(location = request.route_url('view_page',
34                                                     pagename='FrontPage'))
35
36 @view_config(route_name='view_page', renderer='templates/view.pt',
37              permission='view')
38 def view_page(request):
39     pagename = request.matchdict['pagename']
40     page = DBSession.query(Page).filter_by(name=pagename).first()
41     if page is None:
42         return HTTPNotFound('No such page')
43
44     def check(match):
45         word = match.group(1)
46         exists = DBSession.query(Page).filter_by(name=word).all()
47         if exists:
48             view_url = request.route_url('view_page', pagename=word)
49             return '<a href="%s">%s</a>' % (view_url, word)
50         else:
51             add_url = request.route_url('add_page', pagename=word)
52             return '<a href="%s">%s</a>' % (add_url, word)
53
54     content = publish_parts(page.data, writer_name='html')['html_body']
55     content = wikiwords.sub(check, content)
56     edit_url = request.route_url('edit_page', pagename=pagename)
57     return dict(page=page, content=content, edit_url=edit_url,
58                 logged_in=request.authenticated_userid)
59
60 @view_config(route_name='add_page', renderer='templates/edit.pt',
61              permission='edit')
62 def add_page(request):
63     pagename = request.matchdict['pagename']
64     if 'form.submitted' in request.params:
65         body = request.params['body']
66         page = Page(name=pagename, data=body)
67         DBSession.add(page)
68         return HTTPFound(location = request.route_url('view_page',
69                                                         pagename=pagename))
70     save_url = request.route_url('add_page', pagename=pagename)
71     page = Page(name='', data='')
72     return dict(page=page, save_url=save_url,
73                 logged_in=request.authenticated_userid)
74
75 @view_config(route_name='edit_page', renderer='templates/edit.pt',
76              permission='edit')
77 def edit_page(request):
```

```

78     pagename = request.matchdict['pagename']
79     page = DBSession.query(Page).filter_by(name=pagename).one()
80     if 'form.submitted' in request.params:
81         page.data = request.params['body']
82         DBSession.add(page)
83         return HTTPFound(location = request.route_url('view_page',
84                                                         pagename=pagename))
85     return dict(
86         page=page,
87         save_url=request.route_url('edit_page', pagename=pagename),
88         logged_in=request.authenticated_userid
89     )
90
91 @view_config(route_name='login', renderer='templates/login.pt')
92 @forbidden_view_config(renderer='templates/login.pt')
93 def login(request):
94     login_url = request.route_url('login')
95     referrer = request.url
96     if referrer == login_url:
97         referrer = '/' # never use the login form itself as came_from
98     came_from = request.params.get('came_from', referrer)
99     message = ''
100    login = ''
101    password = ''
102    if 'form.submitted' in request.params:
103        login = request.params['login']
104        password = request.params['password']
105        if USERS.get(login) == password:
106            headers = remember(request, login)
107            return HTTPFound(location = came_from,
108                            headers = headers)
109        message = 'Failed login'
110
111    return dict(
112        message = message,
113        url = request.application_url + '/login',
114        came_from = came_from,
115        login = login,
116        password = password,
117    )
118
119 @view_config(route_name='logout')
120 def logout(request):
121     headers = forget(request)
122     return HTTPFound(location = request.route_url('view_wiki'),
123                     headers = headers)

```

(Only the highlighted lines need to be added.)

Our tutorial/tutorial/templates/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>

```

(Only the highlighted lines need to be added.)

Our tutorial/tutorial/templates/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>
60     <div id="footer">
61         <div class="footer"
62             >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
63     </div>
64 </body>
65 </html>
```

(Only the highlighted lines need to be added.)

40.7.4 Viewing the Application in a Browser

We can finally examine our application in a browser (See *Starting the Application*). Launch a browser and visit each of the following URLs, check that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` page object. It is executable by any user.
- `http://localhost:6543/FrontPage` invokes the `view_page` view of the `FrontPage` page object.
- `http://localhost:6543/FrontPage/edit_page` 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.
- `http://localhost:6543/add_page/SomePageName` 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.
- After logging 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.

40.8 Adding Tests

We will now add tests for the models and the views and a few functional tests in the `tests.py`. Tests ensure that an application works, and that it continues to work after changes are made in the future.

40.8.1 Testing the Models

To test the model class `Page` we'll add a new `PageModelTests` class to our `tests.py` file that was generated as part of the `alchemy` scaffold.

40.8.2 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` class that the alchemy scaffold provided, 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.

40.8.3 Functional tests

We'll test the whole application, covering security aspects that are not tested in the unit tests, like logging in, logging out, checking that the `viewer` user cannot add or edit pages, but the `editor` user can, and so on.

40.8.4 Viewing the results of all our edits to `tests.py`

Once we're done with the `tests.py` module, it will look a lot like:

```
1 import unittest
2 import transaction
3
4 from pyramid import testing
5
6
7 def _initTestingDB():
8     from sqlalchemy import create_engine
9     from tutorial.models import (
10         DBSession,
11         Page,
12         Base
13     )
14     engine = create_engine('sqlite://')
15     Base.metadata.create_all(engine)
16     DBSession.configure(bind=engine)
17     with transaction.manager:
18         model = Page(name='FrontPage', data='This is the front page')
19         DBSession.add(model)
20     return DBSession
21
22
23 def _registerRoutes(config):
24     config.add_route('view_page', '{pagename}')
```

```

25     config.add_route('edit_page', '{pagename}/edit_page')
26     config.add_route('add_page', 'add_page/{pagename}')
27
28
29 class ViewWikiTests(unittest.TestCase):
30     def setUp(self):
31         self.config = testing.setUp()
32
33     def tearDown(self):
34         testing.tearDown()
35
36     def _callFUT(self, request):
37         from tutorial.views import view_wiki
38         return view_wiki(request)
39
40     def test_it(self):
41         _registerRoutes(self.config)
42         request = testing.DummyRequest()
43         response = self._callFUT(request)
44         self.assertEqual(response.location, 'http://example.com/FrontPage')
45
46
47 class ViewPageTests(unittest.TestCase):
48     def setUp(self):
49         self.session = _initTestingDB()
50         self.config = testing.setUp()
51
52     def tearDown(self):
53         self.session.remove()
54         testing.tearDown()
55
56     def _callFUT(self, request):
57         from tutorial.views import view_page
58         return view_page(request)
59
60     def test_it(self):
61         from tutorial.models import Page
62         request = testing.DummyRequest()
63         request.matchdict['pagename'] = 'IDoExist'
64         page = Page(name='IDoExist', data='Hello CruelWorld IDoExist')
65         self.session.add(page)
66         _registerRoutes(self.config)
67         info = self._callFUT(request)
68         self.assertEqual(info['page'], page)
69         self.assertEqual(
70             info['content'],

```

```
71         '<div class="document">\n'
72         '<p>Hello <a href="http://example.com/add_page/CruelWorld">'
73         'CruelWorld</a> '
74         '<a href="http://example.com/IDoExist">'
75         'IDoExist</a>'
76         '</p>\n</div>\n')
77     self.assertEqual(info['edit_url'],
78                     'http://example.com/IDoExist/edit_page')
79
80
81     class AddPageTests(unittest.TestCase):
82     def setUp(self):
83         self.session = _initTestingDB()
84         self.config = testing.setUp()
85
86     def tearDown(self):
87         self.session.remove()
88         testing.tearDown()
89
90     def _callFUT(self, request):
91         from tutorial.views import add_page
92         return add_page(request)
93
94     def test_it_notsubmitted(self):
95         _registerRoutes(self.config)
96         request = testing.DummyRequest()
97         request.matchdict = {'pagename': 'AnotherPage'}
98         info = self._callFUT(request)
99         self.assertEqual(info['page'].data, '')
100        self.assertEqual(info['save_url'],
101                        'http://example.com/add_page/AnotherPage')
102
103     def test_it_submitted(self):
104         from tutorial.models import Page
105         _registerRoutes(self.config)
106         request = testing.DummyRequest({'form.submitted': True,
107                                         'body': 'Hello yo!'})
108         request.matchdict = {'pagename': 'AnotherPage'}
109         self._callFUT(request)
110         page = self.session.query(Page).filter_by(name='AnotherPage').one()
111         self.assertEqual(page.data, 'Hello yo!')
112
113
114     class EditPageTests(unittest.TestCase):
115     def setUp(self):
116         self.session = _initTestingDB()
```

```

117         self.config = testing.setUp()
118
119     def tearDown(self):
120         self.session.remove()
121         testing.tearDown()
122
123     def _callFUT(self, request):
124         from tutorial.views import edit_page
125         return edit_page(request)
126
127     def test_it_notsubmitted(self):
128         from tutorial.models import Page
129         _registerRoutes(self.config)
130         request = testing.DummyRequest()
131         request.matchdict = {'pagename': 'abc'}
132         page = Page(name='abc', data='hello')
133         self.session.add(page)
134         info = self._callFUT(request)
135         self.assertEqual(info['page'], page)
136         self.assertEqual(info['save_url'],
137                          'http://example.com/abc/edit_page')
138
139     def test_it_submitted(self):
140         from tutorial.models import Page
141         _registerRoutes(self.config)
142         request = testing.DummyRequest({'form.submitted': True,
143                                         'body': 'Hello yo!'})
144         request.matchdict = {'pagename': 'abc'}
145         page = Page(name='abc', data='hello')
146         self.session.add(page)
147         response = self._callFUT(request)
148         self.assertEqual(response.location, 'http://example.com/abc')
149         self.assertEqual(page.data, 'Hello yo!')
150
151
152 class FunctionalTests(unittest.TestCase):
153
154     viewer_login = '/login?login=viewer&password=viewer' \
155                   '&came_from=FrontPage&form.submitted=Login'
156     viewer_wrong_login = '/login?login=viewer&password=incorrect' \
157                          '&came_from=FrontPage&form.submitted=Login'
158     editor_login = '/login?login=editor&password=editor' \
159                   '&came_from=FrontPage&form.submitted=Login'
160
161     def setUp(self):
162         from tutorial import main

```

```
163     settings = { 'sqlalchemy.url': 'sqlite://' }
164     app = main({}, **settings)
165     from webtest import TestApp
166     self.testapp = TestApp(app)
167     _initTestingDB()
168
169     def tearDown(self):
170         del self.testapp
171         from tutorial.models import DBSession
172         DBSession.remove()
173
174     def test_root(self):
175         res = self.testapp.get('/', status=302)
176         self.assertEqual(res.location, 'http://localhost/FrontPage')
177
178     def test_FrontPage(self):
179         res = self.testapp.get('/FrontPage', status=200)
180         self.assertTrue(b'FrontPage' in res.body)
181
182     def test_unexisting_page(self):
183         self.testapp.get('/SomePage', status=404)
184
185     def test_successful_login(self):
186         res = self.testapp.get(self.viewer_login, status=302)
187         self.assertEqual(res.location, 'http://localhost/FrontPage')
188
189     def test_failed_login(self):
190         res = self.testapp.get(self.viewer_wrong_login, status=200)
191         self.assertTrue(b'login' in res.body)
192
193     def test_logout_link_present_when_logged_in(self):
194         self.testapp.get(self.viewer_login, status=302)
195         res = self.testapp.get('/FrontPage', status=200)
196         self.assertTrue(b'Logout' in res.body)
197
198     def test_logout_link_not_present_after_logged_out(self):
199         self.testapp.get(self.viewer_login, status=302)
200         self.testapp.get('/FrontPage', status=200)
201         res = self.testapp.get('/logout', status=302)
202         self.assertTrue(b'Logout' not in res.body)
203
204     def test_anonymous_user_cannot_edit(self):
205         res = self.testapp.get('/FrontPage/edit_page', status=200)
206         self.assertTrue(b'Login' in res.body)
207
208     def test_anonymous_user_cannot_add(self):
```

```

209         res = self.testapp.get('/add_page/NewPage', status=200)
210         self.assertTrue(b'Login' in res.body)
211
212     def test_viewer_user_cannot_edit(self):
213         self.testapp.get(self.viewer_login, status=302)
214         res = self.testapp.get('/FrontPage/edit_page', status=200)
215         self.assertTrue(b'Login' in res.body)
216
217     def test_viewer_user_cannot_add(self):
218         self.testapp.get(self.viewer_login, status=302)
219         res = self.testapp.get('/add_page/NewPage', status=200)
220         self.assertTrue(b'Login' in res.body)
221
222     def test_editors_member_user_can_edit(self):
223         self.testapp.get(self.editor_login, status=302)
224         res = self.testapp.get('/FrontPage/edit_page', status=200)
225         self.assertTrue(b'Editing' in res.body)
226
227     def test_editors_member_user_can_add(self):
228         self.testapp.get(self.editor_login, status=302)
229         res = self.testapp.get('/add_page/NewPage', status=200)
230         self.assertTrue(b'Editing' in res.body)
231
232     def test_editors_member_user_can_view(self):
233         self.testapp.get(self.editor_login, status=302)
234         res = self.testapp.get('/FrontPage', status=200)
235         self.assertTrue(b'FrontPage' in res.body)

```

40.8.5 Running the Tests

We can run these tests by using `setup.py test` in the same way we did in *Run the Tests*. However, first we must edit our `setup.py` to include a dependency on `WebTest`, which we've used in our `tests.py`. Change the `requires` list in `setup.py` to include `WebTest`.

```

1  requires = [
2      'pyramid',
3      'pyramid_chameleon',
4      'pyramid_debugtoolbar',
5      'pyramid_tm',
6      'SQLAlchemy',
7      'transaction',
8      'zope.sqlalchemy',
9      'waitress',

```

40. SQLALCHEMY + URL DISPATCH WIKI TUTORIAL

```
10     'docutils',
11     'WebTest', # add this
12 ]
```

After we've added a dependency on `WebTest` in `setup.py`, we need to rerun `setup.py develop` to get `WebTest` installed into our virtualenv. Assuming our shell's current working directory is the “tutorial” distribution directory:

On UNIX:

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

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py develop
```

Once that command has completed successfully, we can run the tests themselves:

On UNIX:

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

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py test -q
```

The expected result ends something like:

```
.....
-----
Ran 21 tests in 2.700s

OK
```

40.9 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:

```
$ $VENV/bin/python setup.py sdist
```

On Windows:

```
c:\pyramidtut> %VENV%\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-0.0.tar tutorial-0.0
gzip -f9 dist/tutorial-0.0.tar
removing 'tutorial-0.0' (and everything under it)
```

Note that this command creates a tarball in the “dist” subdirectory named `tutorial-0.0.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.

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 on GitHub at [docs/tutorials/wiki/src](#), which corresponds to the same location if you have Pyramid sources.

41.1 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* a Windows system of any kind.



This tutorial has been written for Python 2. It is unlikely to work without modification under Python 3.

Have fun!

41.2 Design

Following is a quick overview of our wiki application, to help us understand the changes that we will be doing next in our default files generated by the `zodb` scaffold.

41.2.1 Overall

We choose to use `reStructuredText` markup in the wiki text. Translation from `reStructuredText` to HTML is provided by the widely used `docutils` Python module. We will add this module in the dependency list on the project `setup.py` file.

41.2.2 Models

The root resource, named *Wiki*, will be a mapping of wiki page names to page resources. The page resources will be instances of a *Page* class and they store the text content.

URLs like `/PageName` will be traversed using `Wiki[PageName] => page`, and the context that results is the page resource of an existing page.

To add a page to the wiki, a new instance of the page resource is created and its name and reference are added to the Wiki mapping.

A page named *FrontPage* containing the text *This is the front page*, will be created when the storage is initialized, and will be used as the wiki home page.

41.2.3 Views

There will be three views to handle the normal operations of adding, editing, and viewing wiki pages, plus one view for the wiki front page. Two templates will be used, one for viewing, and one for both adding and editing wiki pages.

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.

41.2.4 Security

We'll eventually be adding security to our application. The components we'll use to do this are below.

- `USERS`, a dictionary mapping usernames to their corresponding passwords.
- `GROUPS`, a dictionary mapping usernames to a list of groups to which they belong to.
- `groupfinder`, an *authorization callback* that looks up `USERS` and `GROUPS`. It will be provided in a new *security.py* file.
- An *ACL* is attached to the root *resource*. Each row below details an *ACE*:

| Action | Principal | Permission |
|--------|---------------|------------|
| Allow | Everyone | View |
| Allow | group:editors | Edit |

- Permission declarations are added to the views to assert the security policies as each request is handled.

Two additional views and one template will handle the login and logout tasks.

41.2.5 Summary

The URL, context, actions, template and permission associated to each view are listed in the following table:

| URL | View | Context | Action | Template | Permission |
|---------------------|------------------------|------------------------------|---|----------|------------|
| / | view_wiki | Wiki | Redirect to /FrontPage | | |
| /PageName | view_page ¹ | Page | Display existing page ² | view.pt | view |
| /PageName/edit_page | edit_page | Page | Display edit form with existing content. If the form was submitted, redirect to /PageName | edit.pt | edit |
| /add_page/PageName | add_page | Wiki | Create the page <i>PageName</i> in storage, display the edit form without content. If the form was submitted, redirect to /PageName | edit.pt | edit |
| /login | login | Wiki, Forbidden ³ | Display login form. If the form was submitted, authenticate. <ul style="list-style-type: none">• If authentication successful, redirect to the page that we came from.• If authentication fails, display login form with “login failed” message. | login.pt | |
| /logout | logout | Wiki | Redirect to /FrontPage | | |

41.3 Installation

41.3.1 Preparation

Follow the steps in *Installing Pyramid*, but name the virtualenv directory `pyramidtut`.

Preparation, UNIX

1. Switch to the `pyramidtut` directory:

```
$ cd pyramidtut
```

2. Install tutorial dependencies:

```
$ $ENV/bin/easy_install docutils pyramid_tm pyramid_zodbconn \  
    pyramid_debugtoolbar nose coverage
```

Preparation, Windows

1. Switch to the `pyramidtut` directory:

```
c:\> cd pyramidtut
```

2. Install tutorial dependencies:

```
c:\pyramidtut> %ENV%\Scripts\easy_install docutils pyramid_tm \  
    pyramid_zodbconn pyramid_debugtoolbar nose coverage
```

41.3.2 Make a Project

Your next step is to create a project. For this tutorial, we will use the *scaffold* named `zodb`, which generates an application that uses *ZODB* and *traversal*. Pyramid supplies a variety of scaffolds to generate sample projects.

The below instructions assume your current working directory is the “virtualenv” named “`pyramidtut`”.


On UNIX:


41. ZODB + TRAVERSAL WIKI TUTORIAL

```
$ $VENV/bin/pcreate -s zodb tutorial
```

On Windows:

```
c:\pyramidtut> %VENV%\Scripts\pcreate -s zodb tutorial
```

 You don't have to call it *tutorial* – the code uses relative paths for imports and finding templates and static resources.

 If you are using Windows, the `zodb` scaffold 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.

41.3.3 Install 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 *Make a Project*, and run the “`setup.py develop`” command using `virtualenv` Python interpreter.

On UNIX:

```
$ cd tutorial
$ $VENV/bin/python setup.py develop
```

On Windows:

```
C:\pyramidtut> cd tutorial
C:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py develop
```

41.3.4 Run the Tests

After you've installed the project in development mode, you may run the tests for the project.

On UNIX:

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

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py test -q
```

41.3.5 Expose 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:

```
$ $VENV/bin/nosetests --cover-package=tutorial --cover-erase --with-coverage
```

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\nosetests --cover-package=tutorial ^  
--cover-erase --with-coverage
```

Looks like the code in the `zodb` scaffold for ZODB projects is missing some test coverage, particularly in the file named `models.py`.

41.3.6 Start the Application

Start the application.

On UNIX:

```
$ $VENV/bin/pserve development.ini --reload
```

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\pserve development.ini --reload
```



Your OS firewall, if any, may pop up a dialog asking for authorization to allow python to accept incoming network connections.

41.3.7 Visit the Application in a Browser

In a browser, visit `http://localhost:6543/`. You will see the generated application's default page.

One thing you'll notice is the “debug toolbar” icon on right hand side of the page. You can read more about the purpose of the icon at *The Debug Toolbar*. It allows you to get information about your application while you develop.

41.3.8 Decisions the zodb Scaffold Has Made For You

Creating a project using the `zodb` scaffold makes the following assumptions:

- you are willing to use *ZODB* as persistent storage
- you are willing to use *traversal* to map URLs to code.



Pyramid supports any persistent storage mechanism (e.g., a SQL database or filesystem files). 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*.

41.4 Basic Layout

The starter files generated by the `zodb` scaffold are basic, but they provide a good orientation for the high-level patterns common to most *traversal*-based Pyramid (and *ZODB*-based) projects.

41.4.1 Application Configuration 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` both as a package marker and to contain application configuration code.

When you run the application using the `pserve` command using the `development.ini` generated config file, the application configuration points at a 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 pyramid_zodbconn import get_connection
3  from .models import appmaker
4
5
6  def root_factory(request):
7      conn = get_connection(request)
8      return appmaker(conn.root())
9
10
11 def main(global_config, **settings):
12     """ This function returns a Pyramid WSGI application.
13     """
14     config = Configurator(root_factory=root_factory, settings=settings)
15     config.include('pyramid_chameleon')
16     config.add_static_view('static', 'static', cache_max_age=3600)
17     config.scan()
18     return config.make_wsgi_app()

```

1. *Lines 1-3.* Perform some dependency imports.
2. *Lines 6-8.* Define a root factory for our Pyramid application.
3. *Line 14.* We construct a *Configurator* with a *root factory* and the settings keywords parsed by *PasteDeploy*. The root factory is named `root_factory`.
4. *Line 15.* Include support for the *Chameleon* template rendering bindings, allowing us to use the `.pt` templates.
5. *Line 16.* Register a “static view” which answers requests whose URL path start with `/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 a relative *asset specification*, so it finds the resources it should serve within the `static` directory inside the `tutorial` package. Alternatively the scaffold could have used an *absolute* asset specification as the path (`tutorial:static`).

6. *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. We don’t pass any arguments to `scan()`, which implies that the scan should take place in the current package (in this case, `tutorial`). The scaffold could have equivalently said `config.scan('tutorial')`, but it chose to omit the package name argument.
7. *Line 18.* Use the `pyramid.config.Configurator.make_wsgi_app()` method to return a WSGI application.

41.4.2 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 *zodb* scaffold put the classes that implement our resource objects, each of which also happens to be a domain model object.

Here is the source for `models.py`:

```
1 from persistent.mapping import PersistentMapping
2
3
4 class MyModel(PersistentMapping):
5     __parent__ = __name__ = None
6
7
8 def appmaker(zodb_root):
9     if not 'app_root' in zodb_root:
10         app_root = MyModel()
11         zodb_root['app_root'] = app_root
12         import transaction
13         transaction.commit()
14     return zodb_root['app_root']
```

1. *Lines 4-5.* The `MyModel` *resource* class is implemented here. Instances of this class are 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 8-14.* `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. It is used by the “*root_factory*” we’ve defined in our `__init__.py`.

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.

41.4.3 Views With `views.py`

Our scaffold 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 .models import MyModel
3
4
5 @view_config(context=MyModel, renderer='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 5.* 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 14 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 `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. This asset specification is *relative* (to the `view.py`'s current package). Alternatively we could have used the absolute asset specification `tutorial:templates/mytemplate.pt`, but chose to use the relative version.

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 6-7.* 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 `zodb` scaffold 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.

41.4.4 Configuration in `development.ini`

The `development.ini` (in the *tutorial project* directory, as opposed to the *tutorial package* directory) looks like this:

```
###
# app configuration
# http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/environment.html
###

[app:main]
use = egg:tutorial

pyramid.reload_templates = true
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.debug_routematch = false
pyramid.default_locale_name = en
```

```
pyramid.includes =
    pyramid_debugtoolbar
    pyramid_zodbconn
    pyramid_tm

tm.attempts = 3
zodbconn.uri = file://%(here)s/Data.fs?connection_cache_size=20000

# By default, the toolbar only appears for clients from IP addresses
# '127.0.0.1' and '::1'.
# debugtoolbar.hosts = 127.0.0.1 ::1

###
# wsgi server configuration
###

[server:main]
use = egg:waitress#main
host = 0.0.0.0
port = 6543

###
# logging configuration
# http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/logging.html
###

[loggers]
keys = root, tutorial

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = INFO
handlers = console

[logger_tutorial]
level = DEBUG
handlers =
qualname = tutorial

[handler_console]
class = StreamHandler
```

```
args = (sys.stderr,)
level = NOTSET
formatter = generic

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

Note the existence of a `[app:main]` section which specifies our WSGI application. Our ZODB database settings are specified as the `zodbconn.uri` setting within this section. This value, and the other values within this section are passed as `**settings` to the main function we defined in `__init__.py` when the server is started via `pserve`.

41.5 Defining the Domain Model

The first change we'll make to our stock pcreate-generated application will be to define two *resource* constructors, one representing a wiki page, and another representing the wiki as a mapping of wiki page names to page objects. We'll do this inside our `models.py` file.

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”. 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.

41.5.1 Delete the Database

In the next 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.

41.5.2 Edit `models.py`



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 `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 `Wiki` class. We want it to inherit from the `persistent.mapping.PersistentMapping` class because it provides mapping behavior, and it makes sure that our `Wiki` page is stored as a “first-class” persistent object in our ZODB database.

Our `Wiki` class should have two attributes set to `None` at class scope: `__parent__` and `__name__`. 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 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.

41.5.3 Look 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']
```

41.5.4 View 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 (See *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.

41.6 Defining Views

A *view callable* in a *traversal*-based Pyramid application is typically a simple Python function that accepts two parameters: *context* and *request*. A view callable is assumed to return a *response* object.

 A Pyramid view can also be defined as callable which accepts *only* a *request* argument. You'll see this one-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 *traversal* based applications, URLs are mapped 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, in this tutorial we define views as callables that accept *context* in the callable argument list. 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`.

We're going to define several *view callable* functions, then wire them into Pyramid using some *view configuration*.

41.6.1 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 `pcreate` 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
3  from setuptools import setup, find_packages
4
5  here = os.path.abspath(os.path.dirname(__file__))
6  with open(os.path.join(here, 'README.txt')) as f:
7      README = f.read()
8  with open(os.path.join(here, 'CHANGES.txt')) as f:
9      CHANGES = f.read()
10
11  requires = [
12      'pyramid',
13      'pyramid_chameleon',
14      'pyramid_zodbconn',
15      'transaction',
16      'pyramid_tm',
17      'pyramid_debugtoolbar',
18      'ZODB3',
19      'waitress',
20      'docutils',
21  ]
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 :: Pyramid",
30            "Topic :: Internet :: WWW/HTTP",
31            "Topic :: Internet :: WWW/HTTP :: WSGI :: Application",
32        ],
33        author='',
34        author_email='',
35        url='',
36        keywords='web pylons pyramid',

```

```
37     packages=find_packages(),
38     include_package_data=True,
39     zip_safe=False,
40     install_requires=requires,
41     tests_require=requires,
42     test_suite="tutorial",
43     entry_points="""\
44     [paste.app_factory]
45     main = tutorial:main
46     """,
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.

41.6.2 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.



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

Here is the code for the `view_wiki` view function and its decorator, which will be added to `views.py`:

```
@view_config(context='.models.Wiki')
def view_wiki(context, request):
    return HTTPFound(location=request.resource_url(context, 'FrontPage'))
```

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 "Front-Page". To do so, it returns an instance of the `pyramid.httpexceptions.HTTPFound` class (instances of which implement the `pyramid.interfaces.IResponse` interface like `pyramid.response.Response` does). `pyramid.request.Request.resource_url()` constructs a URL to the `FrontPage` page resource (i.e., `http://localhost:6543/FrontPage`), and uses it as the "location" of the `HTTPFound` response, forming an HTTP redirect.

The `view_page` view function

Here is the code for the `view_page` view function and its decorator, which will be added to `views.py`:

```
@view_config(context='.models.Page', renderer='templates/view.pt')
def view_page(context, request):
    wiki = context.__parent__

    def check(match):
        word = match.group(1)
        if word in wiki:
            page = wiki[word]
            view_url = request.resource_url(page)
            return '<a href="%s">%s</a>' % (view_url, word)
        else:
            add_url = request.application_url + '/add_page/' + word
            return '<a href="%s">%s</a>' % (add_url, word)

    content = publish_parts(context.data, writer_name='html')['html_body']
    content = wikiwords.sub(check, content)
    edit_url = request.resource_url(context, 'edit_page')
    return dict(page = context, content = content, edit_url = edit_url)
```

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.Page` 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 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

Here is the code for the `add_page` view function and its decorator, which will be added to `views.py`:

```
@view_config(name='add_page', context='.models.Wiki',
              renderer='templates/edit.pt')
def add_page(context, request):
    pagename = request.subpath[0]
    if 'form.submitted' in request.params:
        body = request.params['body']
        page = Page(body)
        page.__name__ = pagename
        page.__parent__ = context
        context[pagename] = page
    return HTTPFound(location = request.resource_url(page))
```

```

save_url = request.resource_url(context, 'add_page', pagename)
page = Page('')
page.__name__ = pagename
page.__parent__ = context
return dict(page = page, save_url = save_url)

```

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 uses 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

Here is the code for the `edit_page` view function and its decorator, which will be added to `views.py`:

```
@view_config(name='edit_page', context='.models.Page',
             renderer='templates/edit.pt')
def edit_page(context, request):
    if 'form.submitted' in request.params:
        context.data = request.params['body']
        return HTTPFound(location = request.resource_url(context))

    return dict(page=context,
                save_url=request.resource_url(context, 'edit_page'))
```

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 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.

41.6.3 Viewing the Result of all 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.view import view_config
6
7 from .models import Page
```

```

8
9 # regular expression used to find WikiWords
10 wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+)")
11
12 @view_config(context='.models.Wiki')
13 def view_wiki(context, request):
14     return HTTPFound(location=request.resource_url(context, 'FrontPage'))
15
16 @view_config(context='.models.Page', renderer='templates/view.pt')
17 def view_page(context, request):
18     wiki = context.__parent__
19
20     def check(match):
21         word = match.group(1)
22         if word in wiki:
23             page = wiki[word]
24             view_url = request.resource_url(page)
25             return '<a href="%s">%s</a>' % (view_url, word)
26         else:
27             add_url = request.application_url + '/add_page/' + word
28             return '<a href="%s">%s</a>' % (add_url, word)
29
30     content = publish_parts(context.data, writer_name='html')['html_body']
31     content = wikiwords.sub(check, content)
32     edit_url = request.resource_url(context, 'edit_page')
33     return dict(page = context, content = content, edit_url = edit_url)
34
35 @view_config(name='add_page', context='.models.Wiki',
36             renderer='templates/edit.pt')
37 def add_page(context, request):
38     pagename = request.subpath[0]
39     if 'form.submitted' in request.params:
40         body = request.params['body']
41         page = Page(body)
42         page.__name__ = pagename
43         page.__parent__ = context
44         context[pagename] = page
45         return HTTPFound(location = request.resource_url(page))
46     save_url = request.resource_url(context, 'add_page', pagename)
47     page = Page('')
48     page.__name__ = pagename
49     page.__parent__ = context
50     return dict(page = page, save_url = save_url)
51
52 @view_config(name='edit_page', context='.models.Page',
53             renderer='templates/edit.pt')

```

```
54 def edit_page(context, request):
55     if 'form.submitted' in request.params:
56         context.data = request.params['body']
57         return HTTPFound(location = request.resource_url(context))
58
59     return dict(page=context,
60                 save_url=request.resource_url(context, 'edit_page'))
```

41.6.4 Adding Templates

The `view_page`, `add_page` and `edit_page` views that we've added reference a *template*. Each template is a *Chameleon ZPT* template. These templates 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

Create `tutorial/tutorial/templates/view.pt` and add the following content:

```
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="/static/favicon.ico" />
13  <link rel="stylesheet"
14        href="/static/pylons.css"
15        type="text/css" media="screen" charset="utf-8" />
16  <!--[if lte IE 6]>
17  <link rel="stylesheet"
18        href="/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 Goes
36                 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"></div>
41         </div>
42     </div>
43     <div id="bottom">
44         <div class="bottom">
45             <div tal:replace="structure content">
46                 Page text goes here.
47             </div>
48             <p>
49                 <a tal:attributes="href edit_url" href="">
50                     Edit this page
51                 </a>
52             </p>
53         </div>
54     </div>
55     <div id="footer">
56         <div class="footer"
57             >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
58     </div>
59 </body>
60 </html>

```

This template is used by `view_page()` for displaying a single wiki page. It includes:

- A `div` element that is replaced with the content value provided by the view (rows 45-47). content contains HTML, so the `structure` keyword is used to prevent escaping it (i.e. changing “>” to “>”, etc.)
- A link that points at the “edit” URL which invokes the `edit_page` view for the page being viewed (rows 49-51).

The edit.pt Template

Create tutorial/tutorial/templates/edit.pt and add the following content:

```
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>${page.__name__} - Pyramid tutorial wiki (based on
7     TurboGears 20-Minute Wiki)</title>
8   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
9   <meta name="keywords" content="python web application" />
10  <meta name="description" content="pyramid web application" />
11  <link rel="shortcut icon"
12        href="/static/favicon.ico" />
13  <link rel="stylesheet"
14        href="/static/pylons.css"
15        type="text/css" media="screen" charset="utf-8" />
16  <!--[if lte IE 6]>
17  <link rel="stylesheet"
18        href="/static/ie6.css"
19        type="text/css" media="screen" charset="utf-8" />
20  <![endif]>-->
21 </head>
22 <body>
23   <div id="wrap">
24     <div id="top-small">
25       <div class="top-small align-center">
26         <div>
27           
29         </div>
30       </div>
31     </div>
32     <div id="middle">
33       <div class="middle align-right">
34         <div id="left" class="app-welcome align-left">
35           Editing <b><span tal:replace="page.__name__">Page Name Goes
36             Here</span></b><br/>
37           You can return to the
38           <a href="${request.application_url}">FrontPage</a>.<br/>
39         </div>
40         <div id="right" class="app-welcome align-right"></div>
41       </div>
42     </div>
```

```

43     <div id="bottom">
44         <div class="bottom">
45             <form action="{save_url}" method="post">
46                 <textarea name="body" tal:content="page.data" rows="10"
47                     cols="60"/><br/>
48                 <input type="submit" name="form.submitted" value="Save"/>
49             </form>
50         </div>
51     </div>
52 </div>
53 <div id="footer">
54     <div class="footer"
55         >&copy; Copyright 2008-2011, Agendaless Consulting.</div>
56 </div>
57 </body>
58 </html>

```

This template is used by `add_page()` and `edit_page()` for adding and editing a wiki page. It displays a page containing a form that includes:

- A 10 row by 60 column `textarea` field named `body` that is filled with any existing page data when it is rendered (rows 46-47).
- A submit button that has the name `form.submitted` (row 48).

The form POSTs back to the “`save_url`” argument supplied by the view (row 45). The view will use the `body` and `form.submitted` values.



Our templates use a `request` object that none of our tutorial views return in their dictionary. `request` is one of several names that are available “by default” in a template when a template renderer is used. See *System Values Used During Rendering* for information about other names that are available by default when a template is used as a renderer.

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. `/static/pylons.css` by virtue of the call to `add_static_view` directive we’ve made in the `__init__.py` file. Any number and type of static assets can be placed in this directory (or subdirectories) and are just referred to by URL.

41.6.5 Viewing the Application in a Browser

We can finally examine our application in a browser (See *Start the Application*). Launch a browser and visit each of the following URLs, check that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` Page resource.
- `http://localhost:6543/FrontPage/` 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.
- `http://localhost:6543/FrontPage/edit_page` invokes the `edit` view for the `FrontPage` Page resource.
- `http://localhost:6543/add_page/SomePageName` invokes the `add` view for a Page.
- To generate an error, visit `http://localhost:6543/add_page` which will generate an `IndexError: tuple index out of range` error. You'll see an interactive traceback facility provided by `pyramid_debugtoolbar`.

41.7 Adding Authorization

Pyramid provides facilities for *authentication* and *authorization*. We'll make use of both features to provide security to our application. Our application currently allows anyone with access to the server to view, edit, and add pages to our wiki. We'll change that to allow only people who 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.

We will also add a login page and a logout link on all the pages. The login page will be shown when a user is denied access to any of the views that require a permission, instead of a default "403 Forbidden" page.

We will implement the access control with the following steps:

- Add users and groups (`security.py`, a new module).
- Add an *ACL* (`models.py`).
- Add an *authentication policy* and an *authorization policy* (`__init__.py`).
- Add *permission* declarations to the `edit_page` and `add_page` views (`views.py`).

Then we will add the login and logout feature:

- Add login and logout views (`views.py`).
- Add a login template (`login.pt`).
- Make the existing views return a `logged_in` flag to the renderer (`views.py`).
- Add a "Logout" link to be shown when logged in and viewing or editing a page (`view.pt`, `edit.pt`).

41.7.1 Access Control

Add users and groups

Create a new `tutorial/tutorial/security.py` module 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 accepts a `userid` and a `request` and returns one of these values:

- If the `userid` exists in the system, it 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, it will return `None`.

For example, `groupfinder('editor', request)` returns `['group:editor']`, `groupfinder('viewer', request)` returns `[]`, and `groupfinder('admin', request)` returns `None`. We will use `groupfinder()` as an *authentication policy* “callback” that will provide the *principal* or principals for a user.

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.

Add an ACL

Open `tutorial/tutorial/models.py` and add the following import statement at the head:

```
1 from pyramid.security import (  
2     Allow,  
3     Everyone,  
4 )
```

Add the following lines to the `Wiki` class:

```
1 class Wiki(PersistentMapping):
2     __name__ = None
3     __parent__ = None
4     __acl__ = [ (Allow, Everyone, 'view'),
5                 (Allow, 'group:editors', 'edit') ]
```

We import `Allow`, an action that means that permission is allowed, and `Everyone`, a special *principal* that is associated to all requests. Both are used in the *ACE* entries that make up the ACL.

The ACL is a list that needs to be named `__acl__` and be an attribute of a class. We define an ACL with two *ACE* entries: the first entry allows any user the *view* permission, and the second entry allows the `group:editors` principal the *edit* permission.

The `Wiki` class that contains the ACL is the *resource* constructor for the *root* resource, which is a `Wiki` instance. The ACL is provided to each view in the *context* of the request, as the `context` attribute.

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 need only *one* ACL for the entire system, however, because our security requirements are simple, so this feature is not demonstrated. See *Assigning ACLs to your Resource Objects* for more information about what an ACL represents.

Add Authentication and Authorization Policies

Open `tutorial/__init__.py` and add these import statements:

```
1 from pyramid.authentication import AuthTktAuthenticationPolicy
2 from pyramid.authorization import ACLAuthorizationPolicy
3 from .security import groupfinder
```

Now add those policies to the configuration:

```
1 authn_policy = AuthTktAuthenticationPolicy(
2     'sosecret', callback=groupfinder, hashalg='sha512')
3 authz_policy = ACLAuthorizationPolicy()
4 config = Configurator(root_factory=root_factory, settings=settings)
5 config.set_authentication_policy(authn_policy)
6 config.set_authorization_policy(authz_policy)
```

(Only the highlighted lines need to be added.)

We are enabling an `AuthTktAuthenticationPolicy`, it is based in an auth ticket that may be included in the request, and an `ACLAuthorizationPolicy` that uses an ACL to determine the allow or deny outcome for a view.

Note 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 the `groupfinder()` function that we created before.

Add permission declarations

Open `tutorial/tutorial/views.py`. Add a `permission='edit'` parameter to the `@view_config` decorator for `add_page()` and `edit_page()`, for example:

```
1 @view_config(name='add_page', context='.models.Wiki',
2             renderer='templates/edit.pt',
3             permission='edit')
```

(Only the highlighted line, along with its preceding comma, needs to be added.)

The result is that only users who possess the `edit` permission at the time of the request may invoke those two views.

Add a `permission='view'` parameter to the `@view_config` decorator for `view_wiki()` and `view_page()`, like this:

```
1 @view_config(context='.models.Page', renderer='templates/view.pt',
2             permission='view')
```

(Only the highlighted line, along with its preceding comma, needs to be added.)

This allows anyone to invoke these two views.

We are done with the changes needed to control access. The changes that follow will add the login and logout feature.

41.7.2 Login, Logout

Add 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 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.

Add the following import statements to the head of `tutorial/tutorial/views.py`:

```
1 from pyramid.view import (  
2     view_config,  
3     forbidden_view_config,  
4 )  
5  
6 from pyramid.security import (  
7     remember,  
8     forget,  
9 )  
10  
11 from .security import USERS
```

(Only the highlighted lines, with other necessary modifications, need to be added.)

`forbidden_view_config()` will be used to customize the default 403 Forbidden page. `remember()` and `forget()` help to create and expire an auth ticket cookie.

Now add the login and logout views:

```
1 @view_config(context='.models.Wiki', name='login',  
2             renderer='templates/login.pt')  
3 @forbidden_view_config(renderer='templates/login.pt')  
4 def login(request):  
5     login_url = request.resource_url(request.context, 'login')  
6     referrer = request.url  
7     if referrer == login_url:  
8         referrer = '/' # never use the login form itself as came_from  
9     came_from = request.params.get('came_from', referrer)  
10    message = ''  
11    login = ''  
12    password = ''
```

```

13     if 'form.submitted' in request.params:
14         login = request.params['login']
15         password = request.params['password']
16         if USERS.get(login) == password:
17             headers = remember(request, login)
18             return HTTPFound(location = came_from,
19                             headers = headers)
20         message = 'Failed login'
21
22     return dict(
23         message = message,
24         url = request.application_url + '/login',
25         came_from = came_from,
26         login = login,
27         password = password,
28     )
29
30 @view_config(context='.models.Wiki', name='logout')
31 def logout(request):
32     headers = forget(request)
33     return HTTPFound(location = request.resource_url(request.context),
34                     headers = headers)

```

`login()` has two decorators:

- a `@view_config` decorator which associates it with the `login` route and makes it visible when we visit `/login`,
- a `@forbidden_view_config` decorator which turns it into a *forbidden view*. `login()` will be invoked when a user tries to execute a view callable for which they lack authorization. For example, if a user has not logged in and tries to add or edit a Wiki page, they will be shown the login form before being allowed to continue.

The order of these two *view configuration* decorators is unimportant.

`logout()` is decorated with a `@view_config` decorator which associates it with the `logout` route. It will be invoked when we visit `/logout`.

Add the `login.pt` Template

Create `tutorial/tutorial/templates/login.pt` with the following content:

```
<!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="/static/favicon.ico" />
  <link rel="stylesheet"
    href="/static/pylons.css"
    type="text/css" media="screen" charset="utf-8" />
  <!--[if lte IE 6]>
  <link rel="stylesheet"
    href="/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>
</body>
</html>

```

The above template is referred in the login view that we just added in `views.py`.

Return a `logged_in` flag to the renderer

Add a `logged_in` parameter to the return value of `view_page()`, `edit_page()` and `add_page()`, like this:

```

1  return dict(page = page,
2              content = content,
3              edit_url = edit_url,
4              logged_in = request.authenticated_userid)

```

(Only the highlighted line and a trailing comma on the preceding line need to be added.)

The `pyramid.request.Request.authenticated_userid()` will be `None` if the user is not authenticated, or a user id if the user is authenticated.

Add a “Logout” link when logged in

Open `tutorial/tutorial/templates/edit.pt` and `tutorial/tutorial/templates/view.pt` and add this 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>

```

The attribute `tal:condition="logged_in"` will make the element be included when `logged_in` is any user id. The link will invoke the logout view. The above element will not be included if `logged_in` is `None`, such as when a user is not authenticated.

41.7.3 Seeing Our Changes

Our `tutorial/tutorial/__init__.py` will look something like this when we're done:

```
1 from pyramid.config import Configurator
2 from pyramid_zodbconn import get_connection
3
4 from pyramid.authentication import AuthTktAuthenticationPolicy
5 from pyramid.authorization import ACLAuthorizationPolicy
6
7 from .models import appmaker
8 from .security import groupfinder
9
10 def root_factory(request):
11     conn = get_connection(request)
12     return appmaker(conn.root())
13
14
15 def main(global_config, **settings):
16     """ This function returns a Pyramid WSGI application.
17     """
18     authn_policy = AuthTktAuthenticationPolicy(
19         'sosecret', callback=groupfinder, hashalg='sha512')
20     authz_policy = ACLAuthorizationPolicy()
21     config = Configurator(root_factory=root_factory, settings=settings)
22     config.set_authentication_policy(authn_policy)
23     config.set_authorization_policy(authz_policy)
24     config.include('pyramid_chameleon')
25     config.add_static_view('static', 'static', cache_max_age=3600)
26     config.scan()
27     return config.make_wsgi_app()
```

(Only the highlighted lines need to be added.)

Our `tutorial/tutorial/models.py` will look something like this when we're done:

```
1 from persistent import Persistent
2 from persistent.mapping import PersistentMapping
3
4 from pyramid.security import (
5     Allow,
6     Everyone,
7 )
8
9 class Wiki(PersistentMapping):
```

```

10     __name__ = None
11     __parent__ = None
12     __acl__ = [ (Allow, Everyone, 'view'),
13                 (Allow, 'group:editors', 'edit') ]
14
15 class Page(Persistent):
16     def __init__(self, data):
17         self.data = data
18
19 def appmaker(zodb_root):
20     if not 'app_root' in zodb_root:
21         app_root = Wiki()
22         frontpage = Page('This is the front page')
23         app_root['FrontPage'] = frontpage
24         frontpage.__name__ = 'FrontPage'
25         frontpage.__parent__ = app_root
26         zodb_root['app_root'] = app_root
27         import transaction
28         transaction.commit()
29     return zodb_root['app_root']

```

(Only the highlighted lines need to be added.)

Our `tutorial/tutorial/views.py` 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
6  from pyramid.view import (
7      view_config,
8      forbidden_view_config,
9  )
10
11  from pyramid.security import (
12      remember,
13      forget,
14  )
15
16
17  from .security import USERS
18  from .models import Page
19
20  # regular expression used to find WikiWords

```

```
21 | wikiwords = re.compile(r"\b([A-Z]\w+[A-Z]+\w+)")
22 |
23 | @view_config(context='.models.Wiki',
24 |             permission='view')
25 | def view_wiki(context, request):
26 |     return HTTPFound(location=request.resource_url(context, 'FrontPage'))
27 |
28 | @view_config(context='.models.Page', renderer='templates/view.pt',
29 |             permission='view')
30 | def view_page(context, request):
31 |     wiki = context.__parent__
32 |
33 |     def check(match):
34 |         word = match.group(1)
35 |         if word in wiki:
36 |             page = wiki[word]
37 |             view_url = request.resource_url(page)
38 |             return '<a href="%s">%s</a>' % (view_url, word)
39 |         else:
40 |             add_url = request.application_url + '/add_page/' + word
41 |             return '<a href="%s">%s</a>' % (add_url, word)
42 |
43 |     content = publish_parts(context.data, writer_name='html')['html_body']
44 |     content = wikiwords.sub(check, content)
45 |     edit_url = request.resource_url(context, 'edit_page')
46 |
47 |     return dict(page = context, content = content, edit_url = edit_url,
48 |               logged_in = request.authenticated_userid)
49 |
50 | @view_config(name='add_page', context='.models.Wiki',
51 |             renderer='templates/edit.pt',
52 |             permission='edit')
53 | def add_page(context, request):
54 |     pagename = request.subpath[0]
55 |     if 'form.submitted' in request.params:
56 |         body = request.params['body']
57 |         page = Page(body)
58 |         page.__name__ = pagename
59 |         page.__parent__ = context
60 |         context[pagename] = page
61 |         return HTTPFound(location = request.resource_url(page))
62 |     save_url = request.resource_url(context, 'add_page', pagename)
63 |     page = Page('')
64 |     page.__name__ = pagename
65 |     page.__parent__ = context
66 |
```

```

67         return dict(page=page, save_url=save_url,
68                     logged_in=request.authenticated_userid)
69
70 @view_config(name='edit_page', context='.models.Page',
71             renderer='templates/edit.pt',
72             permission='edit')
73 def edit_page(context, request):
74     if 'form.submitted' in request.params:
75         context.data = request.params['body']
76         return HTTPFound(location = request.resource_url(context))
77
78     return dict(page=context,
79                 save_url=request.resource_url(context, 'edit_page'),
80                 logged_in=request.authenticated_userid)
81
82 @view_config(context='.models.Wiki', name='login',
83             renderer='templates/login.pt')
84 @forbidden_view_config(renderer='templates/login.pt')
85 def login(request):
86     login_url = request.resource_url(request.context, 'login')
87     referrer = request.url
88     if referrer == login_url:
89         referrer = '/' # never use the login form itself as came_from
90     came_from = request.params.get('came_from', referrer)
91     message = ''
92     login = ''
93     password = ''
94     if 'form.submitted' in request.params:
95         login = request.params['login']
96         password = request.params['password']
97         if USERS.get(login) == password:
98             headers = remember(request, login)
99             return HTTPFound(location = came_from,
100                             headers = headers)
101         message = 'Failed login'
102
103     return dict(
104         message = message,
105         url = request.application_url + '/login',
106         came_from = came_from,
107         login = login,
108         password = password,
109     )
110
111 @view_config(context='.models.Wiki', name='logout')
112 def logout(request):

```

```
113     headers = forget(request)
114     return HTTPFound(location = request.resource_url(request.context),
115                     headers = headers)
```

(Only the highlighted lines need to be added.)

Our tutorial/tutorial/templates/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="/static/favicon.ico" />
13   <link rel="stylesheet"
14     href="/static/pylons.css"
15     type="text/css" media="screen" charset="utf-8" />
16   <!--[if lte IE 6]>
17   <link rel="stylesheet"
18     href="/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 </body>
58 </html>

```

(Only the highlighted lines need to be added.)

Our tutorial/tutorial/templates/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="/static/favicon.ico" />
13     <link rel="stylesheet"
14         href="/static/pylons.css"
15         type="text/css" media="screen" charset="utf-8" />
16     <!--[if lte IE 6]>
17     <link rel="stylesheet"

```

```
18         href="/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>
60 </body>
61 </html>
```

(Only the highlighted lines need to be added.)

41.7.4 Viewing the Application in a Browser

We can finally examine our application in a browser (See *Start the Application*). Launch a browser and visit each of the following URLs, check that the result is as expected:

- `http://localhost:6543/` invokes the `view_wiki` view. This always redirects to the `view_page` view of the `FrontPage` Page resource. It is executable by any user.
- `http://localhost:6543/FrontPage` 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.
- `http://localhost:6543/FrontPage/edit_page` 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.
- `http://localhost:6543/add_page/SomePageName` 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.
- After logging 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.

41.8 Adding Tests

We will now add tests for the models and the views and a few functional tests in the `tests.py`. Tests ensure that an application works, and that it continues to work after some changes are made in the future.

41.8.1 Test the Models

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 that was generated as part of the `zodb` scaffold. 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`.

41.8.2 Test 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.

41.8.3 Functional tests

We test the whole application, covering security aspects that are not tested in the unit tests, like logging in, logging out, checking that the `viewer` user cannot add or edit pages, but the `editor` user can, and so on.

41.8.4 View the results of all our edits to `tests.py`

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 .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 .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 .models import appmaker
36             return appmaker(zodb_root)
37
38         def test_it(self):
39             root = {}
40             self._callFUT(root)
41             self.assertEqual(root['app_root']['FrontPage'].data,
42                             'This is the front page')
43
44     class ViewWikiTests(unittest.TestCase):
45         def test_it(self):
46             from .views import view_wiki
47             context = testing.DummyResource()
48             request = testing.DummyRequest()
49             response = view_wiki(context, request)
50             self.assertEqual(response.location, 'http://example.com/FrontPage')
51
52     class ViewPageTests(unittest.TestCase):
53         def _callFUT(self, context, request):
54             from .views import view_page
55             return view_page(context, request)
56
57         def test_it(self):
58             wiki = testing.DummyResource()
59             wiki['IDoExist'] = testing.DummyResource()
60             context = testing.DummyResource(data='Hello CruelWorld IDoExist')
61             context.__parent__ = wiki
62             context.__name__ = 'thepage'
63             request = testing.DummyRequest()
64             info = self._callFUT(context, request)
65             self.assertEqual(info['page'], context)
66             self.assertEqual(
67                 info['content'],
68                 '<div class="document">\n'
69                 '<p>Hello <a href="http://example.com/add_page/CruelWorld">'
70                 'CruelWorld</a> '

```

```
71         '<a href="http://example.com/IDoExist/">'
72         'IDoExist</a>'
73         '</p>\n</div>\n')
74     self.assertEqual(info['edit_url'],
75                       'http://example.com/thepage/edit_page')
76
77
78 class AddPageTests(unittest.TestCase):
79     def _callFUT(self, context, request):
80         from .views import add_page
81         return add_page(context, request)
82
83     def test_it_notsubmitted(self):
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             request.resource_url(context, '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 .views import edit_page
107         return edit_page(context, request)
108
109     def test_it_notsubmitted(self):
110         context = testing.DummyResource()
111         request = testing.DummyRequest()
112         info = self._callFUT(context, request)
113         self.assertEqual(info['page'], context)
114         self.assertEqual(info['save_url'],
115                           request.resource_url(context, 'edit_page'))
116
```

```

117     def test_it_submitted(self):
118         context = testing.DummyResource()
119         request = testing.DummyRequest({'form.submitted':True,
120                                         'body':'Hello yo!'})
121         response = self._callFUT(context, request)
122         self.assertEqual(response.location, 'http://example.com/')
123         self.assertEqual(context.data, 'Hello yo!')
124
125     class FunctionalTests(unittest.TestCase):
126
127         viewer_login = '/login?login=viewer&password=viewer' \
128                        '&came_from=FrontPage&form.submitted=Login'
129         viewer_wrong_login = '/login?login=viewer&password=incorrect' \
130                              '&came_from=FrontPage&form.submitted=Login'
131         editor_login = '/login?login=editor&password=editor' \
132                       '&came_from=FrontPage&form.submitted=Login'
133
134     def setUp(self):
135         import tempfile
136         import os.path
137         from . import main
138         self.tmpdir = tempfile.mkdtemp()
139
140         dbpath = os.path.join( self.tmpdir, 'test.db')
141         uri = 'file://' + dbpath
142         settings = { 'zodbconn.uri' : uri ,
143                     'pyramid.includes': ['pyramid_zodbconn', 'pyramid_tm'] }
144
145         app = main({}, **settings)
146         self.db = app.registry._zodb_databases['']
147         from webtest import TestApp
148         self.testapp = TestApp(app)
149
150     def tearDown(self):
151         import shutil
152         self.db.close()
153         shutil.rmtree( self.tmpdir )
154
155     def test_root(self):
156         res = self.testapp.get('/', status=302)
157         self.assertEqual(res.location, 'http://localhost/FrontPage')
158
159     def test_FrontPage(self):
160         res = self.testapp.get('/FrontPage', status=200)
161         self.assertTrue(b'FrontPage' in res.body)
162

```

```
163 def test_unexisting_page(self):
164     res = self.testapp.get('/SomePage', status=404)
165     self.assertTrue(b'Not Found' in res.body)
166
167 def test_successful_log_in(self):
168     res = self.testapp.get(self.viewer_login, status=302)
169     self.assertEqual(res.location, 'http://localhost/FrontPage')
170
171 def test_failed_log_in(self):
172     res = self.testapp.get(self.viewer_wrong_login, status=200)
173     self.assertTrue(b'login' in res.body)
174
175 def test_logout_link_present_when_logged_in(self):
176     res = self.testapp.get(self.viewer_login, status=302)
177     res = self.testapp.get('/FrontPage', status=200)
178     self.assertTrue(b'Logout' in res.body)
179
180 def test_logout_link_not_present_after_logged_out(self):
181     res = self.testapp.get(self.viewer_login, status=302)
182     res = self.testapp.get('/FrontPage', status=200)
183     res = self.testapp.get('/logout', status=302)
184     self.assertTrue(b'Logout' not in res.body)
185
186 def test_anonymous_user_cannot_edit(self):
187     res = self.testapp.get('/FrontPage/edit_page', status=200)
188     self.assertTrue(b'Login' in res.body)
189
190 def test_anonymous_user_cannot_add(self):
191     res = self.testapp.get('/add_page/NewPage', status=200)
192     self.assertTrue(b'Login' in res.body)
193
194 def test_viewer_user_cannot_edit(self):
195     res = self.testapp.get(self.viewer_login, status=302)
196     res = self.testapp.get('/FrontPage/edit_page', status=200)
197     self.assertTrue(b'Login' in res.body)
198
199 def test_viewer_user_cannot_add(self):
200     res = self.testapp.get(self.viewer_login, status=302)
201     res = self.testapp.get('/add_page/NewPage', status=200)
202     self.assertTrue(b'Login' in res.body)
203
204 def test_editors_member_user_can_edit(self):
205     res = self.testapp.get(self.editor_login, status=302)
206     res = self.testapp.get('/FrontPage/edit_page', status=200)
207     self.assertTrue(b'Editing' in res.body)
208
```

```

209     def test_editors_member_user_can_add(self):
210         res = self.testapp.get(self.editor_login, status=302)
211         res = self.testapp.get('/add_page/NewPage', status=200)
212         self.assertTrue(b'Editing' in res.body)
213
214     def test_editors_member_user_can_view(self):
215         res = self.testapp.get(self.editor_login, status=302)
216         res = self.testapp.get('/FrontPage', status=200)
217         self.assertTrue(b'FrontPage' in res.body)

```

41.8.5 Running the Tests

We can run these tests by using `setup.py test` in the same way we did in *Run the Tests*. However, first we must edit our `setup.py` to include a dependency on `WebTest`, which we've used in our `tests.py`. Change the `requires` list in `setup.py` to include `WebTest`.

```

1  requires = [
2      'pyramid',
3      'pyramid_chameleon',
4      'pyramid_zodbconn',
5      'transaction',
6      'pyramid_tm',
7      'pyramid_debugtoolbar',
8      'ZODB3',
9      'waitress',
10     'docutils',
11     'WebTest', # add this
12 ]

```

After we've added a dependency on `WebTest` in `setup.py`, we need to rerun `setup.py develop` to get `WebTest` installed into our virtualenv. Assuming our shell's current working directory is the “tutorial” distribution directory:

On UNIX:

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

On Windows:

41. ZODB + TRAVERSAL WIKI TUTORIAL

```
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py develop
```

Once that command has completed successfully, we can run the tests themselves:

On UNIX:

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

On Windows:

```
c:\pyramidtut\tutorial> %VENV%\Scripts\python setup.py test -q
```

The expected result looks something like:

```
.....
-----
Ran 23 tests in 1.653s

OK
```

41.9 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:

```
$ $VENV/bin/python setup.py sdist
```

On Windows:

```
c:\pyramidtut> %VENV%\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.

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.

```
$ cd ~
$ mkdir modwsgi
$ cd modwsgi
$ /usr/local/bin/virtualenv env
```

5. Install Pyramid into the newly created virtualenv:

```
$ cd ~/modwsgi/env
$ $VENV/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
$ $VENV/bin/pcreate -s starter myapp
$ cd myapp
$ $VENV/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, setup_logging
ini_path = '/Users/chris/modwsgi/env/myapp/production.ini'
setup_logging(ini_path)
application = get_app(ini_path, 'main')
```

The first argument to `get_app` is the project configuration file name. It's best to use the `production.ini` file provided by your scaffold, 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.

The call to `setup_logging` initializes the standard library's *logging* module to allow logging within your application. See *Logging Configuration*.

There is no need to make the `pyramid.wsgi` script executable. However, you'll need to make sure that *two* users have access to change into the `~/modwsgi/env` directory: your current user (mine is `chris` and the user that Apache will run as often named `apache` or `httpd`). Make sure both of these users can "cd" into that directory.

-
8. 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. See
# http://stackoverflow.com/a/10558360/209039
WSGIApplicationGroup %{GLOBAL}
WSGIPassAuthorization On
WSGIDaemonProcess pyramid user=chrism group=staff 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>
```

9. Restart Apache

```
$ sudo /usr/sbin/apachectl restart
```

10. 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 Documentation

pyramid.authentication

43.1 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, de-  

                                     bug=False, hashalg=<object object at  

                                     0x7fd58ac0b380>, parent_domain=False, do-  

                                     main=None)
```

A Pyramid *authentication policy* which obtains data from a Pyramid “auth ticket” cookie.



The default hash algorithm used in this policy is MD5 and has known hash collision vulnerabilities. The risk of an exploit is low. However, for improved authentication security, use `hashalg='sha512'`.

Constructor Arguments

`secret`

The secret (a string) used for `auth_tkt` cookie signing. This value should be unique across all values provided to Pyramid for various subsystem secrets (see *Admonishment Against Secret-Sharing*). 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 principal identifiers (possibly empty) if the user does exist. If `callback` is `None`, the `userid` will be assumed to exist with no principals. 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.

For IPv6 this option is not recommended. The `mod_auth_tkt` specification does not specify how to handle IPv6 addresses, so using this option in combination with IPv6 addresses may cause an incompatible cookie. It ties the authentication ticket to that individual's IPv6 address.

`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 automatically reissued as the result of a request which requires authentication. The duration is measured as the number of seconds since the last `auth_tkt` cookie was issued and 'now'. If this value is 0, a new ticket cookie will be reissued on every request which requires authentication.

A good rule of thumb: if you want auto-expired cookies based on inactivity: set the `timeout` value to 1200 (20 mins) and set the `reissue_time` value to perhaps a tenth of the `timeout` value (120 or 2 mins). It's nonsensical to set the `timeout` value lower than the `reissue_time` value, as the ticket will never be reissued if so. However, such a configuration is not explicitly prevented.

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. If your site is hosted as `example.com` this will make the cookie available for sites underneath `example.com` such as `www.example.com`. Optional.

`parent_domain`

Default: `False`. An `auth_tkt` cookie will be generated for the parent domain of the current site. For example if your site is hosted under `www.example.com` a cookie will be generated for `.example.com`. This can be useful if you have multiple sites sharing the same domain. This option supercedes the `wild_domain` option. Optional.

This option is available as of Pyramid 1.5.

`domain`

Default: `None`. If provided the `auth_tkt` cookie will only be set for this domain. This option is not compatible with `wild_domain` and `parent_domain`. Optional.

This option is available as of Pyramid 1.5.

`hashalg`

Default: `md5` (the literal string).

Any hash algorithm supported by Python's `hashlib.new()` function can be used as the `hashalg`.

Cookies generated by different instances of `AuthTktAuthenticationPolicy` using different `hashalg` options are not compatible. Switching the `hashalg` will imply that all existing users with a valid cookie will be required to re-login.

A warning is emitted at startup if an explicit `hashalg` is not passed. This is for backwards compatibility reasons.

This option is available as of Pyramid 1.4.

Optional.



`md5` is the default for backwards compatibility reasons. However, if you don't specify `md5` as the `hashalg` explicitly, a warning is issued at application startup time. An explicit value of `sha512` is recommended for improved security, and `sha512` will become the default in a future Pyramid version.

`debug`

Default: `False`. If `debug` is `True`, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

Objects of this class implement the interface described by `pyramid.interfaces.IAuthenticationPolicy`.

`authenticated_userid` (*request*)

Return the authenticated userid or `None`.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a `callback` is registered, this will return the `userid` if and only if the callback returns a value that is not `None`.

`effective_principals` (*request*)

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated `userid`, or the `callback` returns `None`, this will be the only principal:

```
return [Everyone]
```

If the callback does not return `None` and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the callback:

```
extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals
```

forget (*request*)

A list of headers which will delete appropriate cookies.

remember (*request*, *principal*, ***kw*)

Accepts the following kw args: `max_age=<int-seconds>`,
`'tokens=<sequence-of-ascii-strings>'`.

Return a list of headers which will set appropriate cookies on the response.

unauthenticated_userid (*request*)

The `userid` key within the `auth_tkt` cookie.

class RemoteUserAuthenticationPolicy (*environ_key='REMOTE_USER'*, *call-*
back=None, debug=False)

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 principal identifiers (possibly empty) representing groups if the user does exist. If `callback` is `None`, the userid will be assumed to exist with no group principals.

`debug`

Default: False. If debug is True, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

Objects of this class implement the interface described by `pyramid.interfaces.IAuthenticationPolicy`.

authenticated_userid (*request*)

Return the authenticated userid or None.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a callback is registered, this will return the userid if and only if the callback returns a value that is not None.

effective_principals (*request*)

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated userid, or the callback returns None, this will be the only principal:

```
return [Everyone]
```

If the callback does not return None and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the callback:

```
extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals
```

forget (*request*)

A no-op. The REMOTE_USER does not provide a protocol for forgetting the user. This will be application-specific and can be done somewhere else or in a subclass.

remember (*request*, *principal*, ***kw*)

A no-op. The REMOTE_USER does not provide a protocol for remembering the user. This will be application-specific and can be done somewhere else or in a subclass.

unauthenticated_userid (*request*)

The REMOTE_USER value found within the environ.

class SessionAuthenticationPolicy (*prefix='auth.', callback=None, debug=False*)

A Pyramid authentication policy which gets its data from the configured *session*. For this authentication policy to work, you will have to follow the instructions in the *Sessions* to configure a *session factory*.

Constructor Arguments

`prefix`

A prefix used when storing the authentication parameters in the session. Defaults to 'auth.'. Optional.

`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 principal identifiers (possibly empty) if the user does exist. If `callback` is `None`, the `userid` will be assumed to exist with no principals. Optional.

`debug`

Default: `False`. If `debug` is `True`, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

authenticated_userid (*request*)

Return the authenticated userid or `None`.

If no `callback` is registered, this will be the same as `unauthenticated_userid`.

If a `callback` is registered, this will return the `userid` if and only if the `callback` returns a value that is not `None`.

effective_principals (*request*)

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated userid, or the `callback` returns `None`, this will be the only principal:

```
return [Everyone]
```

If the `callback` does not return `None` and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the `callback`:

```
extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals
```

forget (*request*)

Remove the stored principal from the session.

remember (*request, principal, **kw*)

Store a principal in the session.

class BasicAuthAuthenticationPolicy (*check, realm='Realm', debug=False*)

A Pyramid authentication policy which uses HTTP standard basic authentication protocol to authenticate users. To use this policy you will need to provide a callback which checks the supplied user credentials against your source of login data.

Constructor Arguments

check

A callback function passed a username, password and request, in that order as positional arguments. Expected to return `None` if the `userid` doesn't exist or a sequence of principal identifiers (possibly empty) if the user does exist.

realm

Default: "Realm". The Basic Auth Realm string. Usually displayed to the user by the browser in the login dialog.

debug

Default: `False`. If `debug` is `True`, log messages to the Pyramid debug logger about the results of various authentication steps. The output from debugging is useful for reporting to maillist or IRC channels when asking for support.

Issuing a challenge

Regular browsers will not send username/password credentials unless they first receive a challenge from the server. The following recipe will register a view that will send a Basic Auth challenge to the user whenever there is an attempt to call a view which results in a Forbidden response:

```

from pyramid.httpexceptions import HTTPUnauthorized
from pyramid.security import forget
from pyramid.view import forbidden_view_config

@forbidden_view_config()
def basic_challenge(request):
    response = HTTPUnauthorized()
    response.headers.update(forget(request))
    return response

```

authenticated_userid(*request*)

Return the authenticated userid or None.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a callback is registered, this will return the userid if and only if the callback returns a value that is not None.

effective_principals(*request*)

A list of effective principals derived from request.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no authenticated userid, or the callback returns None, this will be the only principal:

```

return [Everyone]

```

If the callback does not return None and an authenticated userid is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the callback:

```

extra_principals = callback(userid, request)
return [Everyone, Authenticated, userid] + extra_principals

```

forget(*request*)

Returns challenge headers. This should be attached to a response to indicate that credentials are required.

remember(*request*, *principal*, ***kw*)

A no-op. Basic authentication does not provide a protocol for remembering the user. Credentials are sent on every request.

unauthenticated_userid (*request*)

The userid parsed from the `Authorization` request header.

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 principal identifiers (possibly empty) representing groups if the user does exist. If `callback` is `None`, the userid will be assumed to exist with no group principals.

Objects of this class implement the interface described by `pyramid.interfaces.IAuthenticationPolicy`.

authenticated_userid (*request*)

Return the authenticated userid or `None`.

If no callback is registered, this will be the same as `unauthenticated_userid`.

If a callback is registered, this will return the userid if and only if the callback returns a value that is not `None`.

effective_principals (*request*)

A list of effective principals derived from the identity.

This will return a list of principals including, at least, `pyramid.security.Everyone`. If there is no identity, or the callback returns `None`, this will be the only principal.

If the callback does not return `None` and an identity is found, then the principals will include `pyramid.security.Authenticated`, the `authenticated_userid` and the list of principals returned by the callback.

forget (*request*)

Forget the current authenticated user.

Return headers that, if included in a response, will delete the cookie responsible for tracking the current user.

remember (*request*, *principal*, ***kw*)

Store the *principal* as `repoze.who.userid`.

The identity to authenticated to `repoze.who` will contain the given *principal* as `userid`, and provide all keyword arguments as additional identity keys. Useful keys could be `max_age` or `userdata`.

unauthenticated_userid (*request*)

Return the `repoze.who.userid` key from the detected identity.

43.2 Helper Classes

```
class AuthTktCookieHelper (secret, cookie_name='auth_tkt', se-  
                           cure=False, include_ip=False, time-  
                           out=None, reissue_time=None, max_age=None,  
                           http_only=False, path='/', wild_domain=True,  
                           hashalg='md5', parent_domain=False, do-  
                           main=None)
```

A helper class for use in third-party authentication policy implementations. See `pyramid.authentication.AuthTktAuthenticationPolicy` for the meanings of the constructor arguments.

```
class AuthTicket (secret, userid, ip, tokens=(), user_data='', time=None,  
                  cookie_name='auth_tkt', secure=False, hashalg='md5')
```

This class represents an authentication token. You must pass in the shared *secret*, the *userid*, and the IP address. Optionally you can include *tokens* (a list of strings, representing role names), *'user_data'*, which is arbitrary data available for your own use in later scripts. Lastly, you can override the cookie name and timestamp.

Once you provide all the arguments, use `.cookie_value()` to generate the appropriate authentication ticket.

Usage:

```
token = AuthTicket('sharedsecret', 'username',
    os.environ['REMOTE_ADDR'], tokens=['admin'])
val = token.cookie_value()
```

exception `AuthTktCookieHelper.BadTicket(msg, expected=None)`

Exception raised when a ticket can't be parsed. If we get far enough to determine what the expected digest should have been, `expected` is set. This should not be shown by default, but can be useful for debugging.

`AuthTktCookieHelper.forget(request)`

Return a set of expires Set-Cookie headers, which will destroy any existing `auth_tkt` cookie when attached to a response

`AuthTktCookieHelper.identify(request)`

Return a dictionary with authentication information, or `None` if no valid `auth_tkt` is attached to request

static `AuthTktCookieHelper.parse_ticket(secret, ticket, ip,`
`hashalg='md5')`

Parse the ticket, returning (timestamp, userid, tokens, user_data).

If the ticket cannot be parsed, a `BadTicket` exception will be raised with an explanation.

`AuthTktCookieHelper.remember(request, userid, max_age=None, to-`
`kens=())`

Return a set of Set-Cookie headers; when set into a response, these headers will represent a valid authentication ticket.

max_age The max age of the `auth_tkt` cookie, in seconds. 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. If this value is `None`, the `max_age` value provided to the helper itself will be used as the `max_age` value. Default: `None`.

tokens A sequence of strings that will be placed into the `auth_tkt` tokens field. Each string in the sequence must be of the Python `str` type and must match the regex `^[A-Za-z][A-Za-z0-9+_-]*$`. Tokens are available in the returned identity when an `auth_tkt` is found in the request and unpacked. Default: `()`.

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.

Objects of this class implement the `pyramid.interfaces.IAuthorizationPolicy` interface.

pyramid.compat

The `pyramid.compat` module provides platform and version compatibility for Pyramid and its add-ons across Python platform and version differences. APIs will be removed from this module over time as Pyramid ceases to support systems which require compatibility imports.

ascii_native_(s)

Python 3: If `s` is an instance of `text_type`, return `s.encode('ascii')`, otherwise return `str(s, 'ascii', 'strict')`

Python 2: If `s` is an instance of `text_type`, return `s.encode('ascii')`, otherwise return `str(s)`

binary_type

Binary type for this platform. For Python 3, it's `bytes`. For Python 2, it's `str`.

bytes_(s, encoding='latin-1', errors='strict')

If `s` is an instance of `text_type`, return `s.encode(encoding, errors)`, otherwise return `s`

class_types

Sequence of class types for this platform. For Python 3, it's `(type,)`. For Python 2, it's `(type, types.ClassType)`.

configparser

On Python 2, the `ConfigParser` module, on Python 3, the `configparser` module.

escape(v)

On Python 2, the `cgi.escape` function, on Python 3, the `html.escape` function.

exec_ (*code*, *globs=None*, *locs=None*)

Exec code in a compatible way on both Python 2 and 3.

im_func

On Python 2, the string value `im_func`, on Python 3, the string value `__func__`.

input_ (*v*)

On Python 2, the `raw_input` function, on Python 3, the `input` function.

integer_types

Sequence of integer types for this platform. For Python 3, it's `(int,)`. For Python 2, it's `(int, long)`.

is_nonstr_iter (*v*)

Return `True` if *v* is a non-str iterable on both Python 2 and Python 3.

iteritems_ (*d*)

Return `d.items()` on Python 3, `d.iteritems()` on Python 2.

intervalues_ (*d*)

Return `d.values()` on Python 3, `d.intervalues()` on Python 2.

iterkeys_ (*d*)

Return `d.keys()` on Python 3, `d.iterkeys()` on Python 2.

long

Long type for this platform. For Python 3, it's `int`. For Python 2, it's `long`.

map_ (*v*)

Return `list(map(v))` on Python 3, `map(v)` on Python 2.

pickle

`cPickle` module if it exists, `pickle` module otherwise.

PY3

`True` if running on Python 3, `False` otherwise.

PYPY

`True` if running on PyPy, `False` otherwise.

reraise (*tp*, *value*, *tb=None*)

Reraise an exception in a compatible way on both Python 2 and Python 3, e.g. `reraise(*sys.exc_info())`.

string_types

Sequence of string types for this platform. For Python 3, it's `(str,)`. For Python 2, it's `(basestring,)`.

SimpleCookie

On Python 2, the `Cookie.SimpleCookie` class, on Python 3, the `http.cookies.SimpleCookie` module.

text_*(s, encoding='latin-1', errors='strict')*

If `s` is an instance of `binary_type`, return `s.decode(encoding, errors)`, otherwise return `s`

text_type

Text type for this platform. For Python 3, it's `str`. For Python 2, it's `unicode`.

native_*(s, encoding='latin-1', errors='strict')*

Python 3: If `s` is an instance of `text_type`, return `s`, otherwise return `str(s, encoding, errors)`

Python 2: If `s` is an instance of `text_type`, return `s.encode(encoding, errors)`, otherwise return `str(s)`

urlparse

`urlparse` module on Python 2, `urllib.parse` module on Python 3.

url_quote

`urllib.quote` function on Python 2, `urllib.parse.quote` function on Python 3.

url_quote_plus

`urllib.quote_plus` function on Python 2, `urllib.parse.quote_plus` function on Python 3.

url_unquote

`urllib.unquote` function on Python 2, `urllib.parse.unquote` function on Python 3.

url_encode

`urllib.urlencode` function on Python 2, `urllib.parse.urlencode` function on Python 3.

url_open

`urllib2.urlopen` function on Python 2, `urllib.request.urlopen` function on Python 3.

url_unquote_text *(v, encoding='utf-8', errors='replace')*

On Python 2, return `url_unquote(v).decode(encoding, errors)`; on Python 3, return the result of `urllib.parse.unquote`.

url_unquote_native *(v, encoding='utf-8', errors='replace')*

On Python 2, return `native_(url_unquote_text_v, encoding, errors)`; on Python 3, return the result of `urllib.parse.unquote`.

pyramid.config

```
class Configurator (registry=None, package=None, settings=None, root_factory=None,
                    authentication_policy=None, authorization_policy=None, renderers=None,
                    debug_logger=None, locale_negotiator=None, request_factory=None,
                    default_permission=None, session_factory=None, default_view_mapper=None,
                    autocommit=False, exception_response_view=<function default_exceptionresponse_view at
                    0x7fd58a66e230>, route_prefix=None, introspection=True)
```

A Configurator is used to configure a Pyramid *application* registry.

If the `registry` argument is not `None`, 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 its construction, the configurator may be used to add further configuration to the registry.



If `registry` is assigned the above-mentioned class instance, all other constructor arguments are ignored, with the exception of `package`.

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 the 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 the 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 the same.

If `authorization_policy` is passed, it should be an instance of an *authorization policy* or a *dotted Python name* to the same.



A `ConfigurationError` will be raised when an authorization policy is supplied without also supplying an authentication policy (authorization requires authentication).

If `renderers` is `None` (the default), a default set of *renderer* factories is used. Else, it should be a list of tuples representing a set of renderer factories which should be configured into this application, and each tuple representing a set of positional values that should be passed to `pyramid.config.Configurator.add_renderer()`.

If `debug_logger` is not passed, a default debug logger that logs to a logger will be used (the logger name will be the package name of the *caller* of this configurator). If it is passed, it should be an instance of the `logging.Logger` (PEP 282) standard library class or a Python logger name. 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 the same. See *Changing the Request Factory*. By default it is `None`, which means use the default request 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:

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 `default_view_mapper` is not passed, a superdefault view mapper will be used.

If `exceptionresponse_view` is passed, it must be a *view callable* or `None`. If it is a view callable, it will be used as an exception view callable when an *exception response* is raised. If `exceptionresponse_view` is `None`, no exception response view will be registered, and all raised exception responses will be bubbled up to Pyramid's caller. By default, the `pyramid.httpexceptions.default_exceptionresponse_view` function is used as the `exceptionresponse_view`.

If `route_prefix` is passed, all routes added with `pyramid.config.Configurator.add_route()` will have the specified path prepended to their pattern.

If `introspection` is passed, it must be a boolean value. If it's `True`, introspection values during actions will be kept for use for tools like the debug toolbar. If it's `False`, introspection values provided by registrations will be ignored. By default, it is `True`.

New in version 1.1: The `exceptionresponse_view` argument.

New in version 1.2: The `route_prefix` argument.

New in version 1.3: The `introspection` argument.

Controlling Configuration State

commit()

Commit any pending configuration actions. If a configuration conflict is detected in the pending configuration actions, 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.

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 onto 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 onto the *thread local* stack (usually by the `begin` method) and returns that value.

include(callable, route_prefix=None)

Include a configuration callable, to support imperative application extensibility.



In versions of Pyramid prior to 1.2, this function accepted **callables*, but this has been changed to support only a single callable.

A configuration callable should be a callable that accepts a single argument named `config`, which will be an instance of a *Configurator*. However, be warned that it will not be the same configurator instance on which you call this method. The code which runs as a 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 `callable` 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 to 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.

If the `route_prefix` is supplied, it must be a string. Any calls to `pyramid.config.Configurator.add_route()` within the included callable will have their pattern prefixed with the value of `route_prefix`. This can be used to help mount a set of routes at a different location than the included callable's author intended, while still maintaining the same route names. For example:

```
1 from pyramid.config import Configurator
2
3 def included(config):
4     config.add_route('show_users', '/show')
5
6 def main(global_config, **settings):
7     config = Configurator()
8     config.include(included, route_prefix='/users')
```

In the above configuration, the `show_users` route will have an effective route pattern of `/users/show`, instead of `/show` because the `route_prefix` argument will be prepended to the pattern.

New in version 1.2: The `route_prefix` parameter.

`make_wsgi_app()`

Commits any pending configuration statements, sends a `pyramid.events.ApplicationCreated` event to all listeners, adds this configuration's registry to `pyramid.config.global_registries`, and returns a Pyramid WSGI application representing the committed configuration state.

`scan(package=None, categories=None, onerror=None, ignore=None, **kw)`

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.

The `onerror` argument, if provided, should be a Venusian `onerror` callback function. The `onerror` function is passed to `venusian.Scanner.scan()` to

influence error behavior when an exception is raised during the scanning process. See the *Venusian* documentation for more information about `onerror` callbacks.

The `ignore` argument, if provided, should be a *Venusian* `ignore` value. Providing an `ignore` argument allows the scan to ignore particular modules, packages, or global objects during a scan. `ignore` can be a string or a callable, or a list containing strings or callables. The simplest usage of `ignore` is to provide a module or package by providing a full path to its dotted name. For example: `config.scan(ignore='my.module.subpackage')` would ignore the `my.module.subpackage` package during a scan, which would prevent the subpackage and any of its submodules from being imported and scanned. See the *Venusian* documentation for more information about the `ignore` argument.

To perform a scan, Pyramid creates a *Venusian* `Scanner` object. The `kw` argument represents a set of keyword arguments to pass to the *Venusian* `Scanner` object's constructor. See the *venusian* documentation (its `Scanner` class) for more information about the constructor. By default, the only keyword arguments passed to the `Scanner` constructor are `{'config':self}` where `self` is this configurator object. This services the requirement of all built-in Pyramid decorators, but extension systems may require additional arguments. Providing this argument is not often necessary; it's an advanced usage.

New in version 1.1: The `**kw` argument.

New in version 1.3: The `ignore` argument.

Adding Routes and Views

`add_route` (*name*, *pattern=None*, *permission=None*, *factory=None*, *for_=None*, *header=None*, *xhr=None*, *accept=None*, *path_info=None*, *request_method=None*, *request_param=None*, *traverse=None*, *custom_predicates=()*, *use_global_views=False*, *path=None*, *pregenerator=None*, *static=False*, ***predicates*)

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 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. See *The Resource Tree* for more information about root factories.

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.request.Request.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.

`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).

`static`

If `static` is `True`, this route will never match an incoming request; it will only be useful for URL generation. By default, `static` is `False`. See *Static Routes*.

New in version 1.1.

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` or a tuple of elements containing HTTP method names. 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.

Changed in version 1.2: The ability to pass a tuple of items as `request_method`. Previous versions allowed only a string.

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_param="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, or if the `Accept` header isn't set at all in the request, this predicate will be true. If this predicate returns `False`, route matching continues.

effective_principals

If specified, this value should be a *principal* identifier or a sequence of principal identifiers. If the `pyramid.request.Request.effective_principals`

property indicates that every principal named in the argument list is present in the current request, this predicate will return `True`; otherwise it will return `False`. For example: `effective_principals=pyramid.security.Authenticated` or `effective_principals=('fred', 'group:admins')`.

New in version 1.4a4.

`custom_predicates`

Deprecated since version 1.5: 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`.

`predicates`

Pass a key/value pair here to use a third-party predicate registered via `pyramid.config.Configurator.add_view_predicate()`. More than one key/value pair can be used at the same time. See *View and Route Predicates* for more information about third-party predicates.

New in version 1.4.

`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 `pyramid.security.NO_PERMISSION_REQUIRED`, 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. If `permission` is specified, the security checking will be performed against the default root factory ACL.

Any other keyword arguments sent to `add_static_view` are passed on to `pyramid.config.Configurator.add_route()` (e.g. `factory`, perhaps to define a custom factory with a custom ACL for this static view).

Usage

The `add_static_view` function is typically used in conjunction with the `pyramid.request.Request.static_url()` method. `add_static_view` adds a view which renders a static asset when some URL is visited; `pyramid.request.Request.static_url()` generates a URL to that asset.

The `name` argument to `add_static_view` is usually a simple URL prefix (e.g. `'images'`). When this is the case, the `pyramid.request.Request.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.request.Request.static_url()`:

```
request.static_url('mypackage:images/logo.png')
```

When `add_static_view` is called with a `name` argument that represents a URL prefix, as it is above, subsequent calls to `pyramid.request.Request.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.request.Request.static_url()`. Furthermore, if a protocol-relative URL (e.g. `//example.com/images`) is used as the `name` argument, the generated URL will use the protocol of the request (`http` or `https`, respectively).

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.request.Request.static_url()` for that static view will be prefixed with `http://example.com/images` (the external webserver listening on `example.com` must be itself configured to respond properly to such a request.):

```
static_url('mypackage:images/logo.png', request)
```

See *Serving Static Assets* for more information.

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=None, accept=None, header=None, path_info=None, custom_predicates=(), context=None, decorator=None, mapper=None, http_cache=None, match_param=None, check_csrf=None, **predicates*)

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

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. This is often a string like `view` or `edit`.

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 value `pyramid.security.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

This knob is most useful when the view definition is a class.

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.

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 unmodified).

`http_cache`

New in version 1.1.

When you supply an `http_cache` value to a view configuration, the `Expires` and `Cache-Control` headers of a response generated by the associated view callable are modified. The value for `http_cache` may be one of the following:

- A nonzero integer. If it's a nonzero integer, it's treated as a number of seconds. This number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=3600` instructs the requesting browser to ‘cache this response for an hour, please’.
- A `datetime.timedelta` instance. If it's a `datetime.timedelta` instance, it will be converted into a number of seconds, and that number of seconds will be used to compute the `Expires` header and the `Cache-Control: max-age` parameter of responses to requests which call this view. For example: `http_cache=datetime.timedelta(days=1)` instructs the requesting browser to ‘cache this response for a day, please’.
- Zero (0). If the value is zero, the `Cache-Control` and `Expires` headers present in all responses from this view will be composed such that client browser cache (and any intermediate caches) are instructed to never cache the response.
- A two-tuple. If it's a two tuple (e.g. `http_cache=(1, {'public':True})`), the first value in the tuple may be a nonzero integer or a `datetime.timedelta` instance; in either case this value will be used as the number of seconds to cache the response. The second value in the tuple must be a dictionary. The values present in the dictionary will be used as input to the `Cache-Control` response header. For example: `http_cache=(3600, {'public':True})` means ‘cache for an hour, and add `public` to the `Cache-Control` header of the response’. All keys and values supported by the `webob.cachecontrol.CacheControl` interface may be added to the dictionary. Supplying `{'public':True}` is equivalent to calling `response.cache_control.public = True`.

Providing a non-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value)` within your view's body.

Providing a two-tuple value as `http_cache` is equivalent to calling `response.cache_expires(value[0], **value[1])` within your view's body.

If you wish to avoid influencing, the `Expires` header, and instead wish to only influence `Cache-Control` headers, pass a tuple as `http_cache` with the first element of `None`, e.g.: `(None, {'public': True})`.

If you wish to prevent a view that uses `http_cache` in its configuration from having its caching response headers changed by this machinery, set `response.cache_control.prevent_auto = True` before returning the response from the view. This effectively disables any HTTP caching done by `http_cache` for that response.

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 Configuration*. 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, or an iterable of the aforementioned) which will be used to decorate the registered *view callable*. The decorator function(s) will be called with the view callable as a single argument. The view callable it is passed will accept `(context, request)`. The decorator(s) must return a replacement view callable which also accepts `(context, request)`.

If decorator is an iterable, the callables will be combined and used in the order provided as a decorator. For example:

```
@view_config(...,
    decorator=(decorator2,
               decorator1))
def myview(request):
    ...
```

Is similar to doing:

```
@view_config(...)
@decorator2
@decorator1
def myview(request):
    ...
```

Except with the existing benefits of `decorator=` (having a common decorator syntax for all view calling conventions and not having to think about preserving function attributes such as `__name__` and `__module__` within decorator logic).

Changed in version 1.4a4: Passing an iterable.

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.

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 be either a strings (such as GET, POST, PUT, DELETE, or HEAD) representing an HTTP REQUEST_METHOD, or a tuple containing one or more of these strings. A view declaration with this argument ensures that the view will only be called when the `method` attribute of the request (aka the REQUEST_METHOD of the WSGI environment) matches a supplied value. Note that use of GET also implies that the view will respond to HEAD as of Pyramid 1.4.

Changed in version 1.2: The ability to pass a tuple of items as `request_method`. Previous versions allowed only a string.

`request_param`

This value can be any string or any sequence of strings. 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 is a string) or values (if the value is a tuple). If any value supplied has a `=` sign in it, e.g. `request_param="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.

`match_param`

New in version 1.2.

This value can be a string of the format “key=value” or a tuple containing one or more of these strings.

A view declaration with this argument ensures that the view will only be called when the *request* has key/value pairs in its *matchdict* that equal those supplied in the predicate. e.g. `match_param="action=edit"` would require the `action` parameter in the *matchdict* match the right hand side of the expression (`edit`) for the view to “match” the current request.

If the `match_param` is a tuple, every key/value pair must match for the predicate to pass.

`containment`

This value should be a Python class or *interface* (or a *dotted Python name*) that an object in the *lineage* of the context 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 mime-types 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`.

check_csrf

If specified, this value should be one of `None`, `True`, `False`, or a string representing the 'check name'. If the value is `True` or a string, CSRF checking will be performed. If the value is `False` or `None`, CSRF checking will not be performed.

If the value provided is a string, that string will be used as the 'check name'. If the value provided is `True`, `csrf_token` will be used as the check name.

If CSRF checking is performed, the checked value will be the value of `request.params[check_name]`. This value will be compared against the value of `request.session.get_csrf_token()`, and the check will pass if these two values are the same. If the check passes, the associated view will be permitted to execute. If the check fails, the associated view will not be permitted to execute.

Note that using this feature requires a *session factory* to have been configured.

New in version 1.4a2.

`physical_path`

If specified, this value should be a string or a tuple representing the *physical path* of the context found via traversal for this predicate to match as true. For example: `physical_path= '/'` or `physical_path= '/a/b/c'` or `physical_path= ('', 'a', 'b', 'c')`. This is not a path prefix match or a regex, it's a whole-path match. It's useful when you want to always potentially show a view when some object is traversed to, but you can't be sure about what kind of object it will be, so you can't use the `context` predicate. The individual path elements inbetween slash characters or in tuple elements should be the Unicode representation of the name of the resource and should not be encoded in any way.

New in version 1.4a3.

`effective_principals`

If specified, this value should be a *principal* identifier or a sequence of principal identifiers. If the `pyramid.request.Request.effective_principals` property indicates that every principal named in the argument list is present in the current request, this predicate will return `True`; otherwise it will return `False`. For example: `effective_principals=pyramid.security.Authenticated` or `effective_principals=('fred', 'group:admins')`.

New in version 1.4a4.

`custom_predicates`

Deprecated since version 1.5: 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. The `predicates` argument to this method and the ability to register third-party view predicates via `pyramid.config.Configurator.add_view_predicate()` obsoletes this argument, but it is kept around for backwards compatibility.

`predicates`

Pass a key/value pair here to use a third-party predicate registered via `pyramid.config.Configurator.add_view_predicate()`. More than one key/value pair can be used at the same time. See *View and Route Predicates* for more information about third-party predicates.

```
add_notfound_view (view=None, attr=None, renderer=None, wrapper=None, route_name=None, request_type=None, request_method=None, request_param=None, containment=None, xhr=None, accept=None, header=None, path_info=None, custom_predicates=(), decorator=None, mapper=None, match_param=None, append_slash=False, **predicates)
```

Add a default Not Found View to the current configuration state. The view will be called when Pyramid or application code raises an `pyramid.httpexceptions.HTTPNotFound` exception (e.g. when a view cannot be found for the request). The simplest example is:

```
def notfound(request):  
    return Response('Not Found', status='404 Not Found')  
  
config.add_notfound_view(notfound)
```

All arguments except `append_slash` have the same meaning as `pyramid.config.Configurator.add_view()` and each predicate argument restricts the set of circumstances under which this notfound view will be invoked. Unlike `pyramid.config.Configurator.add_view()`, this method will raise an exception if passed `name`, `permission`, `context`, `for_`, or `http_cache` keyword arguments. These argument values make no sense in the context of a Not Found View.

If `append_slash` is `True`, when this Not Found View is invoked, and the current path info does not end in a slash, the notfound logic will attempt to find a *route* that matches the request's path info suffixed with a slash. If such a route exists, Pyramid will issue a redirect to the URL implied by the route; if it does not, Pyramid will return the result of the view callable provided as `view`, as normal.

New in version 1.3.

```
add_forbidden_view (view=None, attr=None, renderer=None, wrapper=None, route_name=None, request_type=None, request_method=None, request_param=None, containment=None, xhr=None, accept=None, header=None, path_info=None, custom_predicates=(), decorator=None, mapper=None, match_param=None, **predicates)
```

Add a forbidden view to the current configuration state. The view will be called when Pyramid or application code raises a `pyramid.httpexceptions.HTTPForbidden` exception and the set of circumstances implied by the predicates provided are matched. The simplest example is:

```
def forbidden(request):  
    return Response('Forbidden', status='403 Forbidden')  
  
config.add_forbidden_view(forbidden)
```

All arguments have the same meaning as `pyramid.config.Configurator.add_view()` and each predicate argument restricts the set of circumstances under which this notfound view will be invoked. Unlike `pyramid.config.Configurator.add_view()`, this method will raise an exception if passed `name`, `permission`, `context`, `for_`, or `http_cache` keyword arguments. These argument values make no sense in the context of a forbidden view.

New in version 1.3.

Adding an Event Subscriber

add_subscriber (*subscriber*, *iface=None*, ***predicates*)

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.

Any number of predicate keyword arguments may be passed in ***predicates*. Each predicate named will narrow the set of circumstances in which the subscriber will be invoked. Each named predicate must have been registered via `pyramid.config.Configurator.add_subscriber_predicate()` before it can be used. See *Subscriber Predicates* for more information.

New in version 1.4: The ***predicates* argument.

Using Security

set_authentication_policy (*policy*)

Override the Pyramid *authentication policy* in the current configuration. The `policy` argument must be an instance of an authentication policy or a *dotted Python name* that points at an instance of an authentication policy.



Using the `authentication_policy` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_authorization_policy (*policy*)

Override the Pyramid *authorization policy* in the current configuration. The `policy` argument must be an instance of an authorization policy or a *dotted Python name* that points at an instance of an authorization policy.



Using the `authorization_policy` 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 value of the permission importable as `pyramid.security.NO_PERMISSION_REQUIRED`. 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:

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.

add_permission (*permission_name*)

A configurator directive which registers a free-standing permission without associating it with a view callable. This can be used so that the permission shows up in the introspectable data under the `permissions` category (permissions mentioned via `add_view` already end up in there). For example:

```
config = Configurator()
config.add_permission('view')
```

Extending the Request Object**add_request_method** (*callable=None*, *name=None*, *property=False*, *reify=False*)

Add a property or method to the request object.

When adding a method to the request, `callable` may be any function that receives the request object as the first parameter. If `name` is `None` then it will be computed from the name of the `callable`.

When adding a property to the request, `callable` can either be a callable that accepts the request as its single positional parameter, or it can be a property descriptor. If `name` is `None`, the name of the property will be computed from the name of the `callable`.

If the `callable` is a property descriptor a `ValueError` will be raised if `name` is `None` or `reify` is `True`.

See `pyramid.request.Request.set_property()` for more details on property vs `reify`. When `reify` is `True`, the value of `property` is assumed to also be `True`.

In all cases, `callable` may also be a *dotted Python name* which refers to either a callable or a property descriptor.

If `callable` is `None` then the method is only used to assist in conflict detection between different addons requesting the same attribute on the request object.

This is the recommended method for extending the request object and should be used in favor of providing a custom request factory via `pyramid.config.Configurator.set_request_factory()`.

New in version 1.4.

set_request_property (**args, **kw*)

Using I18N

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')
```

Later calls to `add_translation_dir` insert directories into the beginning of the list of translation directories created by earlier calls. This means that the same translation found in a directory added later in the configuration process will be found before one added earlier in the configuration process. However, if multiple specs are provided in a single call to `add_translation_dirs`, the directories will be inserted into the beginning of the directory list in the order they're provided in the **specs* list argument (items earlier in the list trump ones later in the list).

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.

Overriding Assets

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.

Getting and Adding Settings

add_settings (*settings=None*, ***kw*)

Augment the *deployment settings* with one or more key/value pairs.

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.

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.



the `pyramid.registry.Registry.settings` API performs the same duty.

Hooking Pyramid Behavior

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.

add_resource_url_adapter(adapter, resource_iface=None)

New in version 1.3.

When you add a traverser as described in *Changing the Traverser*, it's convenient to continue to use the `pyramid.request.Request.resource_url()` API. However, since the way traversal is done may have been modified, the URLs that `resource_url` generates by default may be incorrect when resources are returned by a custom traverser.

If you've added a traverser, you can change how `resource_url()` generates a URL for a specific type of resource by calling this method.

The `adapter` argument represents a class that implements the `IResourceURL` interface. The class constructor should accept two arguments in its constructor (the resource and the request) and the resulting instance should provide the attributes detailed in that interface (`virtual_path` and `physical_path`, in particular).

The `resource_iface` argument represents a class or interface that the resource should possess for this url adapter to be used when `pyramid.request.Request.resource_url()` looks up a resource url adapter. If `resource_iface` is not passed, or it is passed as `None`, the url adapter will be used for every type of resource.

See *Changing How `pyramid.request.Request.resource_url()` Generates a URL* for more information.

add_response_adapter (*adapter, type_or_iface*)

When an object of type (or interface) `type_or_iface` is returned from a view callable, Pyramid will use the adapter `adapter` to convert it into an object which implements the `pyramid.interfaces.IResponse` interface. If `adapter` is `None`, an object returned of type (or interface) `type_or_iface` will itself be used as a response object.

`adapter` and `type_or_interface` may be Python objects or strings representing dotted names to importable Python global objects.

See *Changing How Pyramid Treats View Responses* for more information.

add_traverser (*adapter, iface=None*)

The superdefault *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. The section entitled *Changing the Traverser* details how to create a traverser class.

For example, to override the superdefault traverser used by Pyramid, you might do something like this:

```
from myapp.traversal import MyCustomTraverser
config.add_traverser(MyCustomTraverser)
```

This would cause the Pyramid superdefault traverser to never be used; instead all traversal would be done using your `MyCustomTraverser` class, no matter which object was returned by the *root factory* of this application. Note that we passed no arguments to the `iface` keyword parameter. The default value of `iface`, `None` represents that the registered traverser should be used when no other more specific traverser is available for the object returned by the root factory.

However, more than one traversal algorithm can be active at the same time. The traverser used can depend on the result of the *root factory*. For instance, if your root factory returns more than one type of object conditionally, you could claim that an alternate traverser adapter should be used against one particular class or interface returned by that root factory. 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. The `iface` argument represents the class of the object that the root factory might return or an *interface* that the object might implement.

To use a particular traverser only when the root factory returns a particular class:

```
config.add_traverser(MyCustomTraverser, MyRootClass)
```

When more than one traverser is active, the “most specific” traverser will be used (the one that matches the class or interface of the value returned by the root factory most closely).


Note that either `adapter` or `iface` can be a *dotted Python name* or a Python object.

See *Changing the Traverser* for more information.

add_tween (*tween_factory*, *under=None*, *over=None*)

New in version 1.2.

Add a ‘tween factory’. A *tween* (a contraction of ‘between’) is a bit of code that sits between the Pyramid router’s main request handling function and the upstream WSGI component that uses Pyramid as its ‘app’. Tweens are a feature that may be used by Pyramid framework extensions, to provide, for example, Pyramid-specific view timing support, bookkeeping code that examines exceptions before they are returned to the upstream WSGI application, or a variety of other features. Tweens behave a bit like WSGI ‘middleware’ but they have the benefit of running in a context in which they have access to the Pyramid *application registry* as well as the Pyramid rendering machinery.

 You can view the tween ordering configured into a given Pyramid application by using the `ptweens` command. See *Displaying “Tweens”*.

The `tween_factory` argument must be a *dotted Python name* to a global object representing the tween factory.

The `under` and `over` arguments allow the caller of `add_tween` to provide a hint about where in the tween chain this tween factory should be placed when an implicit tween chain is used. These hints are only used when an explicit tween chain is not used (when the `pyramid.tweens` configuration value is not set). Allowable values for `under` or `over` (or both) are:

- `None` (the default).
- A *dotted Python name* to a tween factory: a string representing the dotted name of a tween factory added in a call to `add_tween` in the same configuration session.
- One of the constants `pyramid.tweens.MAIN`, `pyramid.tweens.INGRESS`, or `pyramid.tweens.EXCVIEW`.
- An iterable of any combination of the above. This allows the user to specify fallbacks if the desired tween is not included, as well as compatibility with multiple other tweens.

`under` means ‘closer to the main Pyramid application than’, `over` means ‘closer to the request ingress than’.

For example, calling `add_tween('myapp.tfactory', over=pyramid.tweens.MAIN)` will attempt to place the tween factory represented by the dotted name `myapp.tfactory` directly ‘above’ (in `ptweens` order) the main Pyramid request handler. Likewise, calling `add_tween('myapp.tfactory', over=pyramid.tweens.MAIN, under='mypkg.someothertween')` will attempt to place this tween factory ‘above’ the main handler but ‘below’ (a fictional) ‘`mypkg.someothertween`’ tween factory.

If all options for `under` (or `over`) cannot be found in the current configuration, it is an error. If some options are specified purely for compatibility with other tweens, just add a fallback of `MAIN` or `INGRESS`. For example, `under=('mypkg.someothertween', 'mypkg.someothertween2', INGRESS)`. This constraint will require the tween to be located under both the ‘`mypkg.someothertween`’ tween, the ‘`mypkg.someothertween2`’ tween, and `INGRESS`. If any of these is not in the current configuration, this constraint will only organize itself based on the tweens that are present.

Specifying neither `over` nor `under` is equivalent to specifying `under=INGRESS`.

Implicit tween ordering is obviously only best-effort. Pyramid will attempt to present an implicit order of tweens as best it can, but the only surefire

way to get any particular ordering is to use an explicit tween order. A user may always override the implicit tween ordering by using an explicit `pyramid.tweens` configuration value setting.

`under`, and `over` arguments are ignored when an explicit tween chain is specified using the `pyramid.tweens` configuration value.

For more information, see *Registering Tweens*.

add_route_predicate (*name*, *factory*, *weighs_more_than=None*,
 weighs_less_than=None)
Adds a route predicate factory. The view predicate can later be named as a keyword argument to `pyramid.config.Configurator.add_route()`.

name should be the name of the predicate. It must be a valid Python identifier (it will be used as a keyword argument to `add_view`).

factory should be a *predicate factory* or *dotted Python name* which refers to a predicate factory.

See *View and Route Predicates* for more information.

New in version 1.4.

add_view_predicate (*name*, *factory*, *weighs_more_than=None*,
 weighs_less_than=None)

New in version 1.4.

Adds a view predicate factory. The associated view predicate can later be named as a keyword argument to `pyramid.config.Configurator.add_view()` in the `predicates` anonymous keyword argument dictionary.

name should be the name of the predicate. It must be a valid Python identifier (it will be used as a keyword argument to `add_view` by others).

factory should be a *predicate factory* or *dotted Python name* which refers to a predicate factory.

See *View and Route Predicates* for more information.

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`).

See `pyramid.config.Configurator.add_request_method()` for a less intrusive way to extend the request objects with custom methods and properties.



Using the `request_factory` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_root_factory (*factory*)

Add a *root factory* to the current configuration state. If the *factory* argument is `None` a default root factory will be registered.



Using the `root_factory` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_session_factory (*factory*)

Configure the application with a *session factory*. If this method is called, the *factory* argument must be a session factory callable or a *dotted Python name* to that factory.



Using the `session_factory` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

set_view_mapper (*mapper*)

Setting a *view mapper* makes it possible to make use of *view callable* objects which implement different call signatures than the ones supported by Pyramid as described in its narrative documentation.

The *mapper* should argument be an object implementing `pyramid.interfaces.IViewMapperFactory` or a *dotted*

Python name to such an object. The provided `mapper` will become the default view mapper to be used by all subsequent *view configuration* registrations.

See also:

See also *Using a View Mapper*.



Using the `default_view_mapper` argument to the `pyramid.config.Configurator` constructor can be used to achieve the same purpose.

Extension Author APIs

action (*discriminator*, *callable*=None, *args*=(), *kw*=None, *order*=0, *introspectables*=(), ***extra*)

Register an action which will be executed when `pyramid.config.Configurator.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 task associated with the action when the action is executed. It is optional.

`args` and `kw` are tuple and dict objects respectively, which are passed to `callable` when this action is executed. Both are optional.

`order` is a grouping mechanism; an action with a lower order will be executed before an action with a higher order (has no effect when `autocommit` is `True`).

`introspectables` is a sequence of *introspectable* objects (or the empty sequence if no introspectable objects are associated with this action). If this configurator's `introspection` attribute is `False`, these `introspectables` will be ignored.

`extra` provides a facility for inserting extra keys and values into an action dictionary.

add_directive (*name, directive, action_wrap=True*)

Add a directive method to the configurator.



This method is typically only used by Pyramid framework extension authors, not by Pyramid application developers.

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` 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. `package` may be an actual Python package object or a *dotted Python name* representing a package.

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 method is typically only used by Pyramid framework extension authors, not by Pyramid application developers.

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.

Utility Methods

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.

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.

ZCA-Related APIs

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.

unhook_zca()

Call `zope.component.getSiteManager.reset()` to undo the action of `pyramid.config.Configurator.hook_zca()`.

setup_registry(*settings=None, root_factory=None, authentication_policy=None, authorization_policy=None, renderers=None, debug_logger=None, locale_negotiator=None, request_factory=None, default_permission=None, session_factory=None, default_view_mapper=None, exceptionresponse_view=<function default_exceptionresponse_view at 0x7fd58a66e230>*)

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 a 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.

Testing Helper APIs

`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).

`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_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_securitypolicy (*userid=None, groupids=(), permissive=True, remember_result=None, forget_result=None*)

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.

remember_result, if provided, should be the result returned by the *remember* method of the faux authentication policy. If it is not provided (or it is provided, and is `None`), the default value `[]` (the empty list) will be returned by *remember*.

forget_result, if provided, should be the result returned by the *forget* method of the faux authentication policy. If it is not provided (or it is provided, and is `None`), the default value `[]` (the empty list) will be returned by *forget*.

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.request.Request.authenticated_userid` or `pyramid.request.Request.effective_principals` APIs are used.

This function is most useful when testing code that uses the APIs named `pyramid.request.Request.has_permission()`,

```
pyramid.request.Request.authenticated_userid,  
pyramid.request.Request.effective_principals, and  
pyramid.security.principals_allowed_by_permission().
```

New in version 1.4: The `remember_result` argument.

New in version 1.4: The `forget_result` argument.

Attributes

introspectable

A shortcut attribute which points to the `pyramid.registry.Introspectable` class (used during directives to provide introspection to actions).

New in version 1.3.

introspector

The *introspector* related to this configuration. It is an instance implementing the `pyramid.interfaces.IIntrospector` interface.

New in version 1.3.

registry

The *application registry* which holds the configuration associated with this configurator.

global_registries

The set of registries that have been created for Pyramid applications, one for each call to `pyramid.config.Configurator.make_wsgi_app()` in the current process. The object itself supports iteration and has a `last` property containing the last registry loaded.

The registries contained in this object are stored as weakrefs, thus they will only exist for the lifetime of the actual applications for which they are being used.

class not_(value)

You can invert the meaning of any predicate value by wrapping it in a call to `pyramid.config.not_`.

```
1 from pyramid.config import not_  
2  
3 config.add_view(  
4     'mypackage.views.my_view',  
5     route_name='ok',  
6     request_method=not_('POST')  
7 )
```

The above example will ensure that the view is called if the request method is *not* POST, at least if no other view is more specific.

This technique of wrapping a predicate value in `not_` can be used anywhere predicate values are accepted:

- `pyramid.config.Configurator.add_view()`
- `pyramid.config.Configurator.add_route()`
- `pyramid.config.Configurator.add_subscriber()`
- `pyramid.view.view_config()`
- `pyramid.events.subscriber()`

New in version 1.5.

pyramid.decorator

reify (*wrapped*)

Use as a class method decorator. It operates almost exactly like the Python `@property` decorator, but it puts the result of the method it decorates into the instance dict after the first call, effectively replacing the function it decorates with an instance variable. It is, in Python parlance, a non-data descriptor. An example:

```
class Foo(object):
    @reify
    def jammy(self):
        print('jammy called')
        return 1
```

And usage of Foo:

```
>>> f = Foo()
>>> v = f.jammy
'jammy called'
>>> print(v)
1
>>> f.jammy
1
>>> # jammy func not called the second time; it replaced itself with 1
```

pyramid.events

48.1 Functions

subscriber (**ifaces*, ***predicates*)

Decorator activated via a *scan* which treats the function being decorated as an event subscriber for the set of interfaces passed as **ifaces* and the set of predicate terms passed as ***predicates* 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. The decorated subscriber will be called for each event type.

```
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')
```

Any `**predicate` arguments will be passed along to `pyramid.config.Configurator.add_subscriber()`. See *Subscriber Predicates* for a description of how predicates can narrow the set of circumstances in which a subscriber will be called.

48.2 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 event 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.



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, this attribute of the request will be `None`.

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.



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, rendering_val=None*)

Subscribers to this event may introspect 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 pyramid.events import subscriber
from pyramid.events import BeforeRender

@subscriber(BeforeRender)
def add_global(event):
    event['mykey'] = 'foo'
```

An object of this type is sent as an event just before a *renderer* is invoked.

If a subscriber adds a key via `__setitem__` that already exists in the renderer globals dictionary, it will overwrite the older value there. This can be problematic because event subscribers to the `BeforeRender` event do not possess any relative ordering. For maximum interoperability with other third-party subscribers, if you write an event subscriber meant to be used as a `BeforeRender` subscriber, your subscriber code will need to ensure no value already exists in the renderer globals dictionary before setting an overriding value (which can be done using `.get` or `__contains__` of the event object).

The dictionary returned from the view is accessible through the `rendering_val` attribute of a `BeforeRender` event.

Suppose you return `{'mykey': 'somevalue', 'mykey2': 'somevalue2'}` from your view callable, like so:

```
from pyramid.view import view_config

@view_config(renderer='some_renderer')
def myview(request):
    return {'mykey': 'somevalue', 'mykey2': 'somevalue2'}
```

`rendering_val` can be used to access these values from the `BeforeRender` object:

```
from pyramid.events import subscriber
from pyramid.events import BeforeRender

@subscriber(BeforeRender)
def read_return(event):
    # {'mykey': 'somevalue'} is returned from the view
    print(event.rendering_val['mykey'])
```

In other words, `rendering_val` is the (non-system) value returned by a view or passed to `render*` as value. This feature is new in Pyramid 1.2.

For a description of the values present in the renderer globals dictionary, see *System Values Used During Rendering*.

See also:

See also `pyramid.interfaces.IBeforeRender`.

update (*E*, ***F*)

Update *D* from dict/iterable *E* and *F*. If *E* has a `.keys()` method, does: for *k* in *E*: *D*[*k*] = *E*[*k*]
If *E* lacks `.keys()` method, does: for (*k*, *v*) in *E*: *D*[*k*] = *v*. In either case, this is followed by:
for *k* in *F*: *D*[*k*] = *F*[*k*].

clear () → None. Remove all items from *D*.

copy () → a shallow copy of *D*

static fromkeys (*S*[, *v*]) → New dict with keys from *S* and values equal to *v*.
v defaults to None.

get (*k*[, *d*]) → *D*[*k*] if *k* in *D*, else *d*. *d* defaults to None.

has_key (*k*) → True if *D* has a key *k*, else False

items () → list of *D*'s (key, value) pairs, as 2-tuples

iteritems () → an iterator over the (key, value) items of *D*

iterkeys () → an iterator over the keys of *D*

itervalues () → an iterator over the values of *D*

keys () → list of *D*'s keys

pop (*k*[, *d*]) → *v*, remove specified key and return the corresponding value.
If key is not found, *d* is returned if given, otherwise `KeyError` is raised

popitem () → (*k*, *v*), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if *D* is empty.

setdefault (*k*[, *d*]) → *D*.get(*k*,*d*), also set *D*[*k*]=*d* if *k* not in *D*

values () → list of *D*'s values

viewitems () → a set-like object providing a view on *D*'s items

viewkeys () → a set-like object providing a view on *D*'s keys

viewvalues () → an object providing a view on *D*'s values

See *Using Events* for more information about how to register code which subscribes to these events.

pyramid.exceptions

class BadCSRFToken (*detail=None, headers=None, comment=None, body_template=None, **kw*)

This exception indicates the request has failed cross-site request forgery token validation.

class PredicateMismatch (*detail=None, headers=None, comment=None, body_template=None, **kw*)

This exception is raised by multiviews when no view matches all given predicates.

This exception subclasses the `HTTPNotFound` exception for a specific reason: if it reaches the main exception handler, it should be treated as `HTTPNotFound` by any exception view registrations. Thus, typically, this exception will not be seen publicly.

However, this exception will be raised if the predicates of all views configured to handle another exception context cannot be successfully matched. For instance, if a view is configured to handle a context of `HTTPForbidden` and the configured with additional predicates, then `PredicateMismatch` will be raised if:

- An original view callable has raised `HTTPForbidden` (thus invoking an exception view); and
- The given request fails to match all predicates for said exception view associated with `HTTPForbidden`.

The same applies to any type of exception being handled by an exception view.

Forbidden

alias of `HTTPForbidden`

NotFound

alias of HTTPNotFound

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 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

50.1 HTTP Exceptions

This module contains Pyramid HTTP exception classes. Each class relates to a single HTTP status code. Each class is a subclass of the `HTTPException`. Each exception class is also a *response* object.

Each exception class has a status code according to **RFC 2068**: codes with 100-300 are not really errors; 400s are client errors, and 500s are server errors.

Exception

HTTPException

HTTPSuccessful

- 200 - HTTPOk
- 201 - HTTPCreated
- 202 - HTTPAccepted
- 203 - HTTPNonAuthoritativeInformation
- 204 - HTTPNoContent
- 205 - HTTPResetContent

- 206 - HTTPPartialContent

HTTPRedirection

- 300 - HTTPMultipleChoices
- 301 - HTTPMovedPermanently
- 302 - HTTPFound
- 303 - HTTPSeeOther
- 304 - HTTPNotModified
- 305 - HTTPUseProxy
- 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
- 422 - HTTPUnprocessableEntity
- 423 - HTTPLocked
- 424 - HTTPFailedDependency

HTTPServerError

- 500 - HTTPInternalServerError
- 501 - HTTPNotImplemented
- 502 - HTTPBadGateway
- 503 - HTTPServiceUnavailable
- 504 - HTTPGatewayTimeout
- 505 - HTTPVersionNotSupported
- 507 - HTTPInsufficientStorage

HTTP exceptions are also *response* objects, thus they accept most of the same parameters that can be passed to a regular `Response`. Each HTTP exception also has the following 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}`

detail a plain-text message customization that is not subject to environment or header substitutions; accessible in the template via `${detail}`

body_template a `String.template`-format content fragment used for environment and header substitution; the default template includes both the explanation and further detail provided in the message.

Each HTTP exception accepts the following parameters, any others will be forwarded to its `Response` superclass:

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

body a string that will override the `body_template` and be used as the body of the response.

Substitution of response headers into template values is always performed. Substitution of WSGI environment values is performed if a `request` is passed to the exception's constructor.

The subclasses of `_HTTPMove` (`HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPFound`, `HTTPSeeOther`, `HTTPUseProxy` and `HTTPTemporaryRedirect`) are redirections that require a `Location` field. Reflecting this, these subclasses have one additional keyword argument: `location`, which indicates the location to which to redirect.

status_map

A mapping of integer status code to HTTP exception class (eg. the integer “401” maps to `pyramid.httpexceptions.HTTPUnauthorized`). All mapped exception classes are children of `pyramid.httpexceptions`,

exception_response (*status_code*, ***kw*)

Creates an HTTP exception based on a status code. Example:

```
raise exception_response(404) # raises an HTTPNotFound exception.
```

The values passed as `kw` are provided to the exception's constructor.

```
class HTTPException (detail=None, headers=None, comment=None, body_template=None,  
                    **kw)
```

```
class HTTPOk (detail=None, headers=None, comment=None, body_template=None, **kw)  
    subclass of HTTPSuccessful
```

Indicates that the request has succeeded.

code: 200, title: OK

```
class HTTPRedirection (detail=None, headers=None, comment=None,  
                       body_template=None, **kw)  
    base class for exceptions with status codes in the 300s (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 exceptions with status codes in the 400s and 500s
```

This is an exception which indicates that an error has occurred, and that any work in progress should not be committed.

```
class HTTPClientError (detail=None, headers=None, comment=None,  
                      body_template=None, **kw)  
    base class for the 400s, 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 500s, where the server is in-error
```

This is an error condition in which the server is presumed to be in-error. Unless specialized, this is a '500 Internal Server Error'.

```
class HTTPCreated (detail=None, headers=None, comment=None, body_template=None,  
                  **kw)  
    subclass of HTTPSuccessful
```

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 HTTPSuccessful

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, com-*
 *ment=None, body_template=None, **kw*)
 subclass of HTTPSuccessful

This indicates that the returned metainformation 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 HTTPSuccessful

This indicates that the server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation.

code: 204, title: No Content

class HTTPResetContent (*detail=None, headers=None, comment=None,*
 *body_template=None, **kw*)
 subclass of HTTPSuccessful

This indicates that 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 HTTPSuccessful

This indicates that the server has fulfilled the partial GET request for the resource.

code: 206, title: Partial Content

```
class HTTPMultipleChoices (location='', detail=None, headers=None, comment=None,  
                             body_template=None, **kw)  
    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 (location='', detail=None, headers=None, comment=None,  
                             body_template=None, **kw)  
    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 (location='', detail=None, headers=None, comment=None,  
                  body_template=None, **kw)  
    subclass of _HTTPMove
```

This indicates that the requested resource resides temporarily under a different URI.

code: 302, title: Found

```
class HTTPSeeOther (location='', detail=None, headers=None, comment=None,  
                     body_template=None, **kw)  
    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 (location='', detail=None, headers=None, comment=None,  
                    body_template=None, **kw)  
    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 (location='', detail=None, headers=None, comment=None,  
                              body_template=None, **kw)  
    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)  
    subclass of HTTPClientError
```

This indicates that the body or headers failed validity checks, preventing the server from being able to continue processing.

code: 400, title: Bad Request

```
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,  
                    result=None, **kw)  
    subclass of HTTPClientError
```

This indicates that the server understood the request, but is refusing to fulfill it.

code: 403, title: Forbidden

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*. A *Forbidden* exception will be the context of a *Forbidden View*.

This exception's constructor treats two arguments specially. The first argument, *detail*, should be a string. The value of this string will be used as the *message* attribute of the exception object. The second special keyword argument, *result* is usually an instance of `pyramid.security.Denied` or `pyramid.security.ACLDenied` each of which indicates a reason for the forbidden error. However, *result* is also permitted to be just a plain boolean *False* object or *None*. The *result* value will be used as the *result* attribute of the exception object. It defaults to *None*.

The *Forbidden View* can use the attributes of a *Forbidden* exception as necessary to provide extended information in an error report shown to a user.

```
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

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 *detail* argument (the first argument), which should be a string. The value of this string will be available as the *message* attribute of this exception, for availability to the *Not Found View*.

```
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, com-  
                                       ment=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 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 HTTPUnprocessableEntity (detail=None, headers=None, comment=None,  
                                body_template=None, **kw)  
    subclass of HTTPClientError
```

This indicates that the server is unable to process the contained instructions. Only for WebDAV.

code: 422, title: Unprocessable Entity

```
class HTTPLocked (detail=None, headers=None, comment=None, body_template=None,  
                   **kw)  
    subclass of HTTPClientError
```

This indicates that the resource is locked. Only for WebDAV

code: 423, title: Locked

```
class HTTPFailedDependency (detail=None, headers=None, comment=None,  
                              body_template=None, **kw)  
    subclass of HTTPClientError
```

This indicates that the method could not be performed because the requested action depended on another action and that action failed. Only for WebDAV.

code: 424, title: Failed Dependency

```
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

```
class HTTPInsufficientStorage (detail=None, headers=None, comment=None,  
                                body_template=None, **kw)  
    subclass of HTTPServerError
```

This indicates that the server does not have enough space to save the resource.

code: 507, title: Insufficient Storage

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.

TranslationStringFactory (*domain*)

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 should be unicode strings. There is no reason to use translation string objects as arguments as all metadata is ignored.

`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. 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})
```

If using the gettext plural support, which is required for languages that have pluralisation rules other than `n != 1`, the `singular` argument must be the `message_id` defined in the translation file. The plural argument is not used in this case.

Example:

```
num = 1
translated = localizer.pluralize('item_plural',
                                '',
                                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*)

Deprecated since version 1.5: Use the `pyramid.request.Request.localizer` attribute directly instead. 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.

get_locale_name (*request*)

Deprecated since version 1.5: Use `pyramid.request.Request.locale_name` directly instead. Return the *locale name* associated with the current request.

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*). If the attribute exists and it is not `None`, its value will be used.
- 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.)

make_localizer (*current_locale_name*, *translation_directories*)

Create a `pyramid.i18n.Localizer` object corresponding to the provided locale name from the translations found in the list of translation directories.

See *Internationalization and Localization* for more information about using Pyramid internationalization and localization services within an application.

pyramid.interfaces

52.1 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

Extends: `pyramid.interfaces.IDict`

Subscribers to this event may introspect and modify the set of *renderer globals* before they are passed to a *renderer*. The event object itself provides a dictionary-like interface for adding and removing *renderer globals*. The keys and values of the dictionary are those globals. For example:

```
from repoze.events import subscriber
from pyramid.interfaces import IBeforeRender

@subscriber(IBeforeRender)
def add_global(event):
    event['mykey'] = 'foo'
```

See also:

See also *Using The Before Render Event*.

rendering_val

The value returned by a view or passed to a `render` method for this rendering. This feature is new in Pyramid 1.2.

52.2 Other Interfaces

interface IAuthenticationPolicy

An object representing a Pyramid authentication policy.

remember (*request*, *principal*, ***kw*)

Return a set of headers suitable for ‘remembering’ the principal named *principal* when set in a response. An individual authentication policy and its consumers can decide on the composition and meaning of ***kw*.

authenticated_userid (*request*)

Return the authenticated userid or `None` if no authenticated userid can be found. This method of the policy should ensure that a record exists in whatever persistent store is used related to the user (the user should not have been deleted); if a record associated with the current id does not exist in a persistent store, it should return `None`.

unauthenticated_userid (*request*)

Return the *unauthenticated* userid. This method performs the same duty as *authenticated_userid* but is permitted to return the userid based only on data present in the request; it needn’t (and shouldn’t) check any persistent store to ensure that the user record related to the request userid exists.

effective_principals (*request*)

Return a sequence representing the effective principals including the userid and any groups belonged to by the current user, including ‘system’ groups such as Everyone and Authenticated.

forget (*request*)

Return a set of headers suitable for ‘forgetting’ the current user on subsequent requests.

interface IAuthorizationPolicy

An object representing a Pyramid authorization policy.

principals_allowed_by_permission (*context*, *permission*)

Return a set of principal identifiers allowed by the *permission* in *context*. This behavior is optional; if you choose to not implement it you should define this method as something which raises a `NotImplementedError`. This method will only be called when the `pyramid.security.principals_allowed_by_permission` API is used.

permits (*context, principals, permission*)

Return True if any of the principals is allowed the permission in the current context, else return False

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 (any exception that inherits from `pyramid.response.Response`, including `pyramid.httpexceptions.HTTPNotFound` and `pyramid.httpexceptions.HTTPForbidden`).

prepare (*environ*)

Prepares the response for being called as a WSGI application

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 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.request.Request.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

Extends: `pyramid.interfaces.IDict`

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.

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 re-added.

created

Integer representing Epoch time when created.

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.

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.

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()`

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.

new

Boolean attribute. If True, the session is new.

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()`

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 IRendererFactory

__call__ (*info*)
Return an object that implements `pyramid.interfaces.IRenderer`.
info is an object that implements `pyramid.interfaces.IRendererInfo`.

interface IRenderer

__call__ (*value, system*)
Call the renderer with the result of the view (*value*) passed in and return a result (a string or unicode object useful as a response body). Values computed by the system are passed by the system in the *system* parameter, which is a dictionary. Keys in the dictionary include: *view* (the view callable that returned the *value*), *renderer_name* (the template name or simple name of the renderer), *context* (the context object passed to the view), and *request* (the request object passed to the view).

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`.

interface `IDict`

__delitem__ (*k*)

Delete an item from the dictionary which is passed to the renderer as the renderer globals dictionary.

setdefault (*k*, *default=None*)

Return the existing value for key `k` in the dictionary. If no value with `k` exists in the dictionary, set the `default` value into the dictionary under the `k` name passed. If a value already existed in the dictionary, return it. If a value did not exist in the dictionary, return the default

__getitem__ (*k*)

Return the value for key `k` from the dictionary or raise a `KeyError` if the key doesn't exist

__contains__ (*k*)

Return `True` if key `k` exists in the dictionary.

keys ()

Return a list of keys from the dictionary

items ()

Return a list of [(`k`,`v`)] pairs from the dictionary

clear()

Clear all values from the dictionary

get (*k*, *default=None*)

Return the value for key *k* from the dictionary, or the default if no such value exists.

__setitem__ (*k*, *value*)

Set a key/value pair into the dictionary

pop (*k*, *default=None*)

Pop the key *k* from the dictionary and return its value. If *k* doesn't exist, and default is provided, return the default. If *k* doesn't exist and default is not provided, raise a `KeyError`.

update (*d*)

Update the dictionary with another dictionary *d*.

__iter__ ()

Return an iterator over the keys of this dictionary

has_key (*k*)

Return `True` if key *k* exists in the dictionary.

values ()

Return a list of values from the dictionary

itervalues ()

Return an iterator of values from the dictionary

iteritems ()

Return an iterator of (*k*,*v*) pairs from the dictionary

popitem ()

Pop the item with key *k* from the dictionary and return it as a two-tuple (*k*, *v*). If *k* doesn't exist, raise a `KeyError`.

iterkeys ()

Return an iterator of keys from the dictionary

interface IMultiDict

Extends: `pyramid.interfaces.IDict`

An ordered dictionary that can have multiple values for each key. A multidict adds the methods `getall`, `getone`, `mixed`, `extend`, `add`, and `dict_of_lists` to the normal dictionary interface. A multidict data structure is used as `request.POST`, `request.GET`, and `request.params` within an Pyramid application.

extend (*other=None, **kwargs*)

Add a set of keys and values, not overwriting any previous values. The *other* structure may be a list of two-tuples or a dictionary. If ***kwargs* is passed, its value *will* overwrite existing values.

getall (*key*)

Return a list of all values matching the key (may be an empty list)

add (*key, value*)

Add the key and value, not overwriting any previous value.

getone (*key*)

Get one value matching the key, raising a `KeyError` if multiple values were found.

dict_of_lists ()

Returns a dictionary where each key is associated with a list of values.

mixed ()

Returns a dictionary where the values are either single values, or a list of values when a key/value appears more than once in this dictionary. This is similar to the kind of dictionary often used to represent the variables in a web request.

interface IResponse

Represents a WSGI response using the WebOb response interface. Some attribute and method documentation of this interface references **RFC 2616**.

This interface is most famously implemented by `pyramid.response.Response` and the HTTP exception classes in `pyramid.httpexceptions`.

content_length

Gets and sets and deletes the Content-Length header. For more information on Content-Length see RFC 2616 section 14.17. Converts using `int`.

status

The status string.

encode_content (*encoding='gzip', lazy=False*)

Encode the content with the given encoding (only `gzip` and `identity` are supported).

cache_expires

Get/set the Cache-Control and Expires headers. This sets the response to expire in the number of seconds passed when set.

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

vary
Gets and sets and deletes the Vary header. For more information on Vary see section 14.44. Converts using list.

retry_after
Gets and sets and deletes the Retry-After header. For more information on Retry-After see RFC 2616 section 14.37. Converts using HTTP date or delta seconds.

www_authenticate
Gets and sets and deletes the WWW-Authenticate header. For more information on WWW-Authenticate see RFC 2616 section 14.47. Converts using 'parse_auth' and 'serialize_auth'.

content_language
Gets and sets and deletes the Content-Language header. Converts using list. For more information about Content-Language see RFC 2616 section 14.12.

etag
Gets and sets and deletes the ETag header. For more information on ETag see RFC 2616 section 14.19. Converts using Entity tag.

content_location
Gets and sets and deletes the Content-Location header. For more information on Content-Location see RFC 2616 section 14.14.

server
Gets and sets and deletes the Server header. For more information on Server see RFC216 section 14.38.

unset_cookie (*key*, *strict*=True)
Unset a cookie with the given name (remove it from the response).

pragma
Gets and sets and deletes the Pragma header. For more information on Pragma see RFC 2616 section 14.32.

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)

headers

The headers in a dictionary-like object

charset

Get/set the charset (in the Content-Type)

unicode_body

Get/set the unicode value of the body (using the charset of the Content-Type)

status_int

The status as an integer

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)

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.

content_range

Gets and sets and deletes the Content-Range header. For more information on Content-Range see section 14.16. Converts using ContentRange object.

content_encoding

Gets and sets and deletes the Content-Encoding header. For more information about Content-Encoding see RFC 2616 section 14.11.

__call__ (*environ*, *start_response*)

WSGI call interface, should call the start_response callback and should return an iterable

content_md5

Gets and sets and deletes the Content-MD5 header. For more information on Content-MD5 see RFC 2616 section 14.14.

content_disposition

Gets and sets and deletes the Content-Disposition header. For more information on Content-Disposition see RFC 2616 section 19.5.1.

cache_control

Get/set/modify the Cache-Control header (RFC 2616 section 14.9)

location

Gets and sets and deletes the Location header. For more information on Location see RFC 2616 section 14.30.

body

The body of the response, as a str. This will read in the entire app_iter if necessary.

expires

Gets and sets and deletes the Expires header. For more information on Expires see RFC 2616 section 14.21. Converts using HTTP date.

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.

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

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.

last_modified

Gets and sets and deletes the Last-Modified header. For more information on Last-Modified see RFC 2616 section 14.29. Converts using HTTP date.

RequestClass

Alias for `pyramid.request.Request`

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.

date

Gets and sets and deletes the Date header. For more information on Date see RFC 2616 section 14.18. Converts using HTTP date.

copy()

Makes a copy of the response and returns the copy.

accept_ranges

Gets and sets and deletes the Accept-Ranges header. For more information on Accept-Ranges see RFC 2616, section 14.5

age

Gets and sets and deletes the Age header. Converts using int. For more information on Age see RFC 2616, section 14.6.

request

Return the request associated with this response if any.

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.

headerlist

The list of response headers.

environ

Get/set the request environ associated with this response, if any.

allow

Gets and sets and deletes the Allow header. Converts using list. For more information on Allow see RFC 2616, Section 14.7.

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.

interface IIntrospectable

An introspectable object used for configuration introspection. In addition to the methods below, objects which implement this interface must also implement all the methods of Python's `collections.MutableMapping` (the “dictionary interface”), and must be hashable.

register (*introspector, action_info*)

Register this IIntrospectable with an introspector. This method is invoked during action execution. Adds the introspectable and its relations to the introspector. `introspector` should be an object implementing IIntrospector. `action_info` should be a object implementing the interface `pyramid.interfaces.IActionInfo` representing the call that registered this introspectable. Pseudocode for an implementation of this method:

```
def register(self, introspector, action_info):
    self.action_info = action_info
    introspector.add(self)
    for methodname, category_name, discriminator in self._relations:
        method = getattr(introspector, methodname)
        method((i.category_name, i.discriminator),
               (category_name, discriminator))
```

discriminator_hash

an integer hash of the discriminator

title

Text title describing this introspectable

relate (*category_name, discriminator*)

Indicate an intent to relate this `IIntrospectable` with another `IIntrospectable` (the one associated with the `category_name` and `discriminator`) during action execution.

type_name

Text type name describing this introspectable

unrelate (*category_name, discriminator*)

Indicate an intent to break the relationship between this `IIntrospectable` with another `IIntrospectable` (the one associated with the `category_name` and `discriminator`) during action execution.

action_info

An `IActionInfo` object representing the caller that invoked the creation of this introspectable (usually a sentinel until updated during `self.register`)

__hash__ ()

Introspectables must be hashable. The typical implementation of an introspectable's `__hash__` is:

```
return hash((self.category_name,) + (self.discriminator,))
```

discriminator

introspectable discriminator (within category) (must be hashable)

order

integer order in which registered with introspector (managed by introspector, usually)

category_name

introspection category name

interface IIntrospector**get** (*category_name*, *discriminator*, *default=None*)

Get the `IIntrospectable` related to the `category_name` and the discriminator (or discriminator hash) `discriminator`. If it does not exist in the introspector, return the value of `default`

relate (**pairs*)

Given any number of (`category_name`, `discriminator`) pairs passed as positional arguments, relate the associated introspectables to each other. The introspectable related to each pair must have already been added via `.add` or `.add_intr`; a `KeyError` will result if this is not true. An error will not be raised if any pair has already been associated with another.

This method is not typically called directly, instead it's called indirectly by `pyramid.interfaces.IIntrospector.register()`

get_category (*category_name*, *default=None*, *sort_key=None*)

Get a sequence of dictionaries in the form `[{'introspectable': IIntrospectable, 'related': [sequence of related IIntrospectables]}, ...]` where each introspectable is part of the category associated with `category_name`.

If the category named `category_name` does not exist in the introspector the value passed as `default` will be returned.

If `sort_key` is `None`, the sequence will be returned in the order the introspectables were added to the introspector. Otherwise, `sort_key` should be a function that accepts an `IIntrospectable` and returns a value from it (ala the `key` function of Python's `sorted` callable).

unrelate (*pairs)

Given any number of (category_name, discriminator) pairs passed as positional arguments, unrelate the associated introspectables from each other. The introspectable related to each pair must have already been added via `.add` or `.add_intr`; a `KeyError` will result if this is not true. An error will not be raised if any pair is not already related to another.

This method is not typically called directly, instead it's called indirectly by `pyramid.interfaces.IIntrospector.register()`

remove (category_name, discriminator)

Remove the `IIntrospectable` related to `category_name` and `discriminator` from the introspector, and fix up any relations that the introspectable participates in. This method will not raise an error if an introspectable related to the category name and discriminator does not exist.

add (intr)

Add the `IIntrospectable` `intr` (use instead of `pyramid.interfaces.IIntrospector.add()` when you have a custom `IIntrospectable`). Replaces any existing introspectable registered using the same category/discriminator.

This method is not typically called directly, instead it's called indirectly by `pyramid.interfaces.IIntrospector.register()`

related (intr)

Return a sequence of `IIntrospectables` related to the `IIntrospectable` `intr`. Return the empty sequence if no relations for exist.

categorized (sort_key=None)

Get a sequence of tuples in the form [(category_name, [{ 'introspectable': `IIntrospectable`, 'related': [sequence of related `IIntrospectables`] }, ...])] representing all known introspectables. If `sort_key` is `None`, each introspectables sequence will be returned in the order the introspectables were added to the introspector. Otherwise, `sort_key` should be a function that accepts an `IIntrospectable` and returns a value from it (ala the `key` function of Python's `sorted` callable).

categories ()

Return a sorted sequence of category names known by this introspector

interface IActionInfo

Class which provides code introspection capability associated with an action. The `ParserInfo` class used by ZCML implements the same interface.

line

Starting line number in file (as an integer) of action-invoking code. This will be `None` if the value could not be determined.

file

Filename of action-invoking code as a string

__str__()

Return a representation of the action information (including source code from file, if possible)

interface IAssetDescriptor

Describes an *asset*.

isdir()

Returns `True` if the asset is a directory, otherwise returns `False`.

exists()

Returns `True` if asset exists, otherwise returns `False`.

stream()

Returns an input stream for reading asset contents. Raises an exception if the asset is a directory or does not exist.

abspath()

Returns an absolute path in the filesystem to the asset.

listdir()

Returns iterable of filenames of directory contents. Raises an exception if asset is not a directory.

absspec()

Returns the absolute asset specification for this asset (e.g. `mypackage:templates/foo.pt`).

interface IResourceURL**physical_path_tuple**

The physical url path of the resource as a tuple. (New in 1.5)

virtual_path_tuple

The virtual url path of the resource as a tuple. (New in 1.5)

virtual_path

The virtual url path of the resource as a string.

physical_path

The physical url path of the resource as a string.

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

bootstrap (*config_uri*, *request=None*, *options=None*)

Load a WSGI application from the PasteDeploy config file specified by `config_uri`. The environment will be configured as if it is currently serving `request`, leaving a natural environment in place to write scripts that can generate URLs and utilize renderers.

This function returns a dictionary with `app`, `root`, `closer`, `request`, and `registry` keys. `app` is the WSGI app loaded (based on the `config_uri`), `root` is the traversal root resource of the Pyramid application, and `closer` is a parameterless callback that may be called when your script is complete (it pops a threadlocal stack).



Most operations within Pyramid expect to be invoked within the context of a WSGI request, thus it's important when loading your application to anchor it when executing scripts and other code that is not normally invoked during active WSGI requests.



For a complex config file containing multiple Pyramid applications, this function will setup the environment under the context of the last-loaded Pyramid application. You may load a specific application yourself by using the lower-level functions `pyramid.paster.get_app()` and `pyramid.scripting.prepare()` in conjunction with `pyramid.config.global_registries`.

`config_uri` – specifies the PasteDeploy config file to use for the interactive shell. The format is `inifile#name`. If the name is left off, `main` will be assumed.

`request` – specified to anchor the script to a given set of WSGI parameters. For example, most people would want to specify the host, scheme and port such that their script will generate URLs in relation to those parameters. A request with default parameters is constructed for you if none is provided. You can mutate the request’s `environ` later to setup a specific host/port/scheme/etc.

`options` Is passed to `get_app` for use as variable assignments like `{‘http_port’: 8080}` and then use `%(http_port)s` in the config file.

See *Writing a Script* for more information about how to use this function.

get_app (*config_uri*, *name=None*, *options=None*)

Return the WSGI application named `name` in the PasteDeploy config file specified by `config_uri`.

`options`, if passed, should be a dictionary used as variable assignments like `{‘http_port’: 8080}`. This is useful if e.g. `%(http_port)s` is used in the config file.

If the `name` is `None`, this will attempt to parse the name from the `config_uri` string expecting the format `inifile#name`. If no name is found, the name will default to “main”.

get_appsettings (*config_uri*, *name=None*, *options=None*)

Return a dictionary representing the key/value pairs in an `app` section within the file represented by `config_uri`.

`options`, if passed, should be a dictionary used as variable assignments like `{‘http_port’: 8080}`. This is useful if e.g. `%(http_port)s` is used in the config file.

If the `name` is `None`, this will attempt to parse the name from the `config_uri` string expecting the format `inifile#name`. If no name is found, the name will default to “main”.

setup_logging (*config_uri*)

Set up logging via the logging module’s `fileConfig` function with the filename specified via `config_uri` (a string in the form `filename#sectionname`).

ConfigParser defaults are specified for the special `__file__` and `here` variables, similar to PasteDeploy config loading.

pyramid.path

CALLER_PACKAGE

A constant used by the constructor of `pyramid.path.DottedNameResolver` and `pyramid.path.AssetResolver`.

class DottedNameResolver (*package=pyramid.path.CALLER_PACKAGE*)

A class used to resolve a *dotted Python name* to a package or module object.

New in version 1.3.

The constructor accepts a single argument named `package` which may be any of:

- A fully qualified (not relative) dotted name to a module or package
- a Python module or package object
- The value `None`
- The constant value `pyramid.path.CALLER_PACKAGE`.

The default value is `pyramid.path.CALLER_PACKAGE`.

The `package` is used when a relative dotted name is supplied to the `resolve()` method. A dotted name which has a `.` (dot) or `:` (colon) as its first character is treated as relative.

If `package` is `None`, the resolver will only be able to resolve fully qualified (not relative) names. Any attempt to resolve a relative name will result in an `ValueError` exception.

If `package` is `pyramid.path.CALLER_PACKAGE`, the resolver will treat relative dotted names as relative to the caller of the `resolve()` method.

If `package` is a *module* or *module name* (as opposed to a package or package name), its containing package is computed and this package used to derive the package name (all names are resolved relative to packages, never to modules). For example, if the `package` argument to this type was passed the string `xml.dom.expatbuilder`, and `.minidom` is supplied to the `resolve()` method, the resulting import would be for `xml.minidom`, because `xml.dom.expatbuilder` is a module object, not a package object.

If `package` is a *package* or *package name* (as opposed to a module or module name), this package will be used to relative compute dotted names. For example, if the `package` argument to this type was passed the string `xml.dom`, and `.minidom` is supplied to the `resolve()` method, the resulting import would be for `xml.minidom`.

maybe_resolve (*dotted*)

This method behaves just like `resolve()`, except if the dotted value passed is not a string, it is simply returned. For example:

```
import xml
r = DottedNameResolver()
v = r.maybe_resolve(xml)
# v is the xml module; no exception raised
```

resolve (*dotted*)

This method resolves a dotted name reference to a global Python object (an object which can be imported) to the object itself.

Two dotted name styles are supported:

- `pkg_resources`-style dotted names where non-module attributes of a package are separated from the rest of the path using a `:` e.g. `package.module:attr`.
- `zope.dottedname`-style dotted names where non-module attributes of a package are separated from the rest of the path using a `.` e.g. `package.module.attr`.

These styles can be used interchangeably. If the supplied name contains a `:` (colon), the `pkg_resources` resolution mechanism will be chosen, otherwise the `zope.dottedname` resolution mechanism will be chosen.

If the dotted argument passed to this method is not a string, a `ValueError` will be raised.

When a dotted name cannot be resolved, a `ValueError` error is raised.

Example:

```
r = DottedNameResolver()
v = r.resolve('xml') # v is the xml module
```

class AssetResolver (*package=pyramid.path.CALLER_PACKAGE*)

A class used to resolve an *asset specification* to an *asset descriptor*.

New in version 1.3.

The constructor accepts a single argument named `package` which may be any of:

- A fully qualified (not relative) dotted name to a module or package
- a Python module or package object
- The value `None`
- The constant value `pyramid.path.CALLER_PACKAGE`.

The default value is `pyramid.path.CALLER_PACKAGE`.

The `package` is used when a relative asset specification is supplied to the `resolve()` method. An asset specification without a colon in it is treated as relative.

If `package` is `None`, the resolver will only be able to resolve fully qualified (not relative) asset specifications. Any attempt to resolve a relative asset specification will result in a `ValueError` exception.

If `package` is `pyramid.path.CALLER_PACKAGE`, the resolver will treat relative asset specifications as relative to the caller of the `resolve()` method.

If `package` is a *module* or *module name* (as opposed to a package or package name), its containing package is computed and this package is used to derive the package name (all names are resolved relative to packages, never to modules). For example, if the `package` argument to this type was passed the string `xml.dom.expatbuilder`, and `template.pt` is supplied to the `resolve()` method, the resulting absolute asset spec would be `xml.minidom:template.pt`, because `xml.dom.expatbuilder` is a module object, not a package object.

If `package` is a *package* or *package name* (as opposed to a module or module name), this package will be used to compute relative asset specifications. For example, if the `package` argument to this type was passed the string `xml.dom`, and `template.pt` is supplied to the `resolve()` method, the resulting absolute asset spec would be `xml.minidom:template.pt`.

resolve (*spec*)

Resolve the asset spec named as *spec* to an object that has the attributes and methods described in `pyramid.interfaces.IAssetDescriptor`.

If *spec* is an absolute filename (e.g. `/path/to/myproject/templates/foo.pt`) or an absolute asset spec (e.g. `myproject:templates.foo.pt`), an asset descriptor is returned without taking into account the package passed to this class' constructor.

If *spec* is a *relative* asset specification (an asset specification without a `:` in it, e.g. `templates/foo.pt`), the package argument of the constructor is used as the package portion of the asset spec. For example:

```
a = AssetResolver('myproject')
resolver = a.resolve('templates/foo.pt')
print(resolver.abspath())
# -> /path/to/myproject/templates/foo.pt
```

If the `AssetResolver` is constructed without a package argument of `None`, and a relative asset specification is passed to `resolve`, an `ValueError` exception is raised.

pyramid.registry

class Registry (*name='', bases=()*)

A registry object is an *application registry*. It is used by the framework itself to perform mappings of URLs to view callables, as well as servicing other various framework duties. A registry has its own internal API, but this API is rarely used by Pyramid application developers (it's usually only used by developers of the Pyramid framework). But it has a number of attributes that may be useful to application developers within application code, such as `settings`, which is a dictionary containing application deployment settings.

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.

introspector

New in version 1.3.

When a registry is set up (or created) by a *Configurator*, the registry will be decorated with an instance named `introspector` implementing the `pyramid.interfaces.IIntrospector` interface.

See also:

See also `pyramid.config.Configurator.introspector`.

When a registry is created “by hand”, however, this attribute will not exist until set up by a configurator.

This attribute is often accessed as `request.registry.introspector` in a typical Pyramid application.

notify (*events)

Fire one or more events. All event subscribers to the event(s) will be notified. The subscribers will be called synchronously. This method is often accessed as `request.registry.notify` in Pyramid applications to fire custom events. See *Creating Your Own Events* for more information.

class Introspectable

New in version 1.3.

The default implementation of the interface `pyramid.interfaces.IIntrospectable` used by framework exenders. An instance of this class is created when `pyramid.config.Configurator.introspectable` is called.

class Deferred (func)

Can be used by a third-party configuration extender to wrap a *discriminator* during configuration if an immediately hashable discriminator cannot be computed because it relies on unresolved values. The function should accept no arguments and should return a hashable discriminator.

New in version 1.4.

undefer (v)

Function which accepts an object and returns it unless it is a `pyramid.registry.Deferred` instance. If it is an instance of that class, its `resolve` method is called, and the result of the method is returned.

New in version 1.4.

class predvalseq

A subtype of tuple used to represent a sequence of predicate values

New in version 1.4.

pyramid.renderers

get_renderer (*renderer_name*, *package=None*)

Return the renderer object for the renderer *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 *package* with the relative asset specification *renderer_name*. If *package* is *None* (the default), 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 *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 an *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 *package* with the relative asset specification *renderer_name*. If *package* is *None* (the default), 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*, *renderer_name*, and *view*. See *System Values Used During*

Rendering for the full list. 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 `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 `package` with the relative asset specification `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`, `renderer_name`, and `view`. See *System Values Used During Rendering* for the full list. 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). Keep in mind that if the `request` parameter is not passed in, any changes to `request.response` attributes made before calling this function will be ignored.

class JSON (*serializer=<function dumps at 0x7fd58b5d0938>*, *adapters=()*, ***kw*)

Renderer that returns a JSON-encoded string.

Configure a custom JSON renderer using the `add_renderer()` API at application startup time:

```
from pyramid.config import Configurator

config = Configurator()
config.add_renderer('myjson', JSON(indent=4))
```

Once this renderer is registered as above, you can use `myjson` as the `renderer=` parameter to `@view_config` or `add_view()`:

```

from pyramid.view import view_config

@view_config(renderer='myjson')
def myview(request):
    return {'greeting': 'Hello world'}

```

Custom objects can be serialized using the renderer by either implementing the `__json__` magic method, or by registering adapters with the renderer. See *Serializing Custom Objects* for more information.



The default serializer uses `json.JSONEncoder`. A different serializer can be specified via the `serializer` argument. Custom serializers should accept the object, a callback default, and any extra `kw` keyword arguments passed during renderer construction. This feature isn't widely used but it can be used to replace the stock JSON serializer with, say, `simplejson`. If all you want to do, however, is serialize custom objects, you should use the method explained in *Serializing Custom Objects* instead of replacing the serializer.

New in version 1.4: Prior to this version, there was no public API for supplying options to the underlying serializer without defining a custom renderer.

add_adapter (*type_or_iface*, *adapter*)

When an object of the type (or interface) `type_or_iface` fails to automatically encode using the serializer, the renderer will use the adapter `adapter` to convert it into a JSON-serializable object. The adapter must accept two arguments: the object and the currently active request.

```

class Foo(object):
    x = 5

def foo_adapter(obj, request):
    return obj.x

renderer = JSON(indent=4)
renderer.add_adapter(Foo, foo_adapter)

```

When you've done this, the JSON renderer will be able to serialize instances of the `Foo` class when they're encountered in your view results.

class JSONP (*param_name*='callback', ***kw*)

JSONP renderer factory helper which implements a hybrid json/jsonp renderer. JSONP is useful for making cross-domain AJAX requests.

Configure a JSONP renderer using the `pyramid.config.Configurator.add_renderer()` API at application startup time:

```
from pyramid.config import Configurator

config = Configurator()
config.add_renderer('jsonp', JSONP(param_name='callback'))
```

The class' constructor also accepts arbitrary keyword arguments. All keyword arguments except `param_name` are passed to the `json.dumps` function as its keyword arguments.

```
from pyramid.config import Configurator

config = Configurator()
config.add_renderer('jsonp', JSONP(param_name='callback', indent=4))
```

Changed in version 1.4: The ability of this class to accept a `**kw` in its constructor.

The arguments passed to this class' constructor mean the same thing as the arguments passed to `pyramid.renderers.JSON` (including serializer and adapters).

Once this renderer is registered via `add_renderer()` as above, you can use `jsonp` as the `renderer=` parameter to `@view_config` or `pyramid.config.Configurator.add_view'()`:

```
from pyramid.view import view_config

@view_config(renderer='jsonp')
def myview(request):
    return {'greeting': 'Hello world'}
```

When a view is called that uses the JSONP renderer:

- If there is a parameter in the request's HTTP query string that matches the `param_name` of the registered JSONP renderer (by default, `callback`), the renderer will return a JSONP response.
- If there is no callback parameter in the request's query string, the renderer will return a 'plain' JSON response.

New in version 1.1.

See also:

See also *JSONP Renderer*.

add_adapter (*type_or_iface*, *adapter*)

When an object of the type (or interface) `type_or_iface` fails to automatically encode using the serializer, the renderer will use the adapter `adapter` to convert it into a JSON-serializable object. The adapter must accept two arguments: the object and the currently active request.

```
class Foo(object):
    x = 5

def foo_adapter(obj, request):
    return obj.x

renderer = JSON(indent=4)
renderer.add_adapter(Foo, foo_adapter)
```

When you've done this, the JSON renderer will be able to serialize instances of the `Foo` class when they're encountered in your view results.

null_renderer

An object that can be used in advanced integration cases as input to the view configuration `renderer=` argument. When the null renderer is used as a view renderer argument, Pyramid avoids converting the view callable result into a `Response` object. This is useful if you want to reuse the view configuration and lookup machinery outside the context of its use by the Pyramid router.

pyramid.request

class Request (*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*,
 ***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://webob.org/> 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.

exc_info

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, result of `sys.exc_info()` will be available as the `exc_info` attribute of the request within a *exception view*, a *response callback* or a *finished callback*. If no exception occurred, the value of `request.exc_info` will be `None` within response and finished callbacks.

response

This attribute is actually a “reified” property which returns an instance of the `pyramid.response.Response` class. The response object returned does not exist until this attribute is accessed. Once it is accessed, subsequent accesses to this request object will return the same Response object.

The `request.response` API can be used by renderers. A renderer obtains the response object it will return from a view that uses that renderer by accessing `request.response`. Therefore, it’s possible to use the `request.response` API to set up a response object with “the right” attributes (e.g. by calling `request.response.set_cookie(...)` or `request.response.content_type = 'text/plain'`, etc) within a view that uses a renderer. For example, within a view that uses a *renderer*:

```
response = request.response
response.set_cookie('mycookie', 'mine, all mine!')
return {'text': 'Value that will be used by the renderer'}
```

Mutations to this response object will be preserved in the response sent to the client after rendering. For more information about using `request.response` in conjunction with a renderer, see *Varying Attributes of Rendered Responses*.

Non-renderer code can also make use of `request.response` instead of creating a response “by hand”. For example, in view code:

```
response = request.response
response.body = 'Hello!'
response.content_type = 'text/plain'
return response
```

Note that the response in this circumstance is not “global”; it still must be returned from the view code if a renderer is not used.

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.

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*.

authenticated_userid

New in version 1.5.

A property which returns the userid of the currently authenticated user or `None` if there is no *authentication policy* in effect or there is no currently authenticated user. This differs from `unauthenticated_userid`, because the effective authentication policy will have ensured that a record associated with the userid exists in persistent storage; if it has not, this value will be `None`.

unauthenticated_userid

New in version 1.5.

A property which returns a value 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. Even if the userid does not exist in persistent storage, this value will be the value of the userid *claimed* by the request data.

effective_principals

New in version 1.5.

A property which returns the list of ‘effective’ *principal* identifiers for this 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 a sequence containing only the `pyramid.security.Everyone` principal.

invoke_subrequest (*request*, *use_tweens=False*)

New in version 1.4a1.

Obtain a response object from the Pyramid application based on information in the `request` object provided. The `request` object must be an object that implements the Pyramid request interface (such as a `pyramid.request.Request` instance). If `use_tweens` is `True`, the request will be sent to the *tween* in the tween stack closest to the request ingress. If `use_tweens` is `False`, the request will be sent to the main router handler, and no tweens will be invoked.

This function also:

-
- manages the threadlocal stack (so that `get_current_request()` and `get_current_registry()` work during a request)
 - Adds a registry attribute (the current Pyramid registry) and a `invoke_subrequest` attribute (a callable) to the request object it's handed.
 - sets request extensions (such as those added via `add_request_method()` or `set_request_property()`) on the request it's passed.
 - causes a `NewRequest` event to be sent at the beginning of request processing.
 - causes a `ContextFound` event to be sent when a context resource is found.
 - Ensures that the user implied by the request passed has the necessary authorization to invoke view callable before calling it.
 - Calls any *response callback* functions defined within the request's lifetime if a response is obtained from the Pyramid application.
 - causes a `NewResponse` event to be sent if a response is obtained.
 - Calls any *finished callback* functions defined within the request's lifetime.

`invoke_subrequest` isn't *actually* a method of the Request object; it's a callable added when the Pyramid router is invoked, or when a subrequest is invoked. This means that it's not available for use on a request provided by e.g. the `pshell` environment.

See also:

See also *Invoking a Subrequest*.

has_permission (*permission*, *context=None*)

Given a permission and an optional context, returns an instance of `pyramid.security.Allowed` if the permission is granted to this request with the provided context, or the context already associated with the request. Otherwise, returns an instance of `pyramid.security.Denied`. This method delegates to the current authentication and authorization policies. Returns `pyramid.security.Allowed` unconditionally if no authentication policy has been registered for this request. If `context` is not supplied or is supplied as `None`, the context used is the `request.context` attribute.

Parameters

- **permission** (*unicode*, *str*) – Does this request have the given permission?

- **context** (*object*) – A resource object or None

Returns `pyramid.security.PermitsResult`

New in version 1.5.

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 `tweens` and *before* 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:

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:

See also *Using Finished Callbacks*.

route_url (*route_name*, **elements*, ***kw*)

Generates a fully qualified URL for a named Pyramid *route configuration*.

Use the route’s `name` as the first positional argument. Additional positional arguments (**elements*) 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`:

```
request.route_url('foobar',
                  foo='1')           => <KeyError exception>
request.route_url('foobar',
                  foo='1',
                  bar='2')           => <KeyError exception>
request.route_url('foobar',
                  foo='1',
                  bar='2',
                  traverse=('a','b')) => http://e.com/1/2/a/b
request.route_url('foobar',
                  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 after being URL-quoted-except-for-embedded-slashes.

If no `_query` keyword argument is provided, the request query string will be returned in the URL. If it is present, it will be used to compose a query string that will be tacked on to the end of the URL, replacing any request query string. The value of `_query` may 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. This will produce a query string in the `x-www-form-urlencoded` format. A `non-x-www-form-urlencoded` query string may be used by passing a *string* value as `_query` in which case it will be URL-quoted (e.g. `query="foo bar"` will become `"foo%20bar"`). However, the result will not need to be in `k=v` form as required by `x-www-form-urlencoded`. 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.

Changed in version 1.5: Allow the `_query` option to be a string to enable alternative encodings.

If a keyword argument `_anchor` is present, its string representation will be quoted per **RFC 3986** and 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`).



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.

Changed in version 1.5: The `_anchor` option will be escaped instead of using its raw string representation.

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 any of the keyword arguments `_scheme`, `_host`, or `_port` is passed and is non-None, the provided value will replace the named portion in the generated URL. For example, if you pass `_host='foo.com'`, and the URL that would have been generated without the host replacement is `http://example.com/a`, the result will be `http://foo.com/a`.

Note that if `_scheme` is passed as `https`, and `_port` is not passed, the `_port` value is assumed to have been passed as 443. Likewise, if `_scheme` is passed as `http` and `_port` is not passed, the `_port` value is assumed to have been passed as 80. To avoid this behavior, always explicitly pass `_port` whenever you pass `_scheme`.

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).

If both `_app_url` and any of `_scheme`, `_host`, or `_port` are passed, `_app_url` takes precedence and any values passed for `_scheme`, `_host`, and `_port` will be ignored.

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 passed to this function might be augmented or changed.

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 function accepts the same argument 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 `script_name`, path, query parameters, and anchor data are present in the returned string.

For example, if you’ve defined a route named ‘foobar’ with the path `/ {foo} / {bar}`, this call to `route_path`:

```
request.route_path('foobar', foo='1', bar='2')
```

Will return the string `/1/2`.



Calling `request.route_path('route')` is the same as calling `request.route_url('route', _app_url=request.script_name)`. `pyramid.request.Request.route_path()` is, in fact, implemented in terms of `pyramid.request.Request.route_url()` in just this way. As a result, any `_app_url` passed within the `**kw` values to `route_path` will be ignored.

current_route_url (**elements*, ***kw*)

Generates a fully qualified URL for a named Pyramid *route configuration* based on the ‘current route’.

This function supplements `pyramid.request.Request.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 method have the same meaning as those with the same names passed to `pyramid.request.Request.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` key-word 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 `request.route_url(route_name, *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 `request.current_route_url()` in this situation will be `/foo/1`.

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 `request.current_route_url(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, `request.current_route_url(_route_name=‘fooaction’, page=‘5’)` Will return string like: `/foo/view/5`.

current_route_path(*elements, **kw)


Generates a path (aka a ‘relative URL’, a URL minus the host, scheme, and port) for the Pyramid *route configuration* matched by the current request.

This function accepts the same argument as `pyramid.request.Request.current_route_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the script_name, path, query parameters, and anchor data are present in the returned string.

For example, if the route matched by the current request has the pattern `/foo/{bar}`, this call to `current_route_path`:

```
request.current_route_path(foo=‘1’, bar=‘2’)
```

Will return the string `/1/2`.

 Calling `request.current_route_path(‘route’)` is the same as calling `request.current_route_url(‘route’, _app_url=request.script_name).pyramid.request.Request.current_route_url()` is, in fact, implemented in terms of `pyramid.request.Request.current_route_url()` in just this way. As a result, any `_app_url` passed within the `**kw` values to `current_route_path` will be ignored.

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* (see *Serving Static Assets*).

Example:

```
request.static_url('mypackage:static/foo.css') =>
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 an absolute path (e.g. `/abspath/to/static/foo.css`) or a *asset specification* (e.g. `mypackage:static/foo.css`).

The purpose of the ***kw* argument is the same as the purpose of the `pyramid.request.Request.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.

static_path (*path*, ***kw*)

Generates a path (aka a 'relative URL', a URL minus the host, scheme, and port) for a static resource.

This function accepts the same argument as `pyramid.request.Request.static_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the *script_name*, *path*, *query parameters*, and *anchor data* are present in the returned string.

Example:

```
request.static_path('mypackage:static/foo.css') =>
/static/foo.css
```



Calling `request.static_path(apath)` is the same as calling `request.static_url(apath, _app_url=request.script_name)`. `pyramid.request.Request.static_path()` is, in fact, implemented in terms of `:meth:'pyramid.request.Request.static_url'` in just this way. As a result, any `_app_url` passed within the `**kw` values to `static_path` will be ignored.

resource_url (*resource*, **elements*, ***kw*)

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 method is always a UTF-8 encoded string.

Examples:

```
request.resource_url(resource) =>
    http://example.com/

request.resource_url(resource, 'a.html') =>
    http://example.com/a.html

request.resource_url(resource, 'a.html', query={'q':'1'}) =>
    http://example.com/a.html?q=1

request.resource_url(resource, 'a.html', anchor='abc') =>
    http://example.com/a.html#abc

request.resource_url(resource, app_url='') =>
    /
```

Any positional arguments passed in as *elements* must be strings Unicode objects, or integer 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 any element is an integer, it will be converted to its string representation 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 a 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` may 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 `:func:pyramid.url.urlencode` function. This will produce a query string in the `x-www-form-urlencoded` encoding. A non-`x-www-form-urlencoded` query string may be used by passing a *string* value as `query` in which case it will be URL-quoted (e.g. `query="foo bar"` will become `"foo%20bar"`). However, the result will not need to be in `k=v` form as required by `x-www-form-urlencoded`. 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.

Changed in version 1.5: Allow the `query` option to be a string to enable alternative encodings.

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.

Changed in version 1.5: The `anchor` option will be escaped instead of using its raw string representation.

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 any of the keyword arguments `scheme`, `host`, or `port` is passed and is non-`None`, the provided value will replace the named portion in the generated URL. For example, if you pass `host='foo.com'`, and the URL that would have been generated without the host replacement is `http://example.com/a`, the result will be `http://foo.com/a`.

If `scheme` is passed as `https`, and an explicit `port` is not passed, the `port` value is assumed to have been passed as 443. Likewise, if `scheme` is passed as `http` and `port` is not passed, the `port` value is assumed to have been passed as 80. To avoid this behavior, always explicitly pass `port` whenever you pass `scheme`.

If a keyword argument `app_url` is passed and is not `None`, it should be a string that will be used as the `port/hostname/initial path` portion of the generated URL instead of the default request application URL. For example, if `app_url='http://foo'`, then the resulting url of a resource that has a path of `/baz/bar` will be `http://foo/baz/bar`. If you want to generate completely relative URLs with no leading scheme, host, port, or initial path, you can pass `app_url=''`. Passing `app_url=''` when the resource path is `/baz/bar` will return `/baz/bar`.

New in version 1.3: `app_url`

If `app_url` is passed and any of `scheme`, `port`, or `host` are also passed, `app_url` will take precedence and the values passed for `scheme`, `host`, and/or `port` will be ignored.

If the `resource` passed in has a `__resource_url__` method, it will be used to generate the URL (`scheme`, `host`, `port`, `path`) for the base resource which is operated upon by this function.

See also:

See also *Overriding Resource URL Generation*.

New in version 1.5: `route_name`, `route_kw`, and `route_remainder_name`

If `route_name` is passed, this function will delegate its URL production to the `route_url` function. Calling `resource_url(someresource, 'element1', 'element2', query={'a':1}, route_name='blogentry')` is roughly equivalent to doing:

```
remainder_path = request.resource_path(someobject)
url = request.route_url(
    'blogentry',
    'element1',
    'element2',
    _query={'a':1},
    traverse=traversal_path,
)
```

It is only sensible to pass `route_name` if the route being named has a `*remainder` stararg value such as `*traverse`. The remainder value will be ignored in the output otherwise.

By default, the resource path value will be passed as the name `traverse` when `route_url` is called. You can influence this by passing a different `route_remainder_name` value if the route has a different `*stararg` value at its end. For example if the route pattern you want to replace has a `*subpath` stararg ala `/foo*subpath`:

```
request.resource_url(  
    resource,  
    route_name='myroute',  
    route_remainder_name='subpath'  
)
```

If `route_name` is passed, it is also permissible to pass `route_kw`, which will be passed as additional keyword arguments to `route_url`. Saying `resource_url(someresource, 'element1', 'element2', route_name='blogentry', route_kw={'id': '4'}, _query={'a': '1'})` is roughly equivalent to:

```
remainder_path = request.resource_path_tuple(someobject)  
kw = {'id': '4', '_query': {'a': '1'}, 'traverse': traversal_path}  
url = request.route_url(  
    'blogentry',  
    'element1',  
    'element2',  
    **kw,  
)
```

If `route_kw` or `route_remainder_name` is passed, but `route_name` is not passed, both `route_kw` and `route_remainder_name` will be ignored. If `route_name` is passed, the `__resource_url__` method of the resource passed is ignored unconditionally. This feature is incompatible with resources which generate their own URLs.



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.



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).



For backwards compatibility purposes, this method is also aliased as the `model_url` method of `request`.

resource_path (*resource*, **elements*, ***kw*)

Generates a path (aka a ‘relative URL’, a URL minus the host, scheme, and port) for a *resource*.

This function accepts the same argument as `pyramid.request.Request.resource_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the `script_name`, `path`, query parameters, and anchor data are present in the returned string.



Calling `request.resource_path(resource)` is the same as calling `request.resource_path(resource, app_url=request.script_name).pyramid.request.Request.resource_path()` is, in fact, implemented in terms of `pyramid.request.Request.resource_url()` in just this way. As a result, any `app_url` passed within the `**kw` values to `route_path` will be ignored. `scheme`, `host`, and `port` are also ignored.

json_body

This property will return the JSON-decoded variant of the request body. If the request body is not well-formed JSON, or there is no body associated with this request, this property will raise an exception.

See also:

See also *Dealing With A JSON-Encoded Request Body*.

set_property (*callable*, *name=None*, *reify=False*)

Add a callable or a property descriptor to the request instance.

Properties, unlike attributes, are lazily evaluated by executing an underlying callable when accessed. They can be useful for adding features to an object without any cost if those features go unused.

A property may also be reified via the `pyramid.decorator.reify` decorator by setting `reify=True`, allowing the result of the evaluation to be cached. Thus the value of the property is only computed once for the lifetime of the object.

`callable` can either be a callable that accepts the request as its single positional parameter, or it can be a property descriptor.

If the callable is a property descriptor a `ValueError` will be raised if `name` is `None` or `reify` is `True`.

If `name` is `None`, the name of the property will be computed from the name of the callable.

```
1 def _connect(request):
2     conn = request.registry.dbsession()
3     def cleanup(request):
4         # since version 1.5, request.exception is no
5         # longer eagerly cleared
6         if request.exception is not None:
7             conn.rollback()
8         else:
9             conn.commit()
10        conn.close()
11    request.add_finished_callback(cleanup)
12    return conn
13
14 @subscriber(NewRequest)
15 def new_request(event):
16     request = event.request
17     request.set_property(_connect, 'db', reify=True)
```

The subscriber doesn't actually connect to the database, it just provides the API which, when accessed via `request.db`, will create the connection. Thanks to `reify`, only one connection is made per-request even if `request.db` is accessed many times.

This pattern provides a way to augment the `request` object without having to subclass it, which can be useful for extension authors.

New in version 1.3.

localizer

A *localizer* which will use the current locale name to translate values.

New in version 1.5.

locale_name

The locale name of the current request as computed by the *locale negotiator*.

New in version 1.5.

GET

Return a `MultiDict` containing all the variables from the `QUERY_STRING`.

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 & PATCH requests with an appropriate Content-Type are also supported.

accept

Gets and sets the Accept header (HTTP spec section 14.1).

accept_charset

Gets and sets the Accept-Charset header (HTTP spec section 14.2).

accept_encoding

Gets and sets the Accept-Encoding header (HTTP spec section 14.3).

accept_language

Gets and sets the Accept-Language header (HTTP spec section 14.4).

application_url

The URL including SCRIPT_NAME (no PATH_INFO or query string)

as_bytes (*skip_body=False*)

Return HTTP bytes representing this request. If skip_body is True, exclude the body. If skip_body is an integer larger than one, skip body only if its length is bigger than that number.

authorization

Gets and sets the Authorization header (HTTP spec section 14.8). Converts it using parse_auth and serialize_auth.

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__.

body

Return the content of the request body.

body_file

Input stream of the request (`wsgi.input`). Setting this property resets the `content_length` and seekable flag (unlike setting `req.body_file_raw`).

body_file_raw

Gets and sets the `wsgi.input` key in the environment.

body_file_seekable

Get the body of the request (`wsgi.input`) as a seekable file-like object. Middleware and routing applications should use this attribute over `.body_file`.

If you access this value, `CONTENT_LENGTH` will also be updated.

cache_control

Get/set/modify the Cache-Control header (HTTP spec 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.

client_addr

The effective client IP address as a string. If the `HTTP_X_FORWARDED_FOR` header exists in the WSGI environ, this attribute returns the client IP address present in that header (e.g. if the header value is `192.168.1.1, 192.168.1.2`, the value will be `192.168.1.1`). If no `HTTP_X_FORWARDED_FOR` header is present in the environ at all, this attribute will return the value of the `REMOTE_ADDR` header. If the `REMOTE_ADDR` header is unset, this attribute will return the value None.



It is possible for user agents to put someone else's IP or just any string in `HTTP_X_FORWARDED_FOR` as it is a normal HTTP header. Forward proxies can also provide incorrect values (private IP addresses etc). You cannot "blindly" trust the result of this method to provide you with valid data unless you're certain that `HTTP_X_FORWARDED_FOR` has the correct values. The WSGI server must be behind a trusted proxy for this to be true.

content_length

Gets and sets the `Content-Length` header (HTTP spec section 14.13). Converts it using `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

Return a dictionary of cookies as found in the request.

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 `BytesIO` 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 `Date` header (HTTP spec section 14.8). Converts it using HTTP date.

domain

Returns the domain portion of the host value. Equivalent to:

```
domain = request.host
if ':' in domain:
    domain = domain.split(':', 1)[0]
```

This will be equivalent to the domain portion of the `HTTP_HOST` value in the environment if it exists, or the `SERVER_NAME` value in the environment if it doesn't. For example, if the environment contains an `HTTP_HOST` value of `foo.example.com:8000`, `request.domain` will return `foo.example.com`.

Note that this value cannot be *set* on the request. To set the host value use `webob.request.Request.host()` instead.

classmethod from_bytes (*b*)

Create a request from HTTP bytes data. If the bytes contain extra data after the request, raise a `ValueError`.

classmethod from_file (*fp*)

Read 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 (unless the request is a POST or PUT and has no Content-Length, in that case, the entire file is read).

This reads the request as represented by `str(req)`; it may not read every valid HTTP request properly.

get_response (*application=None, 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.

If `application` is not given, this will send the request to `self.make_default_send_app()`

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_port

The effective server port number as a string. If the `HTTP_HOST` header exists in the WSGI environ, this attribute returns the port number present in that header. If the `HTTP_HOST` header exists but contains no explicit port number: if the WSGI url scheme is "https", this attribute returns "443", if the WSGI url scheme is "http", this attribute returns "80". If no `HTTP_HOST` header is present in the environ at all, this attribute will return the value of the `SERVER_PORT` header (which is guaranteed to be present).

host_url

The URL through the host (no path)

http_version

Gets and sets the `SERVER_PROTOCOL` key in the environment.

if_match

Gets and sets the `If-Match` header (HTTP spec section 14.24). Converts it as a `Etag`.

if_modified_since

Gets and sets the `If-Modified-Since` header (HTTP spec section 14.25). Converts it using HTTP date.

if_none_match

Gets and sets the `If-None-Match` header (HTTP spec section 14.26). Converts it as a `Etag`.

if_range

Gets and sets the `If-Range` header (HTTP spec section 14.27). Converts it using `IfRange` object.

if_unmodified_since

Gets and sets the `If-Unmodified-Since` header (HTTP spec section 14.28). Converts it using HTTP date.

is_body_readable

`webob.is_body_readable` is a flag that tells us that we can read the input stream even though `CONTENT_LENGTH` is missing. This allows `FakeCGIBody` to work and can be used by servers to support chunked encoding in requests. For background see <https://bitbucket.org/ianb/webob/issue/6>

is_body_seekable

Gets and sets the `webob.is_body_seekable` key in the environment.

is_response (*ob*)

Return `True` if the object passed as *ob* is a valid response object, `False` otherwise.

is_xhr

Is `X-Requested-With` header present and equal to `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.

json

Access the body of the request as JSON

localizer

Convenience property to return a localizer

make_body_seekable()

This forces `environ['wsgi.input']` to be seekable. That means that, the content is copied into a BytesIO or temporary file and flagged as seekable, so that it will not be unnecessarily copied again.

After calling this method the `.body_file` is always seeked to the start of file and `.content_length` is not None.

The choice to copy to BytesIO is made from `self.request_body_tempfile_limit`

make_tempfile()

Create a tempfile to store big request body. This API is not stable yet. A 'size' argument might be added.

max_forwards

Gets and sets the `Max-Forwards` header (HTTP spec section 14.31). Converts it using `int`.

method

Gets and sets the `REQUEST_METHOD` key in the environment.

params

A dictionary-like object containing both the parameters from the query string and request body.

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`

pragma

Gets and sets the `Pragma` header (HTTP spec section 14.32).

query_string

Gets and sets the `QUERY_STRING` key in the environment.

range

Gets and sets the `Range` header (HTTP spec section 14.35). Converts it using `Range` object.

referer

Gets and sets the `Referer` header (HTTP spec section 14.36).

referrer

Gets and sets the `Referer` header (HTTP spec 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_path (*resource, *elements, **kw*)

Generates a path (aka a ‘relative URL’, a URL minus the host, scheme, and port) for a *resource*.

This function accepts the same argument as `pyramid.request.Request.resource_url()` and performs the same duty. It just omits the host, port, and scheme information in the return value; only the script_name, path, query parameters, and anchor data are present in the returned string.

 Calling `request.resource_path(resource)` is the same as calling `request.resource_path(resource, app_url=request.script_name).pyramid.request.Request.resource_path()` is, in fact, implemented in terms of `pyramid.request.Request.resource_url()` in just this way. As a result, any `app_url` passed within the `**kw` values to `route_path` will be ignored. `scheme`, `host`, and `port` are also ignored.

response

This attribute is actually a “reified” property which returns an instance of the `pyramid.response.Response` class. The response object returned does not exist until this attribute is accessed. Subsequent accesses will return the same Response object.

The `request.response` API is used by renderers. A render obtains the response object it will return from a view that uses that renderer by accessing `request.response`. Therefore, it’s possible to use the `request.response` API to set up a response object with “the right” attributes (e.g. by calling `request.response.set_cookie()`) within a view that uses a renderer. Mutations to this response object will be preserved in the response sent to the client.

scheme

Gets and sets the `wsgi.url_scheme` key in the environment.

script_name

Gets and sets the `SCRIPT_NAME` key in the environment.

send (*application=None, 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.

If `application` is not given, this will send the request to `self.make_default_send_app()`

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 using `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

str_GET

<Deprecated attribute None>

str_POST

<Deprecated attribute None>

str_cookies

<Deprecated attribute None>

str_params

<Deprecated attribute None>

text

Get/set the text value of the body

upath_info

Gets and sets the `PATH_INFO` key in the environment.

url

The full request URL, including `QUERY_STRING`

url_encoding

Gets and sets the `webob.url_encoding` key in the environment.

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

Gets and sets the `SCRIPT_NAME` key in the environment.

user_agent

Gets and sets the `User-Agent` header (HTTP spec section 14.43).



For information about the API of a *multidict* structure (such as that used as `request.GET`, `request.POST`, and `request.params`), see `pyramid.interfaces.IMultiDict`.

pyramid.response

class Response (*body=None, status=None, headerlist=None, app_iter=None, content_type=None, conditional_response=None, **kw*)

accept_ranges

Gets and sets the Accept-Ranges header (HTTP spec section 14.5).

age

Gets and sets the Age header (HTTP spec section 14.6). Converts it using int.

allow

Gets and sets the Allow header (HTTP spec section 14.7). Converts it using 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 (HTTP spec 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 the Content-Disposition header (HTTP spec section 19.5.1).

content_encoding

Gets and sets the Content-Encoding header (HTTP spec section 14.11).

content_language

Gets and sets the Content-Language header (HTTP spec section 14.12). Converts it using list.

content_length

Gets and sets the Content-Length header (HTTP spec section 14.17). Converts it using int.

content_location

Gets and sets the Content-Location header (HTTP spec section 14.14).

content_md5

Gets and sets the Content-MD5 header (HTTP spec section 14.14).

content_range

Gets and sets the Content-Range header (HTTP spec section 14.16). Converts it using 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 the Date header (HTTP spec section 14.18). Converts it using 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).

etag

Gets and sets the ETag header (HTTP spec section 14.19). Converts it using Entity tag.

expires

Gets and sets the Expires header (HTTP spec section 14.21). Converts it using 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

json

Access the body of the response as JSON

json_body

Access the body of the response as JSON

last_modified

Gets and sets the Last-Modified header (HTTP spec section 14.29). Converts it using HTTP date.

location

Gets and sets the Location header (HTTP spec 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 the Pragma header (HTTP spec section 14.32).

retry_after

Gets and sets the Retry-After header (HTTP spec section 14.37). Converts it using HTTP date or delta seconds.

server

Gets and sets the Server header (HTTP spec 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.

Arguments are:

`key`

The cookie name.

`value`

The cookie value, which should be a string or None. If value is None, it's equivalent to calling the `webob.response.Response.unset_cookie()` method for this cookie key (it effectively deletes the cookie on the client).

`max_age`

An integer representing a number of seconds or None. If this value is an integer, it is used as the Max-Age of the generated cookie. If `expires` is not passed and this value is an integer, the `max_age` value will also influence the Expires value of the cookie (Expires will be set to `now + max_age`). If this value is None, the cookie will not have a Max-Age value (unless `expires` is also sent).

`path`

A string representing the cookie Path value. It defaults to `/`.

`domain`

A string representing the cookie Domain, or None. If domain is None, no Domain value will be sent in the cookie.

`secure`

A boolean. If it's True, the secure flag will be sent in the cookie, if it's False, the secure flag will not be sent in the cookie.

`httponly`

A boolean. If it's True, the HttpOnly flag will be sent in the cookie, if it's False, the HttpOnly flag will not be sent in the cookie.

`comment`

A string representing the cookie Comment value, or None. If comment is None, no Comment value will be sent in the cookie.

`expires`

A `datetime.timedelta` object representing an amount of time or the value None. A non-None value is used to generate the Expires value of the generated cookie. If `max_age` is not passed, but this value is not None, it will influence the Max-Age header (Max-Age will be `'expires_value - datetime.utcnow()'`). If this value is None, the Expires cookie value will be unset (unless `max_age` is also passed).

`overwrite`

If this key is `True`, before setting the cookie, unset any existing cookie.

status

The status string

status_code

The status as an integer

status_int

The status as an integer

text

Get/set the text value of the body (using the charset of the Content-Type)

ubody

Deprecated alias for `.text`

unicode_body

Deprecated alias for `.text`

unset_cookie (*key*, *strict=True*)

Unset a cookie with the given name (remove it from the response).

vary

Gets and sets the Vary header (HTTP spec section 14.44). Converts it using list.

www_authenticate

Gets and sets the WWW-Authenticate header (HTTP spec section 14.47). Converts it using `parse_auth` and `serialize_auth`.

class FileResponse (*path*, *request=None*, *cache_max_age=None*, *content_type=None*, *content_encoding=None*)

A Response object that can be used to serve a static file from disk simply.

`path` is a file path on disk.

`request` must be a Pyramid *request* object. Note that a request *must* be passed if the response is meant to attempt to use the `wsgi.file_wrapper` feature of the web server that you're using to serve your Pyramid application.

`cache_max_age` is the number of seconds that should be used to HTTP cache this response.

`content_type` is the `content_type` of the response.

`content_encoding` is the `content_encoding` of the response. It's generally safe to leave this set to `None` if you're serving a binary file. This argument will be ignored if you also leave `content_type` as `None`.

class FileIter (*file*, *block_size=262144*)

A fixed-block-size iterator for use as a WSGI `app_iter`.

`file` is a Python file pointer (or at least an object with a `read` method that takes a size hint).

`block_size` is an optional block size for iteration.

59.1 Functions

response_adapter (*types_or_ifaces)

Decorator activated via a *scan* which treats the function being decorated as a *response adapter* for the set of types or interfaces passed as *types_or_ifaces to the decorator constructor.

For example, if you scan the following response adapter:

```
from pyramid.response import Response
from pyramid.response import response_adapter

@response_adapter(int)
def myadapter(i):
    return Response(status=i)
```

You can then return an integer from your view callables, and it will be converted into a response with the integer as the status code.

More than one type or interface can be passed as a constructor argument. The decorated response adapter will be called for each type or interface.

```
import json

from pyramid.response import Response
from pyramid.response import response_adapter

@response_adapter(dict, list)
def myadapter(ob):
    return Response(json.dumps(ob))
```

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_adapters')
```

pyramid.scaffolds

class Template (*name*)

Inherit from this base class and override methods to use the Pyramid scaffolding system.

post (*command*, *output_dir*, *vars*)

Called after template is applied.

pre (*command*, *output_dir*, *vars*)

Called before template is applied.

render_template (*content*, *vars*, *filename=None*)

Return a bytestring representing a templated file based on the input (*content*) and the variable names defined (*vars*). *filename* is used for exception reporting.

template_dir ()

Return the template directory of the scaffold. By default, it returns the value of `os.path.join(self.module_dir(), self._template_dir)` (`self.module_dir()` returns the module in which your subclass has been defined). If `self._template_dir` is a tuple this method just returns the value instead of trying to construct a path. If `_template_dir` is a tuple, it should be a 2-element tuple: (`package_name`, `package_relative_path`).

class PyramidTemplate (*name*)

A class that can be used as a base class for Pyramid scaffolding templates.

post (*command*, *output_dir*, *vars*)

Overrides `pyramid.scaffolds.template.Template.post()`, to print “Welcome to Pyramid. Sorry for the convenience.” after a successful scaffolding rendering.

pre (*command*, *output_dir*, *vars*)

Overrides `pyramid.scaffolds.template.Template.pre()`, adding several variables to the default variables list (including `random_string`, and `package_logger`). It also prevents common misnamings (such as naming a package “site” or naming a package logger “root”).

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.

request is passed to the Pyramid application root factory to compute the root. If *request* is *None*, a default will be constructed using the registry's *Request Factory* via the `pyramid.interfaces.IRequestFactory.blank()` method.

prepare (*request=None*, *registry=None*)

This function pushes data onto the Pyramid threadlocal stack (*request* and *registry*), making those objects 'current'. It returns a dictionary useful for bootstrapping a Pyramid application in a scripting environment.

request is passed to the Pyramid application root factory to compute the root. If *request* is *None*, a default will be constructed using the registry's *Request Factory* via the `pyramid.interfaces.IRequestFactory.blank()` method.

If *registry* is not supplied, the last registry loaded from `pyramid.config.global_registries` will be used. If you have loaded more than one Pyramid application in the current process, you may not want to use the last registry loaded, thus you can search the `global_registries` and supply the appropriate one based on your own criteria.

The function returns a dictionary composed of *root*, *closer*, *registry*, *request* and *root_factory*. The *root* returned is the application's root resource object. The *closer* returned is a callable (accepting no arguments) that should be called when your scripting application is finished using the root. *registry* is the registry object passed or the last registry loaded into `pyramid.config.global_registries` if no registry is passed. *request* is the request object passed or the constructed request if no request is passed. *root_factory* is the root factory used to construct the root.

pyramid.security

62.1 Authentication API Functions

authenticated_userid (*request*)

A function that returns the value of the property `pyramid.request.Request.authenticated_userid`.

Deprecated since version 1.5: Use `pyramid.request.Request.authenticated_userid` instead.

unauthenticated_userid (*request*)

A function that returns the value of the property `pyramid.request.Request.unauthenticated_userid`.

Deprecated since version 1.5: Use `pyramid.request.Request.unauthenticated_userid` instead.

effective_principals (*request*)

A function that returns the value of the property `pyramid.request.Request.effective_principals`.

Deprecated since version 1.5: Use `pyramid.request.Request.effective_principals` instead.

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*)

Returns a sequence of header tuples (e.g. `[('Set-Cookie', 'foo=abc')]`) on this request's response. These headers are 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):

```
.. code-block:: python
```

```
from pyramid.security import remember
headers = remember(request,
                    'chrism', password='123', max_age='86400')
response = request.response
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.

62.2 Authorization API Functions

has_permission (*permission*, *context*, *request*)

A function that calls `pyramid.request.Request.has_permission()` and returns its result.

Deprecated since version 1.5: Use `pyramid.request.Request.has_permission()` instead.

Changed in version 1.5a3: If *context* is `None`, then attempt to use the *context* attribute of *self*; if not set, then the `AttributeError` is propagated.

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`. If no view can view found, an exception will be raised.

Changed in version 1.4a4: An exception is raised if no view is found.

62.3 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".

NO_PERMISSION_REQUIRED

A special permission which indicates 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. Its actual value is the string `'__no_permission_required__'`.

62.4 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 the following attributes: `acl`, `ace`, `permission`, `principals`, and `context`. These attributes indicate the security values 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 the following attributes: `acl`, `ace`, `permission`, `principals`, and `context`. These attributes indicate the security values 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.session

signed_serialize (*data*, *secret*)

Serialize any pickleable structure (*data*) and sign it using the *secret* (must be a string). Return the serialization, which includes the signature as its first 40 bytes. The `signed_deserialize` method will deserialize such a value.

This function is useful for creating signed cookies. For example:

```
cookieval = signed_serialize({'a':1}, 'secret')
response.set_cookie('signed_cookie', cookieval)
```

signed_deserialize (*serialized*, *secret*, *hmac*=<module 'hmac' from
'usr/lib/python2.7/hmac.pyc'>)

Deserialize the value returned from `signed_serialize`. If the value cannot be deserialized for any reason, a `ValueError` exception will be raised.

This function is useful for deserializing a signed cookie value created by `signed_serialize`. For example:

```
cookieval = request.cookies['signed_cookie']
data = signed_deserialize(cookieval, 'secret')
```

check_csrf_token (*request*, *token*='csrf_token', *header*='X-CSRF-Token', *raises*=True)

Check the CSRF token in the request's session against the value in `request.params.get(token)` or `request.headers.get(header)`. If a *token* keyword is not supplied to this function, the string `csrf_token` will be used to look up the

token in `request.params`. If a header keyword is not supplied to this function, the string `X-CSRF-Token` will be used to look up the token in `request.headers`.

If the value supplied by param or by header doesn't match the value supplied by `request.session.get_csrf_token()`, and `raises` is `True`, this function will raise an `pyramid.exceptions.BadCSRFToken` exception. If the check does succeed and `raises` is `False`, this function will return `False`. If the CSRF check is successful, this function will return `True` unconditionally.

Note that using this function requires that a *session factory* is configured.

New in version 1.4a2.

SignedCookieSessionFactory (*secret*, *cookie_name*='session', *max_age*=None, *path*='/', *domain*=None, *secure*=False, *httponly*=False, *set_on_exception*=True, *timeout*=1200, *reissue_time*=0, *hashalg*='sha512', *salt*='pyramid.session.', *serializer*=None)

New in version 1.5.

Configure a *session factory* which will provide signed cookie-based sessions. The return value of this function is a *session factory*, which may be provided as the `session_factory` argument of a `pyramid.config.Configurator` constructor, or used as the `session_factory` argument of the `pyramid.config.Configurator.set_session_factory()` method.

The session factory returned by this function will create sessions which are limited to storing fewer than 4000 bytes of data (as the payload must fit into a single cookie).

Parameters:

secret A string which is used to sign the cookie. The secret should be at least as long as the block size of the selected hash algorithm. For `sha512` this would mean a 128 bit (64 character) secret. It should be unique within the set of secret values provided to Pyramid for its various subsystems (see *Admonishment Against Secret-Sharing*).

hashalg The HMAC digest algorithm to use for signing. The algorithm must be supported by the `hashlib` library. Default: `'sha512'`.

salt A namespace to avoid collisions between different uses of a shared secret. Reusing a secret for different parts of an application is strongly discouraged (see *Admonishment Against Secret-Sharing*). Default: `'pyramid.session.'`.

cookie_name The name of the cookie used for sessioning. Default: `'session'`.

max_age The maximum age of the cookie used for sessioning (in seconds). Default: `None` (browser scope).

path The path used for the session cookie. Default: `'/'`.

domain The domain used for the session cookie. Default: `None` (no domain).

secure The ‘secure’ flag of the session cookie. Default: `False`.

httponly Hide the cookie from Javascript by setting the ‘HttpOnly’ flag of the session cookie. Default: `False`.

timeout A number of seconds of inactivity before a session times out. If `None` then the cookie never expires. This lifetime only applies to the *value* within the cookie. Meaning that if the cookie expires due to a lower `max_age`, then this setting has no effect. Default: `1200`.

reissue_time The number of seconds that must pass before the cookie is automatically reissued as the result of accessing the session. The duration is measured as the number of seconds since the last session cookie was issued and ‘now’. If this value is `0`, a new cookie will be reissued on every request accessing the session. If `None` then the cookie’s lifetime will never be extended.

A good rule of thumb: if you want auto-expired cookies based on inactivity: set the `timeout` value to `1200` (20 mins) and set the `reissue_time` value to perhaps a tenth of the `timeout` value (`120` or `2` mins). It’s nonsensical to set the `timeout` value lower than the `reissue_time` value, as the ticket will never be reissued. However, such a configuration is not explicitly prevented.

Default: `0`.

set_on_exception If `True`, set a session cookie even if an exception occurs while rendering a view. Default: `True`.

serializer An object with two methods: `loads` and `dumps`. The `loads` method should accept bytes and return a Python object. The `dumps` method should accept a Python object and return bytes. A `ValueError` should be raised for malformed inputs. If a serializer is not passed, the `pyramid.session.PickleSerializer` serializer will be used.

```
UnencryptedCookieSessionFactoryConfig (secret, timeout=1200,  
                                           cookie_name='session',  
                                           cookie_max_age=None,  
                                           cookie_path='/', cookie_domain=None,  
                                           cookie_secure=False,  
                                           cookie_httponly=False,  
                                           cookie_on_exception=True,  
                                           signed_serialize=<function  
                                           signed_serialize at 0x7fd58af3be60>,  
                                           signed_deserialize=<function  
                                           signed_deserialize at 0x7fd58af3bed8>)
```

Deprecated since version 1.5: Use `pyramid.session.SignedCookieSessionFactory()` instead. Caveat: Cookies generated using `SignedCookieSessionFactory` are not compatible with cookies generated using `UnencryptedCookieSessionFactory`, so existing user session data will be destroyed if you switch to it.

Configure a *session factory* which will provide unencrypted (but signed) cookie-based sessions. The return value of this function is a *session factory*, which may be provided as the `session_factory` argument of a `pyramid.config.Configurator` constructor, or used as the `session_factory` argument of the `pyramid.config.Configurator.set_session_factory()` method.

The session factory returned by this function will create sessions which are limited to storing fewer than 4000 bytes of data (as the payload must fit into a single cookie).

Parameters:

secret A string which is used to sign the cookie.

timeout A number of seconds of inactivity before a session times out.

cookie_name The name of the cookie used for sessioning.

cookie_max_age The maximum age of the cookie used for sessioning (in seconds). Default: None (browser scope).

cookie_path The path used for the session cookie.

cookie_domain The domain used for the session cookie. Default: None (no domain).

cookie_secure The 'secure' flag of the session cookie.

cookie_httponly The 'httpOnly' flag of the session cookie.

cookie_on_exception If True, set a session cookie even if an exception occurs while rendering a view.

signed_serialize A callable which takes more or less arbitrary Python data structure and a secret and returns a signed serialization in bytes. Default: `signed_serialize` (using pickle).

signed_deserialize A callable which takes a signed and serialized data structure in bytes and a secret and returns the original data structure if the signature is valid. Default: `signed_deserialize` (using pickle).

BaseCookieSessionFactory (*serializer*, *cookie_name*='session', *max_age*=None, *path*='/', *domain*=None, *secure*=False, *httponly*=False, *timeout*=1200, *reissue_time*=0, *set_on_exception*=True)

New in version 1.5.

Configure a *session factory* which will provide cookie-based sessions. The return value of this function is a *session factory*, which may be provided as the `session_factory` argument of a `pyramid.config.Configurator` constructor, or used as the `session_factory` argument of the `pyramid.config.Configurator.set_session_factory()` method.

The session factory returned by this function will create sessions which are limited to storing fewer than 4000 bytes of data (as the payload must fit into a single cookie).

Parameters:

serializer An object with two methods: `loads` and `dumps`. The `loads` method should accept bytes and return a Python object. The `dumps` method should accept a Python object and return bytes. A `ValueError` should be raised for malformed inputs.

cookie_name The name of the cookie used for sessioning. Default: `'session'`.

max_age The maximum age of the cookie used for sessioning (in seconds). Default: `None` (browser scope).

path The path used for the session cookie. Default: `'/'`.

domain The domain used for the session cookie. Default: `None` (no domain).

secure The 'secure' flag of the session cookie. Default: `False`.

httponly Hide the cookie from Javascript by setting the 'HttpOnly' flag of the session cookie. Default: `False`.

timeout A number of seconds of inactivity before a session times out. If `None` then the cookie never expires. This lifetime only applies to the *value* within the cookie. Meaning that if the cookie expires due to a lower `max_age`, then this setting has no effect. Default: 1200.

reissue_time The number of seconds that must pass before the cookie is automatically reissued as the result of a request which accesses the session. The duration is measured as the number of seconds since the last session cookie was issued and 'now'. If this value is 0, a new cookie will be reissued on every request accessing the session. If `None` then the cookie's lifetime will never be extended.

A good rule of thumb: if you want auto-expired cookies based on inactivity: set the `timeout` value to 1200 (20 mins) and set the `reissue_time` value to perhaps a tenth of the `timeout` value (120 or 2 mins). It's nonsensical to set the `timeout` value lower than the `reissue_time` value, as the ticket will never be reissued. However, such a configuration is not explicitly prevented.

Default: 0.

set_on_exception If `True`, set a session cookie even if an exception occurs while rendering a view. Default: `True`.

pyramid.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.

aslist (*value*, *flatten=True*)

Return a list of strings, separating the input based on newlines and, if *flatten*=`True` (the default), also split on spaces within each line.

pyramid.static

```
class static_view (root_dir,          cache_max_age=3600,          package_name=None,
                   use_subpath=False, index='index.html')
```

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 one hour).

`use_subpath` influences whether `request.subpath` will be used as `PATH_INFO` when calling the underlying WSGI application which actually serves the static files. If it is `True`, the static application will consider `request.subpath` as `PATH_INFO` input. If it is `False`, the static application will consider `request.environ[PATH_INFO]` as `PATH_INFO` input. By default, this is `False`.



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.

pyramid.testing

setUp (*registry=None, request=None, hook_zca=True, autocommit=True, settings=None, package=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_registry()` 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.

If `package` is `None` it will be set to the caller's package. The package setting in the `pyramid.config.Configurator` will affect any relative imports made via `pyramid.config.Configurator.include()` or `pyramid.config.Configurator.maybe_dotted()`.

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 of `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()` when called with the argument `hook_zca=True`. If `zope.component` cannot be imported, `unhook_zca` is set to `False`.

testConfig (*registry=None, request=None, hook_zca=True, autocommit=True, settings=None*)

Returns a context manager for test set up.

This context manager calls `pyramid.testing.setUp()` when entering and `pyramid.testing.tearDown()` when exiting.

All arguments are passed directly to `pyramid.testing.setUp()`. If the ZCA is hooked, it will always be un-hooked in `tearDown`.

This context manager allows you to write test code like this:

```
1 with testConfig() as config:
2     config.add_route('bar', '/bar/{id}')
3     req = DummyRequest()
4     resp = myview(req),
```

cleanUp (*arg, **kw)

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 0x7fd5873a1850>, __parent__=<object object at 0x7fd5873a1850>, **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 `DummyRequest` object (incompletely) imitates a *request* object.

The `params`, `environ`, `headers`, `path`, and `cookies` arguments correspond to their *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.

Note that `DummyRequest` does not have complete fidelity with a “real” request. For example, by default, the `DummyRequest` `GET` and `POST` attributes are of type `dict`, unlike a normal `Request`'s `GET` and `POST`, which are of type `MultiDict`. If your code uses the features of `MultiDict`, you should either use a real `pyramid.request.Request` or adapt your `DummyRequest` by replacing the attributes with `MultiDict` instances.

Other similar incompatibilities exist. If you need all the features of a `Request`, use the `pyramid.request.Request` class itself rather than this class while writing tests.

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.renderers.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 `/` character, the path is 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 be 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 *text* (Unicode) as the *path* argument are the same as those for a string. In particular, the text may not have any nonascii characters in it.

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 by 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*, *safe*='')

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 `/` character, the path will be 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 be 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)`

Variant of `pyramid.traversal.traversal_path_info()` suitable for decoding paths that are URL-encoded.

If this function is passed a Unicode object instead of a sequence of bytes as `path`, 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. A `UnicodeEncodeError` will be raised if the Unicode cannot be encoded directly to ASCII.

pyramid.tweens

excview_tween_factory (*handler, registry*)

A *tween* factory which produces a tween that catches an exception raised by downstream tweens (or the main Pyramid request handler) and, if possible, converts it into a Response using an *exception view*.

MAIN

Constant representing the main Pyramid handling function, for use in `under` and `over` arguments to `pyramid.config.Configurator.add_tween()`.

INGRESS

Constant representing the request ingress, for use in `under` and `over` arguments to `pyramid.config.Configurator.add_tween()`.

EXCVIEW

Constant representing the exception view tween, for use in `under` and `over` arguments to `pyramid.config.Configurator.add_tween()`.

pyramid.url

Utility functions for dealing with URLs in pyramid

resource_url (*context, request, *elements, query=None, anchor=None*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.resource_url(resource, *elements, **kw)
```

See `pyramid.request.Request.resource_url()` for more information.

route_url (*route_name, request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.route_url(route_name, *elements, **kw)
```

See `pyramid.request.Request.route_url()` for more information.

current_route_url (*request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.current_route_url(*elements, **kw)
```

See `pyramid.request.Request.current_route_url()` for more information.

route_path (*route_name, request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.route_path(route_name, *elements, **kw)
```

See `pyramid.request.Request.route_path()` for more information.

current_route_path (*request, *elements, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.current_route_path(*elements, **kw)
```

See `pyramid.request.Request.current_route_path()` for more information.

static_url (*path, request, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.static_url(path, **kw)
```

See `pyramid.request.Request.static_url()` for more information.

static_path (*path, request, **kw*)

This is a backwards compatibility function. Its result is the same as calling:

```
request.static_path(path, **kw)
```

See `pyramid.request.Request.static_path()` for more information.

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.

Changed in version 1.5: In a key/value pair, if the value is `None` then it will be dropped from the resulting output.

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.httpexceptions.HTTPForbidden` 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 bytestring representation of the return value of this function by calling `b''.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.httpexceptions.HTTPForbidden` 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 bytestring. 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.httpexceptions.HTTPForbidden* exception will be raised; its *args* attribute explains why the view access was disallowed.

If *secure* is *False*, no permission checking is done.

class view_config (***settings*)

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='site1')
```

pyramid.view.view_config supports the following keyword arguments: *context*, *permission*, *name*, *request_type*, *route_name*, *request_method*, *request_param*, *containment*, *xhr*, *accept*, *header*, *path_info*, *custom_predicates*, *decorator*, *mapper*, *http_cache*, *match_param*, *csrf_token*, *physical_path*, and *predicates*.

The meanings of these arguments are the same as the arguments passed to `pyramid.config.Configurator.add_view()`. If any argument is left out, its default will be the equivalent `add_view` default.

An additional keyword argument named `_depth` is provided for people who wish to reuse this class from another decorator. The default value is 0 and should be specified relative to the `view_config` invocation. It will be passed in to the *venusian* `attach` function as the depth of the callstack when Venusian checks if the decorator is being used in a class or module context. It's not often used, but it can be useful in this circumstance. See the `attach` function in Venusian for more information.

See also:

See also *Adding View Configuration Using the @view_config Decorator* for details about using `pyramid.view.view_config`.



`view_config` will work ONLY on module top level members because of the limitation of `venusian.Scanner.scan`.

class `view_defaults` (settings)**

A class *decorator* which, when applied to a class, will provide defaults for all view configurations that use the class. This decorator accepts all the arguments accepted by `pyramid.view.view_config()`, and each has the same meaning.

See *@view_defaults Class Decorator* for more information.

class `notfound_view_config` (settings)**

New in version 1.3.

An analogue of `pyramid.view.view_config` which registers a *Not Found View*.

The `notfound_view_config` constructor accepts most of the same arguments as the constructor of `pyramid.view.view_config`. It can be used in the same places, and behaves in largely the same way, except it always registers a not found exception view instead of a 'normal' view.

Example:

```
from pyramid.view import notfound_view_config
from pyramid.response import Response

@notfound_view_config()
def notfound(request):
    return Response('Not found, dude!', status='404 Not Found')
```

All arguments except `append_slash` have the same meaning as `pyramid.view.view_config()` and each predicate argument restricts the set of circumstances under which this notfound view will be invoked.

If `append_slash` is `True`, when the Not Found View is invoked, and the current path info does not end in a slash, the notfound logic will attempt to find a *route* that matches the request's path info suffixed with a slash. If such a route exists, Pyramid will issue a redirect to the URL implied by the route; if it does not, Pyramid will return the result of the view callable provided as `view`, as normal.

See *Changing the Not Found View* for detailed usage information.

class `forbidden_view_config` (***settings*)

New in version 1.3.

An analogue of `pyramid.view.view_config` which registers a *forbidden* view.

The `forbidden_view_config` constructor accepts most of the same arguments as the constructor of `pyramid.view.view_config`. It can be used in the same places, and behaves in largely the same way, except it always registers a forbidden exception view instead of a 'normal' view.

Example:

```
from pyramid.view import forbidden_view_config
from pyramid.response import Response

@forbidden_view_config()
def forbidden(request):
    return Response('You are not allowed', status='401 Unauthorized')
```

All arguments passed to this function have the same meaning as `pyramid.view.view_config()` and each predicate argument restricts the set of circumstances under which this notfound view will be invoked.

See *Changing the Forbidden View* for detailed usage information.

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. In particular, a new WSGI environment is generated, and the *subpath* of the request passed to `wsgiapp2` is used as the new request's `PATH_INFO` and everything preceding the subpath is used as the `SCRIPT_NAME`. The new environment is passed to the downstream WSGI application.

Part IV

Glossary and Index

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 whether a particular request can be fulfilled given the *authentication* information in the request.

action Represents a pending configuration statement generated by a call to a *configuration directive*. The set of pending configuration actions are processed when `pyramid.config.Configurator.commit()` is called.

add-on A Python *distribution* that uses Pyramid's extensibility to plug into a Pyramid application and provide extra, configurable services.

Agendaless Consulting A consulting organization formed by Paul Everitt, Tres Seaver, and Chris McDonough.

See also:

See also Agendaless Consulting.

Akhet Akhet is a Pyramid library and demo application with a Pylons-like feel. It's most known for its former application scaffold, which helped users transition from Pylons and those preferring a more Pylons-like API. The scaffold has been retired but the demo plays a similar role.

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 descriptor An instance representing an *asset specification* provided by the `pyramid.path.AssetResolver.resolve()` method. It supports the methods and attributes documented in `pyramid.interfaces.IAssetDescriptor`.

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 the *ZPT* templating specification. It is written and maintained by Malthe Borch. It has several extensions, such as the ability to use bracketed (Mako-style) `${name}` syntax. It is also much faster than the reference implementation of ZPT. Pyramid offers Chameleon templating out of the box in ZPT and text flavors.

configuration declaration An individual method call made to a *configuration directive*, 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`.

configuration directive A method of the *Configurator* which causes a configuration action to occur. The method `pyramid.config.Configurator.add_view()` is a configuration directive, and application developers can add their own directives as necessary (see *Adding Methods to the Configurator via add_directive*).

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

conflict resolution Pyramid attempts to resolve ambiguous configuration statements made by application developers via automatic conflict resolution. Automatic conflict resolution is described in *Automatic Conflict Resolution*. If Pyramid cannot resolve ambiguous configuration statements, it is possible to manually resolve them as described in *Manually Resolving Conflicts*.

console script A script written to the `bin` (on UNIX, or `Scripts` on Windows) directory of a Python installation or *virtualenv* as the result of running `setup.py install` or `setup.py develop`.

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 the combination of *configuration decoration* and a *scan* to configure your Pyramid application.

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:

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:

See also *Setting a Default Permission*.

default root factory If an application does not register a *root factory* at Pyramid configuration time, a *default* root factory is used to create the default root object. Use of the default root object is useful in application which use *URL dispatch* for all URL-to-view code mappings, and does not (knowingly) use traversal otherwise.

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 in views or as `config.registry.settings` in configuration code. Deployment settings can be used as global application values.

discriminator The unique identifier of an *action*.

distribute Distribute is a fork of *setuptools* which runs on both Python 2 and Python 3.

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-related 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 Pyramid 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 response A *response* that is generated as the result of a raised exception being caught by an *exception view*.

Exception view An exception view is a *view callable* which may be invoked by Pyramid when an exception is raised during request processing. See *Custom 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.httpexceptions.HTTPForbidden` 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.

Green Unicorn Aka `gunicorn`, a fast *WSGI* server that runs on UNIX under Python 2.6+ or Python 3.1+. See <http://gunicorn.org/> for detailed information.

Grok A web framework based on Zope 3.

HTTP Exception The set of exception classes defined in `pyramid.httpexceptions`. These can be used to generate responses with various status codes when raised or returned from a *view callable*.

See also:

See also *HTTP Exceptions*.

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:

See also *Localization*.

introspectable An object which implements the attributes and methods described in `pyramid.interfaces.IIntrospectable`. Introspectables are used by the *introspector* to display configuration information about a running Pyramid application. An introspectable is associated with a *action* by virtue of the `pyramid.config.Configurator.action()` method.

introspector An object with the methods described by `pyramid.interfaces.IIntrospector` that is available in both configuration code (for registration) and at runtime (for querying) that allows a developer to introspect configuration statements and relationships between those statements.

Jinja2 A text templating language by Armin Ronacher.

jQuery A popular Javascript library.

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.

Lingua A package by Wichert Akkerman which provides the `pot-create` command to extract translatable messages from Python sources and Chameleon ZPT template files.

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:

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 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`, `add` and `dict_of_lists` to the normal dictionary interface. See *Multidict* and `pyramid.interfaces.IMultiDict`.

Not Found View An *exception view* invoked by Pyramid when the developer explicitly raises a `pyramid.httpexceptions.HTTPNotFound` 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.

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.

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 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 answer the question "is this user allowed to do this". Examples of permissions: `read`, or `view_blog_entries`.

physical path The path required by a traversal which resolve a *resource* starting from the *physical root*. For example, the physical path of the `abc` subobject of the physical root object is `/abc`. Physical paths can also be specified as tuples where the first element is the empty string (representing the root), and every other element is a Unicode object, e.g. `(' ', 'abc')`. Physical paths are also sometimes called "traversal paths".

physical root The object returned by the application *root factory*. Unlike the *virtual root* of a request, it is not impacted by *Virtual Hosting*: it will always be the actual object returned by the root factory, never a subobject.

pipeline The *PasteDeploy* 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* and *distribute* 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:

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*.

predicate factory A callable which is used by a third party during the registration of a route, view, or subscriber predicates to extend the configuration system. See *Adding A Third Party View, Route, or Subscriber Predicate* for more information.

pregenerator A pregenerator is a function associated by a developer with a *route*. It is called by `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.

PyPy PyPy is an “alternative implementation of the Python language”: <http://pypy.org/>

Pyramid Cookbook Additional documentation for Pyramid which presents topical, practical uses of Pyramid: http://docs.pylonsproject.org/projects/pyramid_cookbook/en/latest.

pyramid_debugtoolbar A Pyramid add-on which displays a helpful debug toolbar “on top of” HTML pages rendered by your application, displaying request, routing, and database information. `pyramid_debugtoolbar` is configured into the `development.ini` of all applications which use a Pyramid *scaffold*. For more information, see http://docs.pylonsproject.org/projects/pyramid_debugtoolbar/en/latest/.

pyramid_exclog A package which logs Pyramid application exception (error) information to a standard Python logger. This add-on is most useful when used in production applications, because the logger can be configured to log to a file, to UNIX syslog, to the Windows Event Log, or even to email. See its documentation.

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 scaffold named `pyramid_jinja2_starter`, which creates an application package based on the Jinja2 templating system.

pyramid_redis_sessions A package by Eric Rasmussen which allows you to store Pyramid session data in a Redis database. See https://pypi.python.org/pypi/pyramid_redis_sessions 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* or *configuration decoration*.

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 by a `pyramid.event.BeforeRender` event.

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 An object that represents an HTTP request, usually an instance of the `pyramid.request.Request` class. 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 Pyramid-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 returned by a *view callable* that represents response data returned to the requesting user agent. It must implement the `pyramid.interfaces.IResponse` interface. A response object is typically an instance of the `pyramid.response.Response` class or a subclass such as `pyramid.httpexceptions.HTTPFound`. See *Request and Response Objects* for information about response objects.

response adapter A callable which accepts an arbitrary object and “converts” it to a `pyramid.response.Response` object. See *Changing How Pyramid Treats View Responses* for more information.

response callback A user-defined callback executed by the *router* at a point after a *response* object is successfully created.

See also:

See also *Using Response Callbacks*.

reStructuredText A plain text markup format that is the defacto standard for documenting Python projects. The Pyramid documentation is written in reStructuredText.

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 creates a default root object using the *default root factory*.

route A single pattern matched by the *url dispatch* subsystem, which generally resolves to a *root factory* (and then ultimately a *view*).

See also:

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*.

scaffold A project template that generates some of the major parts of a Pyramid application and helps users to quickly get started writing larger applications. Scaffolds are usually used via the `pcreate` command.

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. See *Using The Default Session Factory*, *Using Alternate Session Factories* and `pyramid.config.Configurator.set_session_factory()` for more information.

setuptools Setuptools builds on Python's `distutils` to provide easier building, distribution, and installation of libraries and applications. As of this writing, setuptools runs under Python 2, but not under Python 3. You can use *distribute* under Python 3 instead.

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:

See also the `stdlib` 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).

tween A bit of code that sits between the Pyramid router’s main request handling function and the upstream WSGI component that uses Pyramid as its ‘app’. The word “tween” is a contraction of “between”. A tween may be used by Pyramid framework extensions, to provide, for example, Pyramid-specific view timing support, bookkeeping code that examines exceptions before they are returned to the upstream WSGI application, or a variety of other features. Tweens behave a bit like *WSGI middleware* but they have the benefit of running in a context in which they have access to the *Pyramid application registry* as well as the Pyramid rendering machinery. See *Registering Tweens*.

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 unless *Virtual Hosting* is in use.

virtualenv A term referring both to an isolated Python environment, or the leading tool that allows one to create such environments.

Note: whenever you encounter commands prefixed with `$VENV` (Unix) or `%VENV` (Windows), know that that is the environment variable whose value is the root of the virtual environment in question.

Waitress A *WSGI* server that runs on UNIX and Windows under Python 2.6+ and Python 3.2+. Projects generated via Pyramid scaffolding use Waitress as a *WSGI* server. See <http://docs.pylonsproject.org/projects/waitress/en/latest/> for detailed information.

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.

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.