Zend Framework 2 Documentation

Release 2.1.6dev

Zend Technologies Ltd.

April 17, 2013

CONTENTS

1 Overview **1**

2 Installation **3**

3 Getting Started with Zend Framework 2 **5**
3.1 Some assumptions 5
3.2 The tutorial application 5

4 Getting started: A skeleton application **7**
4.1 Virtual host 8

5 Modules **9**
5.1 Setting up the Album module 9
5.2 Configuration 11
5.3 Informing the application about our new module 11

6 Routing and controllers **13**

7 Create the controller **15**
7.1 Initialise the view scripts 16

8 Database and models **17**
8.1 The database 17
8.2 The model files 17
8.3 Using ServiceManager to configure the table gateway and inject into the AlbumTable 19
8.4 Back to the controller 21
8.5 Listing albums 21

9 Styling and Translations **25**

10 Forms and actions **27**
10.1 Adding new albums 27
10.2 Editing an album 31
10.3 Deleting an album 34
10.4 Ensuring that the home page displays the list of albums 35

11 Conclusion **37**

12 Zend Framework Tool (ZFTool) **39**
12.1 Installation using Composer 39
12.2 Manual installation 39

12.3 Without installation, using the PHAR file . . . 39
12.4 Usage . . . 40

13 Learning Dependency Injection **43**
13.1 Very brief introduction to Di. . . . 43
13.2 Very brief introduction to Di Container. . . . 43
13.3 Simplest usage case (2 classes, one consumes the other) . . . 43
13.4 Simplest Usage Case Without Type-hints . . . 45
13.5 Simplest usage case with Compiled Definition . . . 47
13.6 Creating a precompiled definition for others to use . . . 48
13.7 Using Multiple Definitions From Multiple Sources . . . 48
13.8 Generating Service Locators . . . 49

14 Unit Testing a Zend Framework 2 application **53**
14.1 Setting up the tests directory . . . 53
14.2 Bootstrapping your tests . . . 53
14.3 Your first controller test . . . 56
14.4 A failing test case . . . 57
14.5 Configuring the service manager for the tests . . . 57
14.6 Testing actions with POST . . . 59
14.7 Testing model entities . . . 59
14.8 Testing model tables . . . 61
14.9 Conclusion . . . 64

15 Using the EventManager **65**
15.1 Terminology . . . 65
15.2 Getting started . . . 65
15.3 Shared managers . . . 67

16 Wildcards **69**

17 Listener aggregates **71**
17.1 Introspecting results . . . 72
17.2 Short-ciruiting listener execution . . . 72
17.3 Keeping it in order . . . 73
17.4 Custom event objects . . . 74
17.5 Putting it together: Implementing a simple caching system . . . 75
17.6 Conclusion . . . 77

18 Using Zend\Navigation in your Album Module **79**
18.1 Preparation . . . 79
18.2 Setting Up Zend\Navigation . . . 79
18.3 Configuring our Site Map . . . 80
18.4 Adding the Menu View Helper . . . 81
18.5 Adding Breadcrumbs . . . 81

19 Using Zend\Paginator in your Album Module **83**
19.1 Preparation . . . 83
19.2 Modifying the AlbumTable . . . 83
19.3 Modifying the AlbumController . . . 84
19.4 Updating the View Script . . . 85
19.5 Creating the Pagination Control Partial . . . 85

20 Using the PaginationControl View Helper **87**

21 Setting up a database adapter **89**
21.1 Introduction . . . 89
21.2 Basic setup . . . 89
21.3 Setting a static adapter . . . 90

22 Migration from Zend Framework 1 **91**

23 Namespacing Old Classes **93**
23.1 Namespacing a ZF1 Application . . . 93
23.2 HOWTO Namespace Your Models . . . 95

24 Running Zend Framework 2 and Zend Framework 1 in parallel **97**
24.1 Use ZF2 in a ZF1 project . . . 97
24.2 Use ZF1 in a ZF2 project . . . 98
24.3 Run ZF1 and ZF2 together . . . 98

25 Introduction to Zend\Authentication **101**
25.1 Adapters . . . 101
25.2 Results . . . 102
25.3 Identity Persistence . . . 103
25.4 Usage . . . 106

26 Database Table Authentication **109**
26.1 Introduction . . . 109
26.2 Basic Usage . . . 109
26.3 Advanced Usage: Persisting a DbTable Result Object . . . 111

27 Digest Authentication **115**
27.1 Introduction . . . 115
27.2 Specifics . . . 115
27.3 Identity . . . 115

28 HTTP Authentication Adapter **117**
28.1 Introduction . . . 117
28.2 Design Overview . . . 117
28.3 Configuration Options . . . 117
28.4 Resolvers . . . 118
28.5 Basic Usage . . . 119

29 LDAP Authentication **121**
29.1 Introduction . . . 121
29.2 Usage . . . 121
29.3 The API . . . 123
29.4 Server Options . . . 124
29.5 Collecting Debugging Messages . . . 126
29.6 Common Options for Specific Servers . . . 126

30 Authentication Validator **129**
30.1 Introduction . . . 129
30.2 Basic Usage . . . 129

31 Introduction to Zend\Barcode **131**

32 Barcode creation using Zend\Barcode\Barcode class **133**
32.1 Using Zend\Barcode\Barcode::factory . . . 133
32.2 Drawing a barcode . . . 134

32.3 Rendering a barcode . . . 135

33 Zend\Barcode\Barcode Objects **137**
33.1 Common Options . . . 137
33.2 Common Additional Getters . . . 139

34 Description of shipped barcodes **141**
34.1 Zend\Barcode\Object\Error . . . 141
34.2 Zend\Barcode\Object\Code128 . . . 141
34.3 Zend\Barcode\Object\Codabar . . . 141
34.4 Zend\Barcode\Object\Code25 . . . 142
34.5 Zend\Barcode\Object\Code25interleaved . . . 142
34.6 Zend\Barcode\Object\Ean2 . . . 142
34.7 Zend\Barcode\Object\Ean5 . . . 143
34.8 Zend\Barcode\Object\Ean8 . . . 143
34.9 Zend\Barcode\Object\Ean13 . . . 143
34.10 Zend\Barcode\Object\Code39 . . . 144
34.11 Zend\Barcode\Object\Identcode . . . 144
34.12 Zend\Barcode\Object\Itf14 . . . 144
34.13 Zend\Barcode\Object\Leitcode . . . 145
34.14 Zend\Barcode\Object\Planet . . . 145
34.15 Zend\Barcode\Object\Postnet . . . 145
34.16 Zend\Barcode\Object\Royalmail . . . 146
34.17 Zend\Barcode\Object\Upca . . . 146
34.18 Zend\Barcode\Object\Upce . . . 146

35 Zend\Barcode Renderers **149**
35.1 Common Options . . . 149
35.2 Zend\Barcode\Renderer\Image . . . 150
35.3 Zend\Barcode\Renderer\Pdf . . . 150

36 Zend\Cache\Storage\Adapter **151**
36.1 Overview . . . 151
36.2 Quick Start . . . 151
36.3 Basic Configuration Options . . . 152
36.4 The StorageInterface . . . 152
36.5 The AvailableSpaceCapableInterface . . . 154
36.6 The TotalSpaceCapableInterface . . . 154
36.7 The ClearByNamespaceInterface . . . 154
36.8 The ClearByPrefixInterface . . . 155
36.9 The ClearExpiredInterface . . . 155
36.10 The FlushableInterface . . . 155
36.11 The IterableInterface . . . 155
36.12 The OptimizableInterface . . . 155
36.13 The TaggableInterface . . . 156
36.14 The Apc Adapter . . . 156
36.15 The Dba Adapter . . . 157
36.16 The Filesystem Adapter . . . 158
36.17 The Memcached Adapter . . . 159
36.18 The Memory Adapter . . . 160
36.19 The WinCache Adapter . . . 161
36.20 The XCache Adapter . . . 162
36.21 The ZendServerDisk Adapter . . . 163
36.22 The ZendServerShm Adapter . . . 163
36.23 Examples . . . 164

37 Zend\Cache\Storage\Capabilities **167**
37.1 Overview . . . 167
37.2 Available Methods . . . 167
37.3 Examples . . . 169

38 Zend\Cache\Storage\Plugin **171**
38.1 Overview . . . 171
38.2 Quick Start . . . 171
38.3 The ClearExpiredByFactor Plugin . . . 172
38.4 The ExceptionHandler Plugin . . . 172
38.5 The IgnoreUserAbort Plugin . . . 172
38.6 The OptimizeByFactor Plugin . . . 172
38.7 The Serializer Plugin . . . 173
38.8 Available Methods . . . 173
38.9 Examples . . . 174

39 Zend\Cache\Pattern **175**
39.1 Overview . . . 175
39.2 Quick Start . . . 175
39.3 Available Methods . . . 175

40 Zend\Cache\Pattern\CallbackCache **177**
40.1 Overview . . . 177
40.2 Quick Start . . . 177
40.3 Configuration Options . . . 177
40.4 Available Methods . . . 178
40.5 Examples . . . 178

41 Zend\Cache\Pattern\ClassCache **179**
41.1 Overview . . . 179
41.2 Quick Start . . . 179
41.3 Configuration Options . . . 179
41.4 Available Methods . . . 179
41.5 Examples . . . 180

42 Zend\Cache\Pattern\ObjectCache **183**
42.1 Overview . . . 183
42.2 Quick Start . . . 183
42.3 Configuration Options . . . 183
42.4 Available Methods . . . 184
42.5 Examples . . . 184

43 Zend\Cache\Pattern\OutputCache **187**
43.1 Overview . . . 187
43.2 Quick Start . . . 187
43.3 Configuration Options . . . 187
43.4 Available Methods . . . 187
43.5 Examples . . . 188

44 Zend\Cache\Pattern\CaptureCache **189**
44.1 Overview . . . 189
44.2 Quick Start . . . 189
44.3 Configuration Options . . . 190
44.4 Available Methods . . . 190
44.5 Examples . . . 191

45 Introduction to Zend\Captcha **193**

46 Captcha Operation **195**

47 CAPTCHA Adapters **197**
47.1 Zend\Captcha\AbstractWord . . . 197
47.2 Zend\Captcha\Dumb . . . 198
47.3 Zend\Captcha\Figlet . . . 198
47.4 Zend\Captcha\Image . . . 198
47.5 Zend\Captcha\ReCaptcha . . . 199

48 Introduction to Zend\Config **201**
48.1 Using Zend\Config\Config with a Reader Class . . . 201
48.2 Using Zend\Config\Config with a PHP Configuration File . . . 202

49 Theory of Operation **203**

50 Zend\Config\Reader **205**
50.1 Zend\Config\Reader\Ini . . . 205
50.2 Zend\Config\Reader\Xml . . . 206
50.3 Zend\Config\Reader\Json . . . 207
50.4 Zend\Config\Reader\Yaml . . . 208

51 Zend\Config\Writer **211**
51.1 Zend\Config\Writer\Ini . . . 211
51.2 Zend\Config\Writer\Xml . . . 212
51.3 Zend\Config\Writer\PhpArray . . . 213
51.4 Zend\Config\Writer\Json . . . 213
51.5 Zend\Config\Writer\Yaml . . . 214

52 Zend\Config\Processor **217**
52.1 Zend\Config\Processor\Constant . . . 217
52.2 Zend\Config\Processor\Filter . . . 217
52.3 Zend\Config\Processor\Queue . . . 218
52.4 Zend\Config\Processor\Token . . . 218
52.5 Zend\Config\Processor\Translator . . . 219

53 The Factory **221**
53.1 Loading configuration file . . . 221
53.2 Storing configuration file . . . 221

54 Introduction to Zend\Console **223**
54.1 Writing console routes . . . 223
54.2 Handling console requests . . . 225
54.3 Adding console usage info . . . 226

55 Console routes and routing **229**
55.1 Router configuration . . . 229
55.2 Basic route . . . 230
55.3 Catchall route . . . 234
55.4 Console routes cheat-sheet . . . 235

56 Console-aware modules **237**
56.1 Application banner . . . 237
56.2 Usage information . . . 238
56.3 Best practices . . . 241

57 Console-aware action controllers **243**
57.1 Handling console requests . . . 243
57.2 Sending output to console . . . 245
57.3 Are we in a console? . . . 245
57.4 Reading values from console parameters . . . 247

58 Console adapters **251**
58.1 Retrieving console adapter . . . 251
58.2 Using console adapter . . . 252

59 Console prompts **255**
59.1 Confirm . . . 255
59.2 Line . . . 256
59.3 Char . . . 256
59.4 Select . . . 257

60 Introduction to Zend\Crypt **259**

61 Encrypt/decrypt using block ciphers **261**

62 Key derivation function **263**
62.1 Pbkdf2 adapter . . . 263
62.2 SaltedS2k adapter . . . 264
62.3 Scrypt adapter . . . 264

63 Password **267**
63.1 Bcrypt . . . 267
63.2 Apache . . . 268

64 Public key cryptography **271**
64.1 Diffie-Hellman . . . 271
64.2 RSA . . . 272

65 Zend\Db\Adapter **275**
65.1 Creating an Adapter - Quickstart . . . 275
65.2 Creating an Adapter Using Dependency Injection . . . 276
65.3 Query Preparation Through Zend\Db\Adapter\Adapter::query() . . . 276
65.4 Query Execution Through Zend\Db\Adapter\Adapter::query() . . . 277
65.5 Creating Statements . . . 277
65.6 Using the Driver Object . . . 277
65.7 Using The Platform Object . . . 279
65.8 Using The Parameter Container . . . 280
65.9 Examples . . . 281

66 Zend\Db\ResultSet **283**
66.1 Quickstart . . . 283
66.2 Zend\Db\ResultSet\ResultSet and Zend\Db\ResultSet\AbstractResultSet . . . 284
66.3 Zend\Db\ResultSet\HydratingResultSet . . . 284

67 Zend\Db\Sql **287**
67.1 Zend\Db\Sql\Sql (Quickstart) . . . 287
67.2 Zend\Db\Sql's Select, Insert, Update and Delete . . . 288
67.3 Zend\Db\Sql\Select . . . 288
67.4 Zend\Db\Sql\Insert . . . 291
67.5 Zend\Db\Sql\Update . . . 292
67.6 Zend\Db\Sql\Delete . . . 292

67.7 Zend\Db\Sql\Where & Zend\Db\Sql\Having . . . 292

68 Zend\Db\TableGateway **297**
68.1 Basic Usage . . . 297
68.2 TableGateway Features . . . 299

69 Zend\Db\RowGateway **301**
69.1 Quickstart . . . 301
69.2 ActiveRecord Style Objects . . . 302

70 Zend\Db\Metadata **303**
70.1 Basic Usage . . . 303

71 Introduction to Zend\Di **307**
71.1 Dependency Injection . . . 307
71.2 Dependency Injection Containers . . . 307

72 Zend\Di Quickstart **309**

73 Zend\Di Definition **313**
73.1 DefinitionList . . . 313
73.2 RuntimeDefinition . . . 313
73.3 CompilerDefinition . . . 314
73.4 ClassDefinition . . . 315

74 Zend\Di InstanceManager **317**
74.1 Parameters . . . 317
74.2 Preferences . . . 318
74.3 Aliases . . . 319

75 Zend\Di Configuration **321**

76 Zend\Di Debugging & Complex Use Cases **323**
76.1 Debugging a DiC . . . 323
76.2 Complex Use Cases . . . 323

77 Introduction to Zend\Dom **327**

78 Zend\Dom\Query **329**
78.1 Theory of Operation . . . 329
78.2 Methods Available . . . 330

79 Introduction to Zend\Escaper **333**
79.1 Overview . . . 333
79.2 What Zend\Escaper is not . . . 333

80 Theory of Operation **335**
80.1 The Problem with Inconsistent Functionality . . . 335
80.2 Why Contextual Escaping? . . . 336

81 Configuring Zend\Escaper **339**

82 Escaping HTML **341**
82.1 Examples of Bad HTML Escaping . . . 341
82.2 Examples of Good HTML Escaping . . . 342

83 Escaping HTML Attributes **343**
83.1 Examples of Bad HTML Attribute Escaping . . . 343
83.2 Examples of Good HTML Attribute Escaping . . . 344

84 Escaping Javascript **347**
84.1 Examples of Bad Javascript Escaping . . . 347
84.2 Examples of Good Javascript Escaping . . . 348

85 Escaping Cascading Style Sheets **349**
85.1 Examples of Bad CSS Escaping . . . 349
85.2 Examples of Good CSS Escaping . . . 349

86 Escaping URLs **351**
86.1 Examples of Bad URL Escaping . . . 351
86.2 Examples of Good URL Escaping . . . 351

87 The EventManager **353**
87.1 Overview . . . 353
87.2 Quick Start . . . 353
87.3 Configuration Options . . . 356
87.4 Available Methods . . . 356
87.5 Examples . . . 358

88 Introduction to Zend\Feed **363**

89 Importing Feeds **365**
89.1 Dumping the contents of a feed . . . 365

90 Retrieving Feeds from Web Pages **367**

91 Consuming an RSS Feed **369**

92 Consuming an Atom Feed **371**

93 Consuming a Single Atom Entry **373**

94 Zend\Feed and Security **375**
94.1 Introduction . . . 375
94.2 Filtering data using HTMLPurifier . . . 375
94.3 Escaping data using Zend\Escaper . . . 377

95 Zend\Feed\Reader\Reader **379**
95.1 Introduction . . . 379
95.2 Importing Feeds . . . 379
95.3 Retrieving Underlying Feed and Entry Sources . . . 380
95.4 Cache Support and Intelligent Requests . . . 381
95.5 Locating Feed URIs from Websites . . . 382
95.6 Attribute Collections . . . 383
95.7 Retrieving Feed Information . . . 383
95.8 Retrieving Entry/Item Information . . . 386
95.9 Extending Feed and Entry APIs . . . 387

96 Zend\Feed\Writer\Writer **393**
96.1 Introduction . . . 393
96.2 Architecture . . . 393
96.3 Getting Started . . . 394

96.4 Setting Feed Data Points . . . 396
96.5 Setting Entry Data Points . . . 398

97 Zend\Feed\PubSubHubbub **401**
97.1 What is PubSubHubbub? . . . 401
97.2 Architecture . . . 401
97.3 Zend\Feed\PubSubHubbub\Publisher . . . 402
97.4 Zend\Feed\PubSubHubbub\Subscriber . . . 403

98 Zend\File\ClassFileLocator **409**
98.1 Overview . . . 409
98.2 Available Methods . . . 409
98.3 Examples . . . 409

99 Introduction to Zend\Filter **411**
99.1 What is a filter? . . . 411
99.2 Basic usage of filters . . . 411

100 Using the StaticFilter **413**
100.1 Double filtering . . . 413

101 Standard Filter Classes **415**
101.1 Alnum . . . 415
101.2 Alpha . . . 416
101.3 BaseName . . . 416
101.4 Boolean . . . 417
101.5 Callback . . . 419
101.6 Compress and Decompress . . . 421
101.7 Digits . . . 426
101.8 Dir . . . 427
101.9 Encrypt and Decrypt . . . 427
101.10 HtmlEntities . . . 433
101.11 Int . . . 435
101.12 Null . . . 435
101.13 NumberFormat . . . 436
101.14 PregReplace . . . 437
101.15 RealPath . . . 438
101.16 StringToLower . . . 439
101.17 StringToUpper . . . 439
101.18 StringTrim . . . 440
101.19 StripNewLines . . . 441
101.20 StripTags . . . 441
101.21 UriNormalize . . . 442

102 Word Filters **445**
102.1 CamelCaseToDash . . . 445
102.2 CamelCaseToSeparator . . . 445
102.3 CamelCaseToUnderscore . . . 446
102.4 DashToCamelCase . . . 446
102.5 DashToSeparator . . . 447
102.6 DashToUnderscore . . . 447
102.7 SeparatorToCamelCase . . . 448
102.8 SeparatorToDash . . . 448
102.9 SeparatorToSeparator . . . 449
102.10 UnderscoreToCamelCase . . . 450

102.11 UnderscoreToSeparator . . . 450
102.12 UnderscoreToDash . . . 451

103 File Filter Classes **453**
103.1 Decrypt . . . 453
103.2 Encrypt . . . 453
103.3 Lowercase . . . 453
103.4 Rename . . . 453
103.5 RenameUpload . . . 455
103.6 Uppercase . . . 456

104 Filter Chains **457**
104.1 Setting Filter Chain Order . . . 457
104.2 Using the Plugin Manager . . . 457

105 Zend\Filter\Inflector **459**
105.1 Operation . . . 459
105.2 Using Custom Filters . . . 460
105.3 Setting the Inflector Target . . . 460
105.4 Inflection Rules . . . 461
105.5 Utility Methods . . . 463
105.6 Using a Traversable or an array with Zend\Filter\Inflector . . . 463

106 Writing Filters **465**

107 Introduction to Zend\Form **467**

108 Form Quick Start **469**
108.1 Programmatic Form Creation . . . 469
108.2 Creation via Factory . . . 470
108.3 Factory-backed Form Extension . . . 473
108.4 Validating Forms . . . 475
108.5 Hinting to the Input Filter . . . 475
108.6 Binding an object . . . 477
108.7 Rendering . . . 479
108.8 Validation Groups . . . 481
108.9 Using Annotations . . . 482

109 Form Collections **485**
109.1 Creating Fieldsets . . . 488
109.2 The Form Element . . . 492
109.3 The Controller . . . 493
109.4 The View . . . 493
109.5 Adding New Elements Dynamically . . . 494
109.6 Validation groups for fieldsets and collection . . . 496

110 File Uploading **499**
110.1 Standard Example . . . 499
110.2 File Post-Redirect-Get Plugin . . . 502
110.3 HTML5 Multi-File Uploads . . . 504
110.4 Upload Progress . . . 505
110.5 Additional Info . . . 509

111 Advanced use of forms **511**
111.1 Short names . . . 511

111.2 Creating custom elements . . . 511
111.3 Handling dependencies . . . 515
111.4 The specific case of initializers . . . 517

112Form Elements **519**
112.1 Introduction . . . 519
112.2 Element Base Class . . . 519
112.3 Standard Elements . . . 521
112.4 HTML5 Elements . . . 536

113Form View Helpers **549**
113.1 Introduction . . . 549
113.2 Standard Helpers . . . 549
113.3 HTML5 Helpers . . . 560
113.4 File Upload Progress Helpers . . . 563

114Overview of Zend\Http **565**
114.1 Overview . . . 565
114.2 Zend\Http Request, Response and Headers . . . 565

115The Request Class **567**
115.1 Overview . . . 567
115.2 Quick Start . . . 567
115.3 Configuration Options . . . 568
115.4 Available Methods . . . 568
115.5 Examples . . . 571

116The Response Class **573**
116.1 Overview . . . 573
116.2 Quick Start . . . 573
116.3 Configuration Options . . . 574
116.4 Available Methods . . . 574
116.5 Examples . . . 576

117The Headers Class **579**
117.1 Overview . . . 579
117.2 Quick Start . . . 579
117.3 Configuration Options . . . 580
117.4 Available Methods . . . 580
117.5 Zend\Http\Header* Base Methods . . . 582
117.6 List of HTTP Header Types . . . 582
117.7 Examples . . . 583

118HTTP Client - Overview **587**
118.1 Overview . . . 587
118.2 Quick Start . . . 587
118.3 Configuration Options . . . 588
118.4 Available Methods . . . 588
118.5 Examples . . . 592

119HTTP Client - Connection Adapters **595**
119.1 Overview . . . 595
119.2 The Socket Adapter . . . 595
119.3 The Proxy Adapter . . . 598
119.4 The cURL Adapter . . . 599

119.5 The Test Adapter . . . 599
119.6 Creating your own connection adapters . . . 602

120HTTP Client - Advanced Usage **605**
120.1 HTTP Redirections . . . 605
120.2 Adding Cookies and Using Cookie Persistence . . . 605
120.3 Setting Custom Request Headers . . . 607
120.4 File Uploads . . . 607
120.5 Sending Raw POST Data . . . 608
120.6 HTTP Authentication . . . 609
120.7 Sending Multiple Requests With the Same Client . . . 609
120.8 Data Streaming . . . 610

121HTTP Client - Static Usage **613**
121.1 Overview . . . 613
121.2 Quick Start . . . 613
121.3 Configuration Options . . . 613
121.4 Available Methods . . . 614

122Translating **615**
122.1 Adding translations . . . 615
122.2 Supported formats . . . 616
122.3 Setting a locale . . . 616
122.4 Translating messages . . . 616
122.5 Caching . . . 616

123I18n View Helpers **617**
123.1 Introduction . . . 617
123.2 CurrencyFormat Helper . . . 617
123.3 DateFormat Helper . . . 618
123.4 NumberFormat Helper . . . 619
123.5 Plural Helper . . . 620
123.6 Translate Helper . . . 621
123.7 TranslatePlural Helper . . . 622
123.8 Abstract Translator Helper . . . 622

124I18n Filters **625**
124.1 Alnum . . . 625
124.2 Alpha . . . 626
124.3 NumberFormat . . . 626

125I18n Validators **629**

126Float **631**
126.1 Supported options for Zend\I18n\Validator\Float . . . 631
126.2 Simple float validation . . . 631
126.3 Localized float validation . . . 631

127Int **633**
127.1 Supported options for Zend\I18n\Validator\Int . . . 633
127.2 Simple integer validation . . . 633
127.3 Localized integer validation . . . 633

128Introduction to Zend\InputFilter **635**

129File Upload Input **639**

130Introduction to Zend\Json **641**

131Basic Usage **643**
131.1 Pretty-printing JSON . . . 643

132Advanced Usage of Zend\Json **645**
132.1 JSON Objects . . . 645
132.2 Encoding PHP objects . . . 645
132.3 Internal Encoder/Decoder . . . 646
132.4 JSON Expressions . . . 646

133XML to JSON conversion **647**

134Zend\Json\Server - JSON-RPC server **649**
134.1 Advanced Details . . . 651

135Introduction to Zend\Ldap **657**
135.1 Theory of operation . . . 657

136API overview **661**
136.1 Configuration / options . . . 661
136.2 API Reference . . . 662

137Zend\Ldap\Ldap **663**
137.1 Zend\Ldap\Collection . . . 664

138Zend\Ldap\Attribute **665**

139Zend\Ldap\Converter\Converter **667**

140Zend\Ldap\Dn **669**

141Zend\Ldap\Filter **671**

142Zend\Ldap\Node **673**

143Zend\Ldap\Node\RootDse **675**
143.1 OpenLDAP . . . 677
143.2 ActiveDirectory . . . 677
143.3 eDirectory . . . 678

144Zend\Ldap\Node\Schema **681**
144.1 OpenLDAP . . . 683
144.2 ActiveDirectory . . . 684

145Zend\Ldap\Ldif\Encoder **685**

146Usage Scenarios **687**
146.1 Authentication scenarios . . . 687
146.2 Basic CRUD operations . . . 687
146.3 Extended operations . . . 689

147Tools **691**
147.1 Creation and modification of DN strings . . . 691
147.2 Using the filter API to create search filters . . . 691
147.3 Modify LDAP entries using the Attribute API . . . 691

148 Object-oriented access to the LDAP tree using Zend\Ldap\Node **693**
148.1 Basic CRUD operations . . . 693
148.2 Extended operations . . . 693
148.3 Tree traversal . . . 693

149 Getting information from the LDAP server **695**
149.1 RootDSE . . . 695
149.2 Schema Browsing . . . 695

150 Serializing LDAP data to and from LDIF **697**
150.1 Serialize a LDAP entry to LDIF . . . 697
150.2 Deserialize a LDIF string into a LDAP entry . . . 698

151 The AutoloaderFactory **701**
151.1 Overview . . . 701
151.2 Quick Start . . . 701
151.3 Configuration Options . . . 702
151.4 Available Methods . . . 702
151.5 Examples . . . 702

152 The StandardAutoloader **703**
152.1 Overview . . . 703
152.2 Quick Start . . . 704
152.3 Configuration Options . . . 705
152.4 Available Methods . . . 705
152.5 Examples . . . 706

153 The ClassMapAutoloader **707**
153.1 Overview . . . 707
153.2 Quick Start . . . 707
153.3 Configuration Options . . . 707
153.4 Available Methods . . . 708
153.5 Examples . . . 709

154 The ModuleAutoloader **711**
154.1 Overview . . . 711
154.2 Quickstart . . . 711
154.3 Configuration Options . . . 711
154.4 Available Methods . . . 712
154.5 Examples . . . 712

155 The SplAutoloader Interface **713**
155.1 Overview . . . 713
155.2 Quick Start . . . 713
155.3 Configuration Options . . . 714
155.4 Available Methods . . . 714
155.5 Examples . . . 715

156 The PluginClassLoader **717**
156.1 Overview . . . 717
156.2 Quick Start . . . 717
156.3 Configuration Options . . . 718
156.4 Available Methods . . . 718
156.5 Examples . . . 719

157 The ShortNameLocator Interface **723**
157.1 Overview . . . 723
157.2 Quick Start . . . 723
157.3 Configuration Options . . . 723
157.4 Available Methods . . . 724
157.5 Examples . . . 724

158 The PluginClassLocator interface **725**
158.1 Overview . . . 725
158.2 Quick Start . . . 725
158.3 Configuration Options . . . 725
158.4 Available Methods . . . 725
158.5 Examples . . . 726

159 The Class Map Generator utility: bin/classmap_generator.php **727**
159.1 Overview . . . 727
159.2 Quick Start . . . 727
159.3 Configuration Options . . . 727

160 Overview of Zend\Log **729**
160.1 Creating a Log . . . 729
160.2 Logging Messages . . . 729
160.3 Destroying a Log . . . 730
160.4 Using Built-in Priorities . . . 730
160.5 Understanding Log Events . . . 730
160.6 Log PHP Errors . . . 731

161 Writers **733**
161.1 Writing to Streams . . . 733
161.2 Writing to Databases . . . 734
161.3 Writing to FirePHP . . . 734
161.4 Stubbing Out the Writer . . . 735
161.5 Testing with the Mock . . . 735
161.6 Compositing Writers . . . 735

162 Filters **737**
162.1 Available filters . . . 737

163 Formatters **739**
163.1 Simple Formatting . . . 739
163.2 Formatting to XML . . . 739
163.3 Formatting to FirePhp . . . 740

164 Introduction to Zend\Mail **741**
164.1 Getting started . . . 741
164.2 Configuring the default sendmail transport . . . 742

165 Zend\Mail\Message **743**
165.1 Overview . . . 743
165.2 Quick Start . . . 743
165.3 Configuration Options . . . 745
165.4 Available Methods . . . 745
165.5 Examples . . . 748

166 Zend\Mail\Transport **749**

166.1 Overview . . . 749
166.2 Quick Start . . . 749
166.3 Configuration Options . . . 750
166.4 Available Methods . . . 750
166.5 Examples . . . 751

167 Zend\Mail\Transport\SmtpOptions **753**
167.1 Overview . . . 753
167.2 Quick Start . . . 753
167.3 Configuration Options . . . 755
167.4 Available Methods . . . 755
167.5 Examples . . . 756

168 Zend\Mail\Transport\FileOptions **757**
168.1 Overview . . . 757
168.2 Quick Start . . . 757
168.3 Configuration Options . . . 757
168.4 Available Methods . . . 758
168.5 Examples . . . 758

169 Introduction to Zend\Math **759**
169.1 Random number generator . . . 759
169.2 Big integers . . . 760

170 Zend\Mime **763**
170.1 Introduction . . . 763
170.2 Static Methods and Constants . . . 763
170.3 Instantiating Zend\Mime . . . 764

171 Zend\Mime\Message **765**
171.1 Introduction . . . 765
171.2 Instantiation . . . 765
171.3 Adding MIME Parts . . . 765
171.4 Boundary handling . . . 765
171.5 Parsing a string to create a Zend\Mime\Message object . . . 766
171.6 Available methods . . . 766

172 Zend\Mime\Part **767**
172.1 Introduction . . . 767
172.2 Instantiation . . . 767
172.3 Methods for rendering the message part to a string . . . 767
172.4 Available methods . . . 768

173 Introduction to the Module System **769**
173.1 The autoload_*.php Files . . . 770

174 The Module Manager **771**
174.1 Module Manager Events . . . 771
174.2 Module Manager Listeners . . . 771

175 The Module Class **775**
175.1 A Minimal Module . . . 775
175.2 A Typical Module Class . . . 775
175.3 The “loadModules.post” Event . . . 776
175.4 The MVC “bootstrap” Event . . . 777

176 The Module Autoloader **779**
176.1 Module Autoloader Usage . . . 779
176.2 Non-Standard / Explicit Module Paths . . . 780
176.3 Packaging Modules with Phar . . . 781

177 Best Practices when Creating Modules **783**

178 Introduction to the MVC Layer **785**
178.1 Basic Application Structure . . . 785
178.2 Basic Module Structure . . . 786
178.3 Bootstrapping an Application . . . 788
178.4 Bootstrapping a Modular Application . . . 790
178.5 Conclusion . . . 791

179 Quick Start **793**
179.1 Install the Zend Skeleton Application . . . 793
179.2 Create a New Module . . . 794
179.3 Update the Module Class . . . 794
179.4 Create a Controller . . . 795
179.5 Create a View Script . . . 796
179.6 Create a Route . . . 796
179.7 Tell the Application About our Module . . . 798
179.8 Test it Out! . . . 798

180 Default Services **801**
180.1 Theory of Operation . . . 801
180.2 ServiceManager . . . 801
180.3 ViewManager . . . 805
180.4 Application Configuration Options . . . 806
180.5 Default Configuration Options . . . 807

181 Routing **811**
181.1 Router Types . . . 813
181.2 HTTP Route Types . . . 813
181.3 HTTP Routing Examples . . . 819
181.4 Console Route Types . . . 821

182 The MvcEvent **823**
182.1 Order of events . . . 824
182.2 MvcEvent::EVENT_BOOTSTRAP . . . 824
182.3 MvcEvent::EVENT_ROUTE . . . 825
182.4 MvcEvent::EVENT_DISPATCH . . . 825
182.5 MvcEvent::EVENT_DISPATCH_ERROR . . . 827
182.6 MvcEvent::EVENT_RENDER . . . 828
182.7 MvcEvent::EVENT_RENDER_ERROR . . . 829
182.8 MvcEvent::EVENT_FINISH . . . 830

183 The SendResponseEvent **831**
183.1 Listeners . . . 831
183.2 Triggerers . . . 831

184 Available Controllers **833**
184.1 Common Interfaces Used With Controllers . . . 833
184.2 The AbstractActionController . . . 835
184.3 The AbstractRestfulController . . . 836

185 Controller Plugins **839**
185.1 AcceptableViewModelSelector Plugin . . . 839
185.2 FlashMessenger Plugin . . . 840
185.3 Forward Plugin . . . 842
185.4 Identity Plugin . . . 842
185.5 Layout Plugin . . . 843
185.6 Params Plugin . . . 843
185.7 Post/Redirect/Get Plugin . . . 844
185.8 File Post/Redirect/Get Plugin . . . 844
185.9 Redirect Plugin . . . 846
185.10 Url Plugin . . . 846

186 Examples **847**
186.1 Controllers . . . 847
186.2 Bootstrapping . . . 848

187 Introduction to Zend\Navigation **849**
187.1 Pages and Containers . . . 849
187.2 View Helpers . . . 849

188 Quick Start **851**

189 Pages **853**

190 Common page features **855**

191 Zend\Navigation\Page\Mvc **859**

192 Zend\Navigation\Page\Uri **863**

193 Creating custom page types **865**

194 Creating pages using the page factory **867**

195 Containers **869**
195.1 Creating containers . . . 869
195.2 Adding pages . . . 875
195.3 Removing pages . . . 876
195.4 Finding pages . . . 877
195.5 Iterating containers . . . 879
195.6 Other operations . . . 880

196 View Helpers **883**
196.1 Translation of labels and titles . . . 884
196.2 Integration with ACL . . . 884
196.3 Navigation setup used in examples . . . 885

197 View Helper - Breadcrumbs **891**

198 View Helper - Links **895**

199 View Helper - Menu **899**

200 View Helper - Sitemap **907**

201 View Helper - Navigation Proxy **913**

202 Introduction to Zend\Paginator **915**

203 Usage **917**
203.1 Paginating data collections . . . 917
203.2 The DbSelect adapter . . . 918
203.3 Rendering pages with view scripts . . . 919

204 Configuration **925**

205 Advanced usage **927**
205.1 Custom data source adapters . . . 927
205.2 Custom scrolling styles . . . 927
205.3 Caching features . . . 928

206 Introduction to Zend\Permissions\Acl **931**
206.1 Resources . . . 931
206.2 Roles . . . 932
206.3 Creating the Access Control List . . . 933
206.4 Registering Roles . . . 933
206.5 Defining Access Controls . . . 934
206.6 Querying an ACL . . . 935

207 Refining Access Controls **937**
207.1 Precise Access Controls . . . 937
207.2 Removing Access Controls . . . 938

208 Advanced Usage **941**
208.1 Storing ACL Data for Persistence . . . 941
208.2 Writing Conditional ACL Rules with Assertions . . . 941

209 Introduction to Zend\Permissions\Rbac **943**
209.1 Roles . . . 943
209.2 Permissions . . . 943
209.3 Dynamic Assertions . . . 943

210 Methods **945**

211 Examples **947**
211.1 Roles . . . 947
211.2 Permissions . . . 948
211.3 Dynamic Assertions . . . 948

212 Progress Bars **951**
212.1 Introduction . . . 951
212.2 Basic Usage . . . 951
212.3 Persistent Progress . . . 951
212.4 Standard Adapters . . . 952

213 File Upload Handlers **955**
213.1 Introduction . . . 955
213.2 Methods of Reporting Progress . . . 955
213.3 Standard Handlers . . . 956

214 Introduction to Zend\Serializer **959**
214.1 Quick Start . . . 959
214.2 Basic configuration Options . . . 960

214.3 Available Methods . . . 960

215 Zend\Serializer\Adapter **963**
215.1 The PhpSerialize Adapter . . . 963
215.2 The IgBinary Adapter . . . 963
215.3 The Wddx Adapter . . . 963
215.4 The Json Adapter . . . 964
215.5 The PythonPickle Adapter . . . 964
215.6 The PhpCode Adapter . . . 965

216 Introduction to Zend\Server **967**

217 Zend\Server\Reflection **969**
217.1 Introduction . . . 969
217.2 Usage . . . 969

218 Zend\ServiceManager **971**

219 Zend\ServiceManager Quick Start **975**
219.1 Using Configuration . . . 975
219.2 Modules as Service Providers . . . 976
219.3 Examples . . . 976

220 Session Config **981**

221 Standard Config **983**
221.1 Basic Configuration Options . . . 983

222 Basic Usage **985**

223 Session Config **987**
223.1 Basic Configuration Options . . . 987

224 Basic Usage **989**
224.1 Custom Configuration . . . 989

225 Session Container **991**
225.1 Basic Usage . . . 991
225.2 Setting the Default Session Manager . . . 991

226 Session Manager **993**
226.1 Initializing the Session Manager . . . 993
226.2 Session Compatibility . . . 995

227 Session Save Handlers **997**

228 Cache **999**
228.1 Basic usage . . . 999

229 DbTableGateway **1001**
229.1 Creating the database table . . . 1001
229.2 Basic usage . . . 1001

230 MongoDB **1003**
230.1 Basic Usage . . . 1003
230.2 Custom Save Handlers . . . 1003

231 Session Storage **1005**

232 Array Storage **1007**
232.1 Basic Usage . . . 1007

233 Session Storage **1009**
233.1 Basic Usage . . . 1009

234 Session Array Storage **1011**
234.1 Basic Usage . . . 1011
234.2 Custom Storage . . . 1011

235 Session Validators **1013**

236 Http User Agent **1015**
236.1 Basic Usage . . . 1015

237 Remote Addr **1017**
237.1 Basic Usage . . . 1017
237.2 Custom Validators . . . 1017

238 Zend\Soap\Server **1019**
238.1 Zend\Soap\Server constructor . . . 1019
238.2 Methods to define Web Service API . . . 1020
238.3 Request and response objects handling . . . 1021
238.4 Document/Literal WSDL Handling . . . 1023

239 Zend\Soap\Client **1025**
239.1 Zend\Soap\Client Constructor . . . 1025
239.2 Performing SOAP Requests . . . 1026

240 WSDL Accessor **1029**
240.1 Zend\Soap\Wsdl constructor . . . 1029
240.2 addMessage() method . . . 1029
240.3 addPortType() method . . . 1030
240.4 addPortOperation() method . . . 1030
240.5 addBinding() method . . . 1030
240.6 addBindingOperation() method . . . 1030
240.7 addSoapBinding() method . . . 1031
240.8 addSoapOperation() method . . . 1031
240.9 addService() method . . . 1031
240.10 Type mapping . . . 1032
240.11 addDocumentation() method . . . 1033
240.12 Get finalized WSDL document . . . 1033

241 AutoDiscovery **1035**
241.1 AutoDiscovery Introduction . . . 1035
241.2 Class autodiscovering . . . 1036
241.3 Functions autodiscovering . . . 1037
241.4 Autodiscovering Datatypes . . . 1037
241.5 WSDL Binding Styles . . . 1037

242 Zend\Stdlib\Hydrator **1039**
242.1 HydratorInterface . . . 1039
242.2 Usage . . . 1039
242.3 Available Implementations . . . 1040

243 Zend\Stdlib\Hydrator\Filter **1041**
243.1 Filter implementations . . . 1041
243.2 Remove filters . . . 1042
243.3 Add filters . . . 1042
243.4 Use the composite for complex filters . . . 1043
243.5 Using the provider interface . . . 1044

244 Zend\Stdlib\Hydrator\Strategy **1047**
244.1 Adding strategies to the hydrators . . . 1047
244.2 Available implementations . . . 1048
244.3 Writing custom strategies . . . 1048

245 Introduction to Zend\Tag **1051**

246 Creating tag clouds with Zend\Tag\Cloud **1053**
246.1 Decorators . . . 1054

247 Introduction to Zend\Test **1057**

248 Unit testing with PHPUnit **1059**

249 Setup your TestCase **1061**

250 Testing your Controllers and MVC Applications **1063**

251 Assertions **1065**

252 Request Assertions **1067**

253 CSS Selector Assertions **1069**

254 XPath Assertions **1071**

255 Redirect Assertions **1073**

256 Response Header Assertions **1075**

257 Zend\Text\Figlet **1077**

258 Zend\Text\Table **1079**

259 Zend\Uri **1081**
259.1 Overview . . . 1081
259.2 Creating a New URI . . . 1081
259.3 Manipulating an Existing URI . . . 1082
259.4 Common Instance Methods . . . 1082

260 Introduction to Zend\Validator **1087**
260.1 What is a validator? . . . 1087
260.2 Basic usage of validators . . . 1087
260.3 Customizing messages . . . 1088
260.4 Translating messages . . . 1089

261 Standard Validation Classes **1091**

262 Alnum **1093**
262.1 Supported options for Zend\Validator\Alnum . . . 1093

262.2 Basic usage . . . 1093
262.3 Using whitespaces . . . 1093
262.4 Using different languages . . . 1094

263Alpha **1095**
263.1 Supported options for Zend\Validator\Alpha . . . 1095
263.2 Basic usage . . . 1095
263.3 Using whitespaces . . . 1095
263.4 Using different languages . . . 1096

264Barcode **1097**
264.1 Supported options for Zend\Validator\Barcode . . . 1099
264.2 Basic usage . . . 1100
264.3 Optional checksum . . . 1100
264.4 Writing custom adapters . . . 1100

265Between **1103**
265.1 Supported options for Zend\Validator\Between . . . 1103
265.2 Default behaviour for Zend\Validator\Between . . . 1103
265.3 Validation exclusive the border values . . . 1103

266Callback **1105**
266.1 Supported options for Zend\Validator\Callback . . . 1105
266.2 Basic usage . . . 1105
266.3 Usage with closures . . . 1105
266.4 Usage with class-based callbacks . . . 1106
266.5 Adding options . . . 1107

267CreditCard **1109**
267.1 Supported options for Zend\Validator\CreditCard . . . 1110
267.2 Basic usage . . . 1110
267.3 Accepting defined credit cards . . . 1110
267.4 Validation by using foreign APIs . . . 1111
267.5 Ccnum . . . 1112

268Date **1113**
268.1 Supported options for Zend\Validator\Date . . . 1113
268.2 Default date validation . . . 1113
268.3 Localized date validation . . . 1113
268.4 Self defined date validation . . . 1114

269Db\RecordExists and Db\NoRecordExists **1115**
269.1 Supported options for Zend\Validator\Db* . . . 1115
269.2 Basic usage . . . 1115
269.3 Excluding records . . . 1116
269.4 Database Schemas . . . 1117
269.5 Using a Select object . . . 1117

270Digits **1119**
270.1 Supported options for Zend\Validator\Digits . . . 1119
270.2 Validating digits . . . 1119

271EmailAddress **1121**
271.1 Basic usage . . . 1121
271.2 Options for validating Email Addresses . . . 1121

271.3 Complex local parts . . . 1122
271.4 Validating only the local part . . . 1122
271.5 Validating different types of hostnames . . . 1122
271.6 Checking if the hostname actually accepts email . . . 1122
271.7 Validating International Domains Names . . . 1123
271.8 Validating Top Level Domains . . . 1124
271.9 Setting messages . . . 1124

272GreaterThan **1125**
272.1 Supported options for Zend\Validator\GreaterThan . . . 1125
272.2 Basic usage . . . 1125
272.3 Validation inclusive the border value . . . 1125

273Hex **1127**
273.1 Supported options for Zend\Validator\Hex . . . 1127

274Hostname **1129**
274.1 Supported options for Zend\Validator\Hostname . . . 1129
274.2 Basic usage . . . 1129
274.3 Validating different types of hostnames . . . 1129
274.4 Validating International Domains Names . . . 1130
274.5 Validating Top Level Domains . . . 1131

275Iban **1133**
275.1 Supported options for Zend\Validator\Iban . . . 1133
275.2 IBAN validation . . . 1133

276Identical **1135**
276.1 Supported options for Zend\Validator\Identical . . . 1135
276.2 Basic usage . . . 1135
276.3 Identical objects . . . 1135
276.4 Form elements . . . 1136
276.5 Strict validation . . . 1138
276.6 Configuration . . . 1138

277InArray **1139**
277.1 Supported options for Zend\Validator\InArray . . . 1139
277.2 Simple array validation . . . 1139
277.3 Array validation modes . . . 1140
277.4 Recursive array validation . . . 1141

278Ip **1143**
278.1 Supported options for Zend\Validator\Ip . . . 1143
278.2 Basic usage . . . 1143
278.3 Validate IPv4 or IPV6 alone . . . 1144

279Isbn **1145**
279.1 Supported options for Zend\Validator\Isbn . . . 1145
279.2 Basic usage . . . 1145
279.3 Setting an explicit ISBN validation type . . . 1145
279.4 Specifying a separator restriction . . . 1146

280LessThan **1147**
280.1 Supported options for Zend\Validator\LessThan . . . 1147
280.2 Basic usage . . . 1147

280.3 Validation inclusive the border value . . . 1147

281 NotEmpty **1149**
281.1 Supported options for Zend\Validator\NotEmpty . . . 1149
281.2 Default behaviour for Zend\Validator\NotEmpty . . . 1149
281.3 Changing behaviour for Zend\Validator\NotEmpty . . . 1149

282 PostCode **1151**
282.1 Constructor options . . . 1152
282.2 Supported options for Zend\Validator\PostCode . . . 1152

283 Regex **1153**
283.1 Supported options for Zend\Validator\Regex . . . 1153
283.2 Validation with Zend\Validator\Regex . . . 1153
283.3 Pattern handling . . . 1153

284 Sitemap Validators **1155**
284.1 Sitemap\Changefreq . . . 1155
284.2 Sitemap\Lastmod . . . 1155
284.3 Sitemap\Loc . . . 1155
284.4 Sitemap\Priority . . . 1156
284.5 Supported options for Zend\Validator\Sitemap_* . . . 1156

285 Step **1157**
285.1 Supported options for Zend\Validator\Step . . . 1157
285.2 Basic usage . . . 1157
285.3 Using floating-point values . . . 1157

286 StringLength **1159**
286.1 Supported options for Zend\Validator\StringLength . . . 1159
286.2 Default behaviour for Zend\Validator\StringLength . . . 1159
286.3 Limiting the maximum allowed length of a string . . . 1159
286.4 Limiting the minimal required length of a string . . . 1160
286.5 Limiting a string on both sides . . . 1160
286.6 Encoding of values . . . 1160

287 File Validation Classes **1163**
287.1 Crc32 . . . 1163
287.2 ExcludeExtension . . . 1164
287.3 ExcludeMimeType . . . 1164
287.4 Exists . . . 1164
287.5 Extension . . . 1165
287.6 Hash . . . 1166
287.7 ImageSize . . . 1166
287.8 IsCompressed . . . 1167
287.9 IsImage . . . 1168
287.10 Md5 . . . 1168
287.11 MimeType . . . 1169
287.12 NotExists . . . 1170
287.13 Sha1 . . . 1171
287.14 Size . . . 1172
287.15 UploadFile . . . 1172
287.16 WordCount . . . 1173

288 Validator Chains **1175**

289 Writing Validators **1177**

290 Validation Messages **1181**
290.1 Using pre-translated validation messages . . . 1181
290.2 Limit the size of a validation message . . . 1182

291 Getting the Zend Framework Version **1183**

292 Zend\View Quick Start **1185**

293 Overview **1187**

294 Usage **1189**
294.1 Configuration . . . 1189
294.2 Controllers and View Models . . . 1190
294.3 Nesting View Models . . . 1191
294.4 Dealing with Layouts . . . 1194
294.5 Creating and Registering Alternate Rendering and Response Strategies . . . 1197

295 The PhpRenderer **1203**
295.1 Usage . . . 1203
295.2 Options and Configuration . . . 1207
295.3 Additional Methods . . . 1207

296 PhpRenderer View Scripts **1209**
296.1 Escaping Output . . . 1210

297 The ViewEvent **1211**
297.1 Order of events . . . 1211
297.2 ViewEvent::EVENT_RENDERER . . . 1212
297.3 ViewEvent::EVENT_RENDERER_POST . . . 1213
297.4 ViewEvent::EVENT_RESPONSE . . . 1213

298 View Helpers **1215**

299 Included Helpers **1217**
299.1 BasePath Helper . . . 1217
299.2 Cycle Helper . . . 1218
299.3 Doctype Helper . . . 1219
299.4 HeadLink Helper . . . 1220
299.5 HeadMeta Helper . . . 1221
299.6 HeadScript Helper . . . 1223
299.7 HeadStyle Helper . . . 1226
299.8 HeadTitle Helper . . . 1229
299.9 HtmlList Helper . . . 1231
299.10 HTML Object Helpers . . . 1233
299.11 Identity Helper . . . 1234
299.12 InlineScript Helper . . . 1235
299.13 JSON Helper . . . 1235
299.14 Partial Helper . . . 1236
299.15 Placeholder Helper . . . 1238
299.16 URL Helper . . . 1240

300 Advanced usage of helpers **1243**
300.1 Registering Helpers . . . 1243
300.2 Writing Custom Helpers . . . 1244

300.3 Registering Concrete Helpers . . . 1245

301 Introduction to Zend\XmlRpc **1247**
301.1 Quick Start . . . 1247

302 Zend\XmlRpc\Client **1249**
302.1 Introduction . . . 1249
302.2 Method Calls . . . 1249
302.3 Types and Conversions . . . 1250
302.4 Server Proxy Object . . . 1251
302.5 Error Handling . . . 1252
302.6 Server Introspection . . . 1253
302.7 From Request to Response . . . 1254
302.8 HTTP Client and Testing . . . 1254

303 Zend\XmlRpc\Server **1255**
303.1 Introduction . . . 1255
303.2 Basic Usage . . . 1255
303.3 Server Structure . . . 1255
303.4 Anatomy of a webservice . . . 1256
303.5 Conventions . . . 1256
303.6 Utilizing Namespaces . . . 1257
303.7 Custom Request Objects . . . 1257
303.8 Custom Responses . . . 1258
303.9 Handling Exceptions via Faults . . . 1258
303.10 Caching Server Definitions Between Requests . . . 1258
303.11 Usage Examples . . . 1259
303.12 Performance optimization . . . 1263

304 ZendServiceAkismet **1265**
304.1 Introduction . . . 1265
304.2 Verify an API key . . . 1265
304.3 Check for spam . . . 1265
304.4 Submitting known spam . . . 1266
304.5 Submitting false positives (ham) . . . 1267
304.6 Zend-specific Methods . . . 1267

305 ZendServiceAmazon **1269**
305.1 Introduction . . . 1269
305.2 Country Codes . . . 1270
305.3 Looking up a Specific Amazon Item by ASIN . . . 1270
305.4 Performing Amazon Item Searches . . . 1271
305.5 Using the Alternative Query API . . . 1271
305.6 ZendServiceAmazon Classes . . . 1272

306 `ZendService\Apple\Apns` **1277**
306.1 Introduction . . . 1277
306.2 Quick Start . . . 1277
306.3 Feedback Service . . . 1279

307 ZendServiceAudioscrobbler **1281**
307.1 Introduction . . . 1281
307.2 Users . . . 1281
307.3 Artists . . . 1283
307.4 Tracks . . . 1283

307.5 Tags . . . 1284
307.6 Groups . . . 1284
307.7 Forums . . . 1284

308ZendServiceDelicious **1287**
308.1 Introduction . . . 1287
308.2 Retrieving posts . . . 1287
308.3 ZendServiceDeliciousPostList . . . 1288
308.4 Editing posts . . . 1289
308.5 Deleting posts . . . 1289
308.6 Adding new posts . . . 1290
308.7 Tags . . . 1290
308.8 Bundles . . . 1291
308.9 Public data . . . 1291
308.10HTTP client . . . 1292

309Zend_Service_DeveloperGarden **1293**
309.1 Introduction to DeveloperGarden . . . 1293
309.2 BaseUserService . . . 1294
309.3 IP Location . . . 1295
309.4 Local Search . . . 1296
309.5 Send SMS . . . 1296
309.6 SMS Validation . . . 1297
309.7 Voice Call . . . 1297
309.8 ConferenceCall . . . 1298
309.9 Performance and Caching . . . 1300

310ZendServiceFlickr **1301**
310.1 Introduction . . . 1301
310.2 Finding Flickr Users' Photos and Information . . . 1301
310.3 Finding photos From a Group Pool . . . 1302
310.4 Retrieving Flickr Image Details . . . 1302
310.5 ZendServiceFlickr Result Classes . . . 1302

311`ZendService\LiveDocx` **1305**
311.1 Introduction to LiveDocx . . . 1305
311.2 `ZendService\LiveDocx\MailMerge` . . . 1307

312ZendServiceNirvanix **1319**
312.1 Introduction . . . 1319
312.2 Registering with Nirvanix . . . 1319
312.3 API Documentation . . . 1319
312.4 Features . . . 1319
312.5 Getting Started . . . 1320
312.6 Understanding the Proxy . . . 1320
312.7 Examining Results . . . 1321
312.8 Handling Errors . . . 1322

313Zend\Service\Rackspace **1323**
313.1 Introduction . . . 1323
313.2 Registering with Rackspace . . . 1323
313.3 Cloud Files . . . 1323
313.4 Cloud Servers . . . 1324
313.5 Available Methods . . . 1324

314 ZendServiceReCaptcha **1327**
314.1 Introduction . . . 1327
314.2 Simplest use . . . 1327
314.3 Hiding email addresses . . . 1328

315 ZendServiceSlideShare **1331**
315.1 Getting Started with ZendServiceSlideShare . . . 1331
315.2 The SlideShow object . . . 1331
315.3 Retrieving a single slide show . . . 1334
315.4 Retrieving Groups of Slide Shows . . . 1334
315.5 ZendServiceSlideShare Caching policies . . . 1335
315.6 Changing the behavior of the HTTP Client . . . 1335

316 ZendServiceStrikeIron **1337**
316.1 Overview . . . 1337
316.2 Registering with StrikeIron . . . 1337
316.3 Getting Started . . . 1338
316.4 Making Your First Query . . . 1338
316.5 Examining Results . . . 1339
316.6 Handling Errors . . . 1340
316.7 Checking Your Subscription . . . 1340

317 ZendServiceTechnorati **1343**
317.1 Introduction . . . 1343
317.2 Getting Started . . . 1343
317.3 Making Your First Query . . . 1343
317.4 Consuming Results . . . 1344
317.5 Handling Errors . . . 1346
317.6 Checking Your API Key Daily Usage . . . 1346
317.7 Available Technorati Queries . . . 1346
317.8 ZendServiceTechnorati Classes . . . 1350

318 ZendServiceTwitter **1355**
318.1 Introduction . . . 1355
318.2 Authentication . . . 1355
318.3 Account Methods . . . 1357
318.4 Application Methods . . . 1357
318.5 Blocking Methods . . . 1358
318.6 Direct Message Methods . . . 1358
318.7 Favorites Methods . . . 1359
318.8 Friendship Methods . . . 1360
318.9 Search Methods . . . 1360
318.10 Status Methods . . . 1361
318.11 User Methods . . . 1363

319 ZendServiceWindowsAzure **1365**
319.1 Introduction . . . 1365
319.2 Installing the Windows Azure SDK . . . 1365
319.3 API Documentation . . . 1365
319.4 Features . . . 1365
319.5 Architecture . . . 1366

320 ZendServiceWindowsAzureStorageBlob **1367**
320.1 API Examples . . . 1367
320.2 Root container . . . 1369

320.3 Blob storage stream wrapper . . . 1369
320.4 Shared Access Signature . . . 1370

321 Zend\Service\WindowsAzure\Storage\Table 1373
321.1 Operations on tables . . . 1373
321.2 Operations on entities . . . 1374
321.3 Table storage session handler . . . 1380

322 Zend\Service\WindowsAzure\Storage\Queue 1383
322.1 API Examples . . . 1383

323 Copyright Information 1387

324 Introduction to Zend Framework 2 1389

325 User Guide 1391

326 Zend Framework Tool (ZFTool) 1393

327 Learning Zend Framework 2 1395

328 Migration 1397

329 Zend Framework 2 Reference 1399
329.1 Zend\Authentication . . . 1399
329.2 Zend\Barcode . . . 1399
329.3 Zend\Cache . . . 1399
329.4 Zend\Captcha . . . 1400
329.5 Zend\Config . . . 1400
329.6 Zend\Console . . . 1400
329.7 Zend\Crypt . . . 1400
329.8 Zend\Db . . . 1400
329.9 Zend\Di . . . 1401
329.10 Zend\Dom . . . 1401
329.11 Zend\Escaper . . . 1401
329.12 Zend\EventManager . . . 1401
329.13 Zend\Feed . . . 1401
329.14 Zend\File . . . 1402
329.15 Zend\Filter . . . 1402
329.16 Zend\Form . . . 1402
329.17 Zend\Http . . . 1402
329.18 Zend\I18n . . . 1403
329.19 Zend\InputFilter . . . 1403
329.20 Zend\Json . . . 1403
329.21 Zend\Ldap . . . 1403
329.22 Zend\Loader . . . 1404
329.23 Zend\Log . . . 1404
329.24 Zend\Mail . . . 1404
329.25 Zend\Math . . . 1404
329.26 Zend\Mime . . . 1404
329.27 Zend\ModuleManager . . . 1405
329.28 Zend\Mvc . . . 1405
329.29 Zend\Navigation . . . 1405
329.30 Zend\Paginator . . . 1405
329.31 Zend\Permissions\Acl . . . 1406

329.32 Zend\Permissions\Rbac . . . 1406
329.33 Zend\ProgressBar . . . 1406
329.34 Zend\Serializer . . . 1406
329.35 Zend\Server . . . 1406
329.36 Zend\ServiceManager . . . 1406
329.37 Zend\Session . . . 1407
329.38 Zend\Soap . . . 1407
329.39 Zend\Stdlib . . . 1407
329.40 Zend\Tag . . . 1407
329.41 Zend\Test . . . 1407
329.42 Zend\Text . . . 1407
329.43 Zend\Uri . . . 1408
329.44 Zend\Validator . . . 1408
329.45 Zend\Version . . . 1408
329.46 Zend\View . . . 1408
329.47 Zend\XmlRpc . . . 1408

330 Services for Zend Framework 2 Reference **1409**
330.1 ZendService\Akismet . . . 1409
330.2 ZendService\Amazon . . . 1409
330.3 ZendService\AppleApns . . . 1409
330.4 ZendService\Audioscrobbler . . . 1409
330.5 ZendService\Del.icio.us . . . 1409
330.6 ZendService\Developer Garden . . . 1409
330.7 ZendService\Flickr . . . 1409
330.8 ZendService\Google\Gcm . . . 1410
330.9 ZendService\LiveDocx . . . 1410
330.10 ZendService\Nirvanix . . . 1410
330.11 ZendService\Rackspace . . . 1410
330.12 ZendService\ReCaptcha . . . 1410
330.13 ZendService\SlideShare . . . 1410
330.14 ZendService\StrikeIron . . . 1410
330.15 ZendService\Technorati . . . 1410
330.16 ZendService\Twitter . . . 1410
330.17 ZendService\Windows Azure . . . 1410

331 Copyright **1411**

332 Indices and tables **1413**

CHAPTER

ONE

OVERVIEW

Zend Framework 2 is an open source framework for developing web applications and services using *PHP* 5.3+. Zend Framework 2 uses 100% object-oriented code and utilises most of the new features of PHP 5.3, namely namespaces, late static binding, lambda functions and closures.

Zend Framework 2 evolved from Zend Framework 1, a successful PHP framework with over 15 million downloads.

Note: *ZF2* is not backward compatible with *ZF1*, because of the new features in PHP 5.3+ implemented by the framework, and due to major rewrites of many components.

The component structure of Zend Framework 2 is unique; each component is designed with few dependencies on other components. ZF2 follows the SOLID object-oriented design principle. This loosely coupled architecture allows developers to use whichever components they want. We call this a "use-at-will" design. We support Pyrus and Composer as installation and dependency tracking mechanisms for the framework as a whole and for each component, further enhancing this design.

We use PHPUnit to test our code and Travis CI as a Continuous Integration service.

While they can be used separately, Zend Framework 2 components in the standard library form a powerful and extensible web application framework when combined. Also, it offers a robust, high performance MVC implementation, a database abstraction that is simple to use, and a forms component that implements HTML5 form rendering, validation, and filtering so that developers can consolidate all of these operations using one easy-to-use, object oriented interface. Other components, such as `Zend\Authentication` and `Zend\Permissions\Acl`, provide user authentication and authorization against all common credential stores.

Still others, with the `ZendService` namespace, implement client libraries to simply access the most popular web services available. Whatever your application needs are, you're likely to find a Zend Framework 2 component that can be used to dramatically reduce development time with a thoroughly tested foundation.

The principal sponsor of the project 'Zend Framework 2' is Zend Technologies, but many companies have contributed components or significant features to the framework. Companies such as Google, Microsoft, and StrikeIron have partnered with Zend to provide interfaces to web services and other technologies they wish to make available to Zend Framework 2 developers.

Zend Framework 2 could not deliver and support all of these features without the help of the vibrant Zend Framework 2 community. Community members, including contributors, make themselves available on mailing lists, IRC channels and other forums. Whatever question you have about Zend Framework 2, the community is always available to address it.

CHAPTER

TWO

INSTALLATION

- **New to Zend Framework?** Download the latest stable release. Available in `.zip` and `.tar.gz` formats.
- **Brave, cutting edge?** Download Zend Framework's Git repository using a Git client. Zend Framework is open source software, and the Git repository used for its development is publicly available on GitHub. Consider using Git to get Zend Framework if you want to contribute back to the framework, or need to upgrade your framework version more often than releases occur.

Once you have a copy of Zend Framework available, your application needs to be able to access the framework classes found in the library folder. There are several ways to achieve this.

Failing to find a Zend Framework 2 installation, the following error occurs:

```
Fatal error: Uncaught exception 'RuntimeException' with message
'Unable to load ZF2. Run `php composer.phar install` or define
a ZF2_PATH environment variable.'
```

To fix that, you can add the Zend Framework's library path to the *PHP* include_path. Also, you should set an environment path named 'ZF2_PATH' in httpd.conf (or equivalent). i.e. `SetEnv ZF2_PATH /var/ZF2` running Linux.

Rob Allen has kindly provided the community with an introductory tutorial, Getting Started with Zend Framework 2. Other Zend Framework community members are actively working on expanding the tutorial.

CHAPTER

THREE

GETTING STARTED WITH ZEND FRAMEWORK 2

This tutorial is intended to give an introduction to using Zend Framework 2 by creating a simple database driven application using the Model-View-Controller paradigm. By the end you will have a working ZF2 application and you can then poke around the code to find out more about how it all works and fits together.

3.1 Some assumptions

This tutorial assumes that you are running at least PHP 5.3.3 with the Apache web server and MySQL, accessible via the PDO extension. Your Apache installation must have the mod_rewrite extension installed and configured.

You must also ensure that Apache is configured to support `.htaccess` files. This is usually done by changing the setting:

```
AllowOverride None
```

to

```
AllowOverride FileInfo
```

in your `httpd.conf` file. Check with your distribution's documentation for exact details. You will not be able to navigate to any page other than the home page in this tutorial if you have not configured mod_rewrite and .htaccess usage correctly.

3.2 The tutorial application

The application that we are going to build is a simple inventory system to display which albums we own. The main page will list our collection and allow us to add, edit and delete CDs. We are going to need four pages in our website:

Page	Description
List of albums	This will display the list of albums and provide links to edit and delete them. Also, a link to enable adding new albums will be provided.
Add new album	This page will provide a form for adding a new album.
Edit album	This page will provide a form for editing an album.
Delete album	This page will confirm that we want to delete an album and then delete it.

We will also need to store our data into a database. We will only need one table with these fields in it:

Field name	Type	Null?	Notes
id	integer	No	Primary key, auto-increment
artist	varchar(100)	No	
title	varchar(100)	No	

CHAPTER

FOUR

GETTING STARTED: A SKELETON APPLICATION

In order to build our application, we will start with the ZendSkeletonApplication available on github. Use Composer (http://getcomposer.org) to create a new project from scratch with Zend Framework:

```
php composer.phar create-project --repository-url="http://packages.zendframework.com" zendframework/s
```

Note: Another way to install the ZendSkeletonApplication is to use github. Go to https://github.com/zendframework/ZendSkeletonApplication and click the "Zip" button. This will download a file with a name like `ZendSkeletonApplication-master.zip` or similar.

Unzip this file into the directory where you keep all your vhosts and rename the resultant directory to `zf2-tutorial`.

ZendSkeletonApplication is set up to use Composer (http://getcomposer.org) to resolve its dependencies. In this case, the dependency is Zend Framework 2 itself.

To install Zend Framework 2 into our application we simply type:

```
php composer.phar self-update
php composer.phar install
```

from the `zf2-tutorial` folder. This takes a while. You should see an output like:

```
Installing dependencies from lock file
- Installing zendframework/zendframework (dev-master)
  Cloning 18c8e223f070deb07c17543ed938b54542aa0ed8

Generating autoload files
```

Note: If you see this message:

```
[RuntimeException]
  The process timed out.
```

then your connection was too slow to download the entire package in time, and composer timed out. To avoid this, instead of running:

```
php composer.phar install
```

run instead:

```
COMPOSER_PROCESS_TIMEOUT=5000 php composer.phar install
```

We can now move on to the virtual host.

4.1 Virtual host

You now need to create an Apache virtual host for the application and edit your hosts file so that http://zf2-tutorial.localhost will serve `index.php` from the `zf2-tutorial/public` directory.

Setting up the virtual host is usually done within `httpd.conf` or `extra/httpd-vhosts.conf`. If you are using `httpd-vhosts.conf`, ensure that this file is included by your main `httpd.conf` file. Some Linux distributions (ex: Ubuntu) package Apache so that configuration files are stored in `/etc/apache2` and create one file per virtual host inside folder `/etc/apache2/sites-enabled`. In this case, you would place the virtual host block below into the file `/etc/apache2/sites-enabled/zf2-tutorial`.

Ensure that `NameVirtualHost` is defined and set to "*:80" or similar, and then define a virtual host along these lines:

```
<VirtualHost *:80>
    ServerName zf2-tutorial.localhost
    DocumentRoot /path/to/zf2-tutorial/public
    SetEnv APPLICATION_ENV "development"
    <Directory /path/to/zf2-tutorial/public>
        DirectoryIndex index.php
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Make sure that you update your `/etc/hosts` or `c:\windows\system32\drivers\etc\hosts` file so that `zf2-tutorial.localhost` is mapped to `127.0.0.1`. The website can then be accessed using http://zf2-tutorial.localhost.

```
127.0.0.1               zf2-tutorial.localhost localhost
```

Restart your web server. If you've done it right, you should see something like this:

To test that your `.htaccess` file is working, navigate to http://zf2-tutorial.localhost/1234 and you should see this:

If you see a standard Apache 404 error, then you need to fix `.htaccess` usage before continuing. If you're are using IIS with the URL Rewrite Module, import the following:

```
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^.*$ index.php [NC,L]
```

You now have a working skeleton application and we can start adding the specifics for our application.

CHAPTER

FIVE

MODULES

Zend Framework 2 uses a module system and you organise your main application-specific code within each module. The Application module provided by the skeleton is used to provide bootstrapping, error and routing configuration to the whole application. It is usually used to provide application level controllers for, say, the home page of an application, but we are not going to use the default one provided in this tutorial as we want our album list to be the home page, which will live in our own module.

We are going to put all our code into the Album module which will contain our controllers, models, forms and views, along with configuration. We'll also tweak the Application module as required.

Let's start with the directories required.

5.1 Setting up the Album module

Start by creating a directory called `Album` under `module` with the following subdirectories to hold the module's files:

```
zf2-tutorial/
    /module
        /Album
            /config
            /src
                /Album
                    /Controller
                    /Form
                    /Model
            /view
                /album
                    /album
```

As you can see the `Album` module has separate directories for the different types of files we will have. The PHP files that contain classes within the `Album` namespace live in the `src/Album` directory so that we can have multiple namespaces within our module should we require it. The view directory also has a sub-folder called `album` for our module's view scripts.

In order to load and configure a module, Zend Framework 2 has a `ModuleManager`. This will look for `Module.php` in the root of the module directory (`module/Album`) and expect to find a class called `Album\Module` within it. That is, the classes within a given module will have the namespace of the module's name, which is the directory name of the module.

Create `Module.php` in the `Album` module: Create a file called `Module.php` under `zf2-tutorial/module/Album`:

```php
<?php
namespace Album;

class Module
{
    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\ClassMapAutoloader' => array(
                __DIR__ . '/autoload_classmap.php',
            ),
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
                ),
            ),
        );
    }

    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php';
    }
}
```

The `ModuleManager` will call `getAutoloaderConfig()` and `getConfig()` automatically for us.

5.1.1 Autoloading files

Our `getAutoloaderConfig()` method returns an array that is compatible with ZF2's `AutoloaderFactory`. We configure it so that we add a class map file to the `ClassmapAutoloader` and also add this module's namespace to the `StandardAutoloader`. The standard autoloader requires a namespace and the path where to find the files for that namespace. It is PSR-0 compliant and so classes map directly to files as per the PSR-0 rules.

As we are in development, we don't need to load files via the classmap, so we provide an empty array for the classmap autoloader. Create a file called `autoload_classmap.php` under `zf2-tutorial/module/Album`:

```php
<?php
return array();
```

As this is an empty array, whenever the autoloader looks for a class within the `Album` namespace, it will fall back to the to `StandardAutoloader` for us.

Note: If you are using Composer, you could instead just create an empty `getAutoloaderConfig() { }` and add to composer.json:

```json
"autoload": {
    "psr-0": { "Album": "module/Album/src/" }
},
```

If you go this way, then you need to run `php composer.phar update` to update the composer autoloading files.

5.2 Configuration

Having registered the autoloader, let's have a quick look at the `getConfig()` method in `Album\Module`. This method simply loads the `config/module.config.php` file.

Create a file called `module.config.php` under `zf2-tutorial/module/Album/config`:

```
<?php
return array(
    'controllers' => array(
        'invokables' => array(
            'Album\Controller\Album' => 'Album\Controller\AlbumController',
        ),
    ),
    'view_manager' => array(
        'template_path_stack' => array(
            'album' => __DIR__ . '/../view',
        ),
    ),
);
```

The config information is passed to the relevant components by the `ServiceManager`. We need two initial sections: `controllers` and `view_manager`. The controllers section provides a list of all the controllers provided by the module. We will need one controller, `AlbumController`, which we'll reference as `Album\Controller\Album`. The controller key must be unique across all modules, so we prefix it with our module name.

Within the `view_manager` section, we add our view directory to the `TemplatePathStack` configuration. This will allow it to find the view scripts for the `Album` module that are stored in our `view/` directory.

5.3 Informing the application about our new module

We now need to tell the `ModuleManager` that this new module exists. This is done in the application's `config/application.config.php` file which is provided by the skeleton application. Update this file so that its `modules` section contains the `Album` module as well, so the file now looks like this:

(Changes required are highlighted using comments.)

```
<?php
return array(
    'modules' => array(
        'Application',
        'Album',                  // <-- Add this line
    ),
    'module_listener_options' => array(
        'config_glob_paths'    => array(
            'config/autoload/{,*.}{global,local}.php',
        ),
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

As you can see, we have added our `Album` module into the list of modules after the `Application` module.

We have now set up the module ready for putting our custom code into it.

CHAPTER

SIX

ROUTING AND CONTROLLERS

We will build a very simple inventory system to display our album collection. The home page will list our collection and allow us to add, edit and delete albums. Hence the following pages are required:

Page	Description
Home	This will display the list of albums and provide links to edit and delete them. Also, a link to enable adding new albums will be provided.
Add new album	This page will provide a form for adding a new album.
Edit album	This page will provide a form for editing an album.
Delete album	This page will confirm that we want to delete an album and then delete it.

Before we set up our files, it's important to understand how the framework expects the pages to be organised. Each page of the application is known as an *action* and actions are grouped into *controllers* within *modules*. Hence, you would generally group related actions into a controller; for instance, a news controller might have actions of `current`, `archived` and `view`.

As we have four pages that all apply to albums, we will group them in a single controller `AlbumController` within our `Album` module as four actions. The four actions will be:

Page	Controller	Action
Home	`AlbumController`	`index`
Add new album	`AlbumController`	`add`
Edit album	`AlbumController`	`edit`
Delete album	`AlbumController`	`delete`

The mapping of a URL to a particular action is done using routes that are defined in the module's `module.config.php` file. We will add a route for our album actions. This is the updated module config file with the new code highlighted.

```
<?php
return array(
    'controllers' => array(
        'invokables' => array(
            'Album\Controller\Album' => 'Album\Controller\AlbumController',
        ),
    ),

    // The following section is new and should be added to your file
    'router' => array(
        'routes' => array(
            'album' => array(
                'type'    => 'segment',
                'options' => array(
```

```
                    'route'    => '/album[/][:action][/:id]',
                    'constraints' => array(
                        'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'id'     => '[0-9]+',
                    ),
                    'defaults' => array(
                        'controller' => 'Album\Controller\Album',
                        'action'     => 'index',
                    ),
                ),
            ),
        ),
    ),

    'view_manager' => array(
        'template_path_stack' => array(
            'album' => __DIR__ . '/../view',
        ),
    ),
);
```

The name of the route is 'album' and has a type of 'segment'. The segment route allows us to specify placeholders in the URL pattern (route) that will be mapped to named parameters in the matched route. In this case, the route is **''/album[/:action][/:id]''** which will match any URL that starts with `/album`. The next segment will be an optional action name, and then finally the next segment will be mapped to an optional id. The square brackets indicate that a segment is optional. The constraints section allows us to ensure that the characters within a segment are as expected, so we have limited actions to starting with a letter and then subsequent characters only being alphanumeric, underscore or hyphen. We also limit the id to a number.

This route allows us to have the following URLs:

URL	Page	Action
`/album`	Home (list of albums)	`index`
`/album/add`	Add new album	`add`
`/album/edit/2`	Edit album with an id of 2	`edit`
`/album/delete/4`	Delete album with an id of 4	`delete`

CHAPTER

SEVEN

CREATE THE CONTROLLER

We are now ready to set up our controller. In Zend Framework 2, the controller is a class that is generally called `{Controller name}Controller`. Note that `{Controller name}` must start with a capital letter. This class lives in a file called `{Controller name}Controller.php` within the `Controller` directory for the module. In our case that is `module/Album/src/Album/Controller`. Each action is a public method within the controller class that is named `{action name}Action`. In this case `{action name}` should start with a lower case letter.

Note: This is by convention. Zend Framework 2 doesn't provide many restrictions on controllers other than that they must implement the `Zend\Stdlib\Dispatchable` interface. The framework provides two abstract classes that do this for us: `Zend\Mvc\Controller\AbstractActionController` and `Zend\Mvc\Controller\AbstractRestfulController`. We'll be using the standard `AbstractActionController`, but if you're intending to write a RESTful web service, `AbstractRestfulController` may be useful.

Let's go ahead and create our controller class `AlbumController.php` at `zf2-tutorials/module/Album/src/Album/Controller`:

```
<?php
namespace Album\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class AlbumController extends AbstractActionController
{
    public function indexAction()
    {
    }

    public function addAction()
    {
    }

    public function editAction()
    {
    }

    public function deleteAction()
    {
    }
}
```

Note: We have already informed the module about our controller in the 'controller' section of `module/Album/config/module.config.php`.

We have now set up the four actions that we want to use. They won't work yet until we set up the views. The URLs for each action are:

URL	Method called
http://zf2-tutorial.localhost/album	`Album\Controller\AlbumController::indexAction`
http://zf2-tutorial.localhost/album/add	`Album\Controller\AlbumController::addAction`
http://zf2-tutorial.localhost/album/edit	`Album\Controller\AlbumController::editAction`
http://zf2-tutorial.localhost/album/delete	`Album\Controller\AlbumController::deleteAction`

We now have a working router and the actions are set up for each page of our application.

It's time to build the view and the model layer.

7.1 Initialise the view scripts

To integrate the view into our application all we need to do is create some view script files. These files will be executed by the `DefaultViewStrategy` and will be passed any variables or view models that are returned from the controller action method. These view scripts are stored in our module's views directory within a directory named after the controller. Create these four empty files now:

- `module/Album/view/album/album/index.phtml`
- `module/Album/view/album/album/add.phtml`
- `module/Album/view/album/album/edit.phtml`
- `module/Album/view/album/album/delete.phtml`

We can now start filling everything in, starting with our database and models.

CHAPTER

EIGHT

DATABASE AND MODELS

8.1 The database

Now that we have the `Album` module set up with controller action methods and view scripts, it is time to look at the model section of our application. Remember that the model is the part that deals with the application's core purpose (the so-called "business rules") and, in our case, deals with the database. We will make use of the Zend Framework class `Zend\Db\TableGateway\TableGateway` which is used to find, insert, update and delete rows from a database table.

We are going to use MySQL, via PHP's PDO driver, so create a database called `zf2tutorial`, and run these SQL statements to create the album table with some data in it.

```
CREATE TABLE album (
  id int(11) NOT NULL auto_increment,
  artist varchar(100) NOT NULL,
  title varchar(100) NOT NULL,
  PRIMARY KEY (id)
);
INSERT INTO album (artist, title)
    VALUES ('The Military Wives', 'In My Dreams');
INSERT INTO album (artist, title)
    VALUES ('Adele', '21');
INSERT INTO album (artist, title)
    VALUES ('Bruce Springsteen', 'Wrecking Ball (Deluxe)');
INSERT INTO album (artist, title)
    VALUES ('Lana Del Rey', 'Born To Die');
INSERT INTO album (artist, title)
    VALUES ('Gotye', 'Making Mirrors');
```

(The test data chosen happens to be the Bestsellers on Amazon UK at the time of writing!)

We now have some data in a database and can write a very simple model for it.

8.2 The model files

Zend Framework does not provide a `Zend\Model` component because the model is your business logic and it's up to you to decide how you want it to work. There are many components that you can use for this depending on your needs. One approach is to have model classes represent each entity in your application and then use mapper objects that load and save entities to the database. Another is to use an Object-relational mapping (ORM) technology, such as Doctrine or Propel.

For this tutorial, we are going to create a very simple model by creating an `AlbumTable` class that uses the `Zend\Db\TableGateway\TableGateway` class in which each album object is an `Album` object (known as an *entity*). This is an implementation of the Table Data Gateway design pattern to allow for interfacing with data in a database table. Be aware though that the Table Data Gateway pattern can become limiting in larger systems. There is also a temptation to put database access code into controller action methods as these are exposed by `Zend\Db\TableGateway\AbstractTableGateway`. *Don't do this*!

Let's start by creating a file called `Album.php` under `module/Album/src/Album/Model`:

```
<?php
namespace Album\Model;

class Album
{
    public $id;
    public $artist;
    public $title;

    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id'])) ? $data['id'] : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title'])) ? $data['title'] : null;
    }
}
```

Our `Album` entity object is a simple PHP class. In order to work with `Zend\Db`'s `TableGateway` class, we need to implement the `exchangeArray()` method. This method simply copies the data from the passed in array to our entity's properties. We will add an input filter for use with our form later.

Next, we create our `AlbumTable.php` file in `module/Album/src/Album/Model` directory like this:

```
<?php
namespace Album\Model;

use Zend\Db\TableGateway\TableGateway;

class AlbumTable
{
    protected $tableGateway;

    public function __construct(TableGateway $tableGateway)
    {
        $this->tableGateway = $tableGateway;
    }

    public function fetchAll()
    {
        $resultSet = $this->tableGateway->select();
        return $resultSet;
    }

    public function getAlbum($id)
    {
        $id  = (int) $id;
        $rowset = $this->tableGateway->select(array('id' => $id));
        $row = $rowset->current();
        if (!$row) {
            throw new \Exception("Could not find row $id");
```

```
        }
        return $row;
    }

    public function saveAlbum(Album $album)
    {
        $data = array(
            'artist' => $album->artist,
            'title'  => $album->title,
        );

        $id = (int)$album->id;
        if ($id == 0) {
            $this->tableGateway->insert($data);
        } else {
            if ($this->getAlbum($id)) {
                $this->tableGateway->update($data, array('id' => $id));
            } else {
                throw new \Exception('Form id does not exist');
            }
        }
    }

    public function deleteAlbum($id)
    {
        $this->tableGateway->delete(array('id' => $id));
    }
}
```

There's a lot going on here. Firstly, we set the protected property `$tableGateway` to the `TableGateway` instance passed in the constructor. We will use this to perform operations on the database table for our albums.

We then create some helper methods that our application will use to interface with the table gateway. `fetchAll()` retrieves all albums rows from the database as a `ResultSet`, `getAlbum()` retrieves a single row as an `Album` object, `saveAlbum()` either creates a new row in the database or updates a row that already exists and `deleteAlbum()` removes the row completely. The code for each of these methods is, hopefully, self-explanatory.

8.3 Using ServiceManager to configure the table gateway and inject into the AlbumTable

In order to always use the same instance of our `AlbumTable`, we will use the `ServiceManager` to define how to create one. This is most easily done in the Module class where we create a method called `getServiceConfig()` which is automatically called by the `ModuleManager` and applied to the `ServiceManager`. We'll then be able to retrieve it in our controller when we need it.

To configure the `ServiceManager`, we can either supply the name of the class to be instantiated or a factory (closure or callback) that instantiates the object when the `ServiceManager` needs it. We start by implementing `getServiceConfig()` to provide a factory that creates an `AlbumTable`. Add this method to the bottom of the `Module.php` file in `module/Album`.

```
<?php
namespace Album;

// Add these import statements:
use Album\Model\Album;
use Album\Model\AlbumTable;
```

```
use Zend\Db\ResultSet\ResultSet;
use Zend\Db\TableGateway\TableGateway;

class Module
{
    // getAutoloaderConfig() and getConfig() methods here

    // Add this method:
    public function getServiceConfig()
    {
        return array(
            'factories' => array(
                'Album\Model\AlbumTable' =>  function($sm) {
                    $tableGateway = $sm->get('AlbumTableGateway');
                    $table = new AlbumTable($tableGateway);
                    return $table;
                },
                'AlbumTableGateway' => function ($sm) {
                    $dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');
                    $resultSetPrototype = new ResultSet();
                    $resultSetPrototype->setArrayObjectPrototype(new Album());
                    return new TableGateway('album', $dbAdapter, null, $resultSetPrototype);
                },
            ),
        );
    }
}
```

This method returns an array of `factories` that are all merged together by the `ModuleManager` before passing to the `ServiceManager`. The factory for `Album\Model\AlbumTable` uses the `ServiceManager` to create an `AlbumTableGateway` to pass to the `AlbumTable`. We also tell the `ServiceManager` that an `AlbumTableGateway` is created by getting a `Zend\Db\Adapter\Adapter` (also from the `ServiceManager`) and using it to create a `TableGateway` object. The `TableGateway` is told to use an `Album` object whenever it creates a new result row. The TableGateway classes use the prototype pattern for creation of result sets and entities. This means that instead of instantiating when required, the system clones a previously instantiated object. See PHP Constructor Best Practices and the Prototype Pattern for more details.

Finally, we need to configure the `ServiceManager` so that it knows how to get a `Zend\Db\Adapter\Adapter`. This is done using a factory called `Zend\Db\Adapter\AdapterServiceFactory` which we can configure within the merged config system. Zend Framework 2's `ModuleManager` merges all the configuration from each module's `module.config.php` file and then merges in the files in `config/autoload` (`*.global.php` and then `*.local.php` files). We'll add our database configuration information to `global.php` which you should commit to your version control system. You can use `local.php` (outside of the VCS) to store the credentials for your database if you want to. Modify `config/autoload/global.php` (in the Zend Skeleton root, not inside the Album module) with following code:

```
<?php
return array(
    'db' => array(
        'driver'         => 'Pdo',
        'dsn'            => 'mysql:dbname=zf2tutorial;host=localhost',
        'driver_options' => array(
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
    ),
    'service_manager' => array(
        'factories' => array(
            'Zend\Db\Adapter\Adapter'
```

```
                    => 'Zend\Db\Adapter\AdapterServiceFactory',
        ),
    ),
);
```

You should put your database credentials in `config/autoload/local.php` so that they are not in the git repository (as `local.php` is ignored):

```
<?php
return array(
    'db' => array(
        'username' => 'YOUR USERNAME HERE',
        'password' => 'YOUR PASSWORD HERE',
    ),
);
```

8.4 Back to the controller

Now that the `ServiceManager` can create an `AlbumTable` instance for us, we can add a method to the controller to retrieve it. Add `getAlbumTable()` to the `AlbumController` class:

```
// module/Album/src/Album/Controller/AlbumController.php:
    public function getAlbumTable()
    {
        if (!$this->albumTable) {
            $sm = $this->getServiceLocator();
            $this->albumTable = $sm->get('Album\Model\AlbumTable');
        }
        return $this->albumTable;
    }
```

You should also add:

```
protected $albumTable;
```

to the top of the class.

We can now call `getAlbumTable()` from within our controller whenever we need to interact with our model.

If the service locator was configured correctly in `Module.php`, then we should get an instance of `Album\Model\AlbumTable` when calling `getAlbumTable()`.

8.5 Listing albums

In order to list the albums, we need to retrieve them from the model and pass them to the view. To do this, we fill in `indexAction()` within `AlbumController`. Update the `AlbumController`'s `indexAction()` like this:

```
// module/Album/src/Album/Controller/AlbumController.php:
// ...
    public function indexAction()
    {
        return new ViewModel(array(
            'albums' => $this->getAlbumTable()->fetchAll(),
        ));
```

```
    }
// ...
```

With Zend Framework 2, in order to set variables in the view, we return a `ViewModel` instance where the first parameter of the constructor is an array from the action containing data we need. These are then automatically passed to the view script. The `ViewModel` object also allows us to change the view script that is used, but the default is to use `{controller name}/{action name}`. We can now fill in the `index.phtml` view script:

```
<?php
// module/Album/view/album/album/index.phtml:

$title = 'My albums';
$this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>
<p>
    <a href="<?php echo $this->url('album', array('action'=>'add'));?>">Add new album</a>
</p>

<table class="table">
<tr>
    <th>Title</th>
    <th>Artist</th>
    <th> </th>
</tr>
<?php foreach ($albums as $album) : ?>
<tr>
    <td><?php echo $this->escapeHtml($album->title);?></td>
    <td><?php echo $this->escapeHtml($album->artist);?></td>
    <td>
        <a href="<?php echo $this->url('album',
            array('action'=>'edit', 'id' => $album->id));?>">Edit</a>
        <a href="<?php echo $this->url('album',
            array('action'=>'delete', 'id' => $album->id));?>">Delete</a>
    </td>
</tr>
<?php endforeach; ?>
</table>
```

The first thing we do is to set the title for the page (used in the layout) and also set the title for the `<head>` section using the `headTitle()` view helper which will display in the browser's title bar. We then create a link to add a new album.

The `url()` view helper is provided by Zend Framework 2 and is used to create the links we need. The first parameter to `url()` is the route name we wish to use for construction of the URL, and the second parameter is an array of all the variables to fit into the placeholders to use. In this case we use our 'album' route which is set up to accept two placeholder variables: `action` and `id`.

We iterate over the `$albums` that we assigned from the controller action. The Zend Framework 2 view system automatically ensures that these variables are extracted into the scope of the view script, so that we don't have to worry about prefixing them with `$this->` as we used to have to do with Zend Framework 1; however you can do so if you wish.

We then create a table to display each album's title and artist, and provide links to allow for editing and deleting the record. A standard `foreach:` loop is used to iterate over the list of albums, and we use the alternate form using a colon and `endforeach;` as it is easier to scan than to try and match up braces. Again, the `url()` view helper is used to create the edit and delete links.

Note: We always use the `escapeHtml()` view helper to help protect ourselves from Cross Site Scripting (XSS) vulnerabilities (see http://en.wikipedia.org/wiki/Cross-site_scripting).

If you open http://zf2-tutorial.localhost/album you should see this:

CHAPTER

NINE

STYLING AND TRANSLATIONS

We've picked up the SkeletonApplication's styling, which is fine, but we need to change the title and remove the copyright message.

The ZendSkeletonApplication is set up to use `Zend\I18n`'s translation functionality for all the text. It uses `.po` files that live in `Application/language`, and you need to use poedit to change the text. Start poedit and open `application/language/en_US.po`. Click on "Skeleton Application" in the list of `Original` strings and then type in "Tutorial" as the translation.

Press Save in the toolbar and poedit will create an `en_US.mo` file for us. If you find that no `.mo` file is generated, check `Preferences -> Editor -> Behavior` and see if the checkbox marked `Automatically compile .mo file on save` is checked.

To remove the copyright message, we need to edit the `Application` module's `layout.phtml` view script:

```
// module/Application/view/layout/layout.phtml:
// Remove this line:
<p>&copy; 2005 - 2012 by Zend Technologies Ltd. <?php echo $this->translate('All
rights reserved.') ?></p>
```

The page now looks ever so slightly better now!

CHAPTER

TEN

FORMS AND ACTIONS

10.1 Adding new albums

We can now code up the functionality to add new albums. There are two bits to this part:

- Display a form for user to provide details
- Process the form submission and store to database

We use `Zend\Form` to do this. The `Zend\Form` component manages the form and, for validation, we add a `Zend\InputFilter` to our `Album` entity. We start by creating a new class `Album\Form\AlbumForm` that extends from `Zend\Form\Form` to define our form. Create a file called `AlbumForm.php` in `module/Album/src/Album/Form`:

```
<?php
namespace Album\Form;

use Zend\Form\Form;

class AlbumForm extends Form
{
    public function __construct($name = null)
    {
        // we want to ignore the name passed
        parent::__construct('album');
        $this->setAttribute('method', 'post');
        $this->add(array(
            'name' => 'id',
            'type' => 'Hidden',
        ));
        $this->add(array(
            'name' => 'title',
            'type' => 'Text',
            'options' => array(
                'label' => 'Title',
            ),
        ));
        $this->add(array(
            'name' => 'artist',
            'type' => 'Text',
            'options' => array(
                'label' => 'Artist',
            ),
        ));
```

```
        $this->add(array(
            'name' => 'submit',
            'type' => 'Submit',
            'attributes' => array(
                'value' => 'Go',
                'id' => 'submitbutton',
            ),
        ));
    }
}
```

Within the constructor of `AlbumForm` we do several things. First, we set the name of the form as we call the parent's constructor. We then set the form's method, in this case, `post`. Finally, we create four form elements: the id, title, artist, and submit button. For each item we set various attributes and options, including the label to be displayed.

We also need to set up validation for this form. In Zend Framework 2 this is done using an input filter, which can either be standalone or defined within any class that implements the `InputFilterAwareInterface` interface, such as a model entity. In our case, we are going to add the input filter to the Album class, which resides in the `Album.php` file in `module/Album/src/Album/Model`:

```
<?php
namespace Album\Model;

// Add these import statements
use Zend\InputFilter\Factory as InputFactory;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\InputFilterAwareInterface;
use Zend\InputFilter\InputFilterInterface;

class Album implements InputFilterAwareInterface
{
    public $id;
    public $artist;
    public $title;
    protected $inputFilter;                       // <-- Add this variable

    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id']))     ? $data['id']     : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title']))  ? $data['title']  : null;
    }

    // Add content to these methods:
    public function setInputFilter(InputFilterInterface $inputFilter)
    {
        throw new \Exception("Not used");
    }

    public function getInputFilter()
    {
        if (!$this->inputFilter) {
            $inputFilter = new InputFilter();
            $factory     = new InputFactory();

            $inputFilter->add($factory->createInput(array(
                'name'     => 'id',
                'required' => true,
```

```
            'filters'  => array(
                array('name' => 'Int'),
            ),
        )));

        $inputFilter->add($factory->createInput(array(
            'name'     => 'artist',
            'required' => true,
            'filters'  => array(
                array('name' => 'StripTags'),
                array('name' => 'StringTrim'),
            ),
            'validators' => array(
                array(
                    'name'    => 'StringLength',
                    'options' => array(
                        'encoding' => 'UTF-8',
                        'min'      => 1,
                        'max'      => 100,
                    ),
                ),
            ),
        )));

        $inputFilter->add($factory->createInput(array(
            'name'     => 'title',
            'required' => true,
            'filters'  => array(
                array('name' => 'StripTags'),
                array('name' => 'StringTrim'),
            ),
            'validators' => array(
                array(
                    'name'    => 'StringLength',
                    'options' => array(
                        'encoding' => 'UTF-8',
                        'min'      => 1,
                        'max'      => 100,
                    ),
                ),
            ),
        )));

        $this->inputFilter = $inputFilter;
    }

    return $this->inputFilter;
  }
}
```

The `InputFilterAwareInterface` defines two methods: `setInputFilter()` and `getInputFilter()`. We only need to implement `getInputFilter()` so we simply throw an exception in `setInputFilter()`.

Within `getInputFilter()`, we instantiate an `InputFilter` and then add the inputs that we require. We add one input for each property that we wish to filter or validate. For the `id` field we add an `Int` filter as we only need integers. For the text elements, we add two filters, `StripTags` and `StringTrim`, to remove unwanted HTML and unnecessary white space. We also set them to be *required* and add a `StringLength` validator to ensure that the user doesn't enter more characters than we can store into the database.

We now need to get the form to display and then process it on submission. This is done within the `AlbumController`'s `addAction()`:

```
// module/Album/src/Album/Controller/AlbumController.php:

//...
use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Album\Model\Album;          // <-- Add this import
use Album\Form\AlbumForm;       // <-- Add this import
//...

    // Add content to this method:
    public function addAction()
    {
        $form = new AlbumForm();
        $form->get('submit')->setValue('Add');

        $request = $this->getRequest();
        if ($request->isPost()) {
            $album = new Album();
            $form->setInputFilter($album->getInputFilter());
            $form->setData($request->getPost());

            if ($form->isValid()) {
                $album->exchangeArray($form->getData());
                $this->getAlbumTable()->saveAlbum($album);

                // Redirect to list of albums
                return $this->redirect()->toRoute('album');
            }
        }
        return array('form' => $form);
    }
//...
```

After adding the `AlbumForm` to the use list, we implement `addAction()`. Let's look at the `addAction()` code in a little more detail:

```
$form = new AlbumForm();
$form->get('submit')->setValue('Add');
```

We instantiate `AlbumForm` and set the label on the submit button to "Add". We do this here as we'll want to re-use the form when editing an album and will use a different label.

```
$request = $this->getRequest();
if ($request->isPost()) {
    $album = new Album();
    $form->setInputFilter($album->getInputFilter());
    $form->setData($request->getPost());
    if ($form->isValid()) {
```

If the `Request` object's `isPost()` method is true, then the form has been submitted and so we set the form's input filter from an album instance. We then set the posted data to the form and check to see if it is valid using the `isValid()` member function of the form.

```
$album->exchangeArray($form->getData());
$this->getAlbumTable()->saveAlbum($album);
```

If the form is valid, then we grab the data from the form and store to the model using `saveAlbum()`.

```
// Redirect to list of albums
return $this->redirect()->toRoute('album');
```

After we have saved the new album row, we redirect back to the list of albums using the `Redirect` controller plugin.

```
return array('form' => $form);
```

Finally, we return the variables that we want assigned to the view. In this case, just the form object. Note that Zend Framework 2 also allows you to simply return an array containing the variables to be assigned to the view and it will create a `ViewModel` behind the scenes for you. This saves a little typing.

We now need to render the form in the add.phtml view script:

```
<?php
// module/Album/view/album/album/add.phtml:

$title = 'Add new album';
$this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>
<?php
$form = $this->form;
$form->setAttribute('action', $this->url('album', array('action' => 'add')));
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formHidden($form->get('id'));
echo $this->formRow($form->get('title'));
echo $this->formRow($form->get('artist'));
echo $this->formSubmit($form->get('submit'));
echo $this->form()->closeTag();
```

Again, we display a title as before and then we render the form. Zend Framework provides some view helpers to make this a little easier. The `form()` view helper has an `openTag()` and `closeTag()` method which we use to open and close the form. Then for each element with a label, we can use `formRow()`, but for the two elements that are standalone, we use `formHidden()` and `formSubmit()`.

Alternatively, the process of rendering the form can be simplified by using the bundled `formCollection` view helper. For example, in the view script above replace all the form-rendering echo statements with:

```
echo $this->formCollection($form);
```

Note: You still need to call the `openTag` and `closeTag` methods of the form. You replace the other echo statements with the call to `formCollection`, above.

This will iterate over the form structure, calling the appropriate label, element and error view helpers for each element, but you still have to wrap formCollection($form) with the open and close form tags. This helps reduce the complexity of your view script in situations where the default HTML rendering of the form is acceptable.

You should now be able to use the "Add new album" link on the home page of the application to add a new album record.

10.2 Editing an album

Editing an album is almost identical to adding one, so the code is very similar. This time we use `editAction()` in the `AlbumController`:

```
// module/Album/src/Album/Controller/AlbumController.php:
//...

    // Add content to this method:
    public function editAction()
    {
        $id = (int) $this->params()->fromRoute('id', 0);
        if (!$id) {
            return $this->redirect()->toRoute('album', array(
                'action' => 'add'
            ));
        }

        // Get the Album with the specified id.  An exception is thrown
        // if it cannot be found, in which case go to the index page.
        try {
            $album = $this->getAlbumTable()->getAlbum($id);
        }
        catch (\Exception $ex) {
            return $this->redirect()->toRoute('album', array(
                'action' => 'index'
            ));
        }

        $form  = new AlbumForm();
        $form->bind($album);
        $form->get('submit')->setAttribute('value', 'Edit');

        $request = $this->getRequest();
        if ($request->isPost()) {
            $form->setInputFilter($album->getInputFilter());
            $form->setData($request->getPost());

            if ($form->isValid()) {
                $this->getAlbumTable()->saveAlbum($form->getData());

                // Redirect to list of albums
                return $this->redirect()->toRoute('album');
            }
        }

        return array(
            'id' => $id,
            'form' => $form,
        );
    }
//...
```

This code should look comfortably familiar. Let's look at the differences from adding an album. Firstly, we look for the `id` that is in the matched route and use it to load the album to be edited:

```
$id = (int) $this->params()->fromRoute('id', 0);
if (!$id) {
    return $this->redirect()->toRoute('album', array(
        'action' => 'add'
    ));
}
```

```
// Get the album with the specified id.  An exception is thrown
// if it cannot be found, in which case go to the index page.
try {
    $album = $this->getAlbumTable()->getAlbum($id);
}
catch (\Exception $ex) {
    return $this->redirect()->toRoute('album', array(
        'action' => 'index'
    ));
}
```

`params` is a controller plugin that provides a convenient way to retrieve parameters from the matched route. We use it to retrieve the `id` from the route we created in the modules' `module.config.php`. If the `id` is zero, then we redirect to the add action, otherwise, we continue by getting the album entity from the database.

We have to check to make sure that the Album with the specified `id` can actually be found. If it cannot, then the data access method throws an exception. We catch that exception and re-route the user to the index page.

```
$form = new AlbumForm();
$form->bind($album);
$form->get('submit')->setAttribute('value', 'Edit');
```

The form's `bind()` method attaches the model to the form. This is used in two ways:

- When displaying the form, the initial values for each element are extracted from the model.
- After successful validation in isValid(), the data from the form is put back into the model.

These operations are done using a hydrator object. There are a number of hydrators, but the default one is `Zend\Stdlib\Hydrator\ArraySerializable` which expects to find two methods in the model: `getArrayCopy()` and `exchangeArray()`. We have already written `exchangeArray()` in our `Album` entity, so just need to write `getArrayCopy()`:

```
// module/Album/src/Album/Model/Album.php:
// ...
    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id']))     ? $data['id']     : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title']))  ? $data['title']  : null;
    }

    // Add the following method:
    public function getArrayCopy()
    {
        return get_object_vars($this);
    }
// ...
```

As a result of using `bind()` with its hydrator, we do not need to populate the form's data back into the `$album` as that's already been done, so we can just call the mappers' `saveAlbum()` to store the changes back to the database.

The view template, `edit.phtml`, looks very similar to the one for adding an album:

```
<?php
// module/Album/view/album/album/edit.phtml:

$title = 'Edit album';
$this->headTitle($title);
?>
```

```
<h1><?php echo $this->escapeHtml($title); ?></h1>

<?php
$form = $this->form;
$form->setAttribute('action', $this->url(
    'album',
    array(
        'action' => 'edit',
        'id'     => $this->id,
    )
));
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formHidden($form->get('id'));
echo $this->formRow($form->get('title'));
echo $this->formRow($form->get('artist'));
echo $this->formSubmit($form->get('submit'));
echo $this->form()->closeTag();
```

The only changes are to use the 'Edit Album' title and set the form's action to the 'edit' action too.

You should now be able to edit albums.

10.3 Deleting an album

To round out our application, we need to add deletion. We have a Delete link next to each album on our list page and the naive approach would be to do a delete when it's clicked. This would be wrong. Remembering our HTTP spec, we recall that you shouldn't do an irreversible action using GET and should use POST instead.

We shall show a confirmation form when the user clicks delete and if they then click "yes", we will do the deletion. As the form is trivial, we'll code it directly into our view (`Zend\Form` is, after all, optional!).

Let's start with the action code in `AlbumController::deleteAction()`:

```
// module/Album/src/Album/Controller/AlbumController.php:
//...
    // Add content to the following method:
    public function deleteAction()
    {
        $id = (int) $this->params()->fromRoute('id', 0);
        if (!$id) {
            return $this->redirect()->toRoute('album');
        }

        $request = $this->getRequest();
        if ($request->isPost()) {
            $del = $request->getPost('del', 'No');

            if ($del == 'Yes') {
                $id = (int) $request->getPost('id');
                $this->getAlbumTable()->deleteAlbum($id);
            }

            // Redirect to list of albums
            return $this->redirect()->toRoute('album');
        }
```

```
        return array(
            'id'    => $id,
            'album' => $this->getAlbumTable()->getAlbum($id)
        );
    }
//...
```

As before, we get the `id` from the matched route, and check the request object's `isPost()` to determine whether to show the confirmation page or to delete the album. We use the table object to delete the row using the `deleteAlbum()` method and then redirect back the list of albums. If the request is not a POST, then we retrieve the correct database record and assign to the view, along with the `id`.

The view script is a simple form:

```
<?php
// module/Album/view/album/album/delete.phtml:

$title = 'Delete album';
$this->headTitle($title);
?>
<h1><?php echo $this->escapeHtml($title); ?></h1>

<p>Are you sure that you want to delete
    '<?php echo $this->escapeHtml($album->title); ?>' by
    '<?php echo $this->escapeHtml($album->artist); ?>'?
</p>
<?php
$url = $this->url('album', array(
    'action' => 'delete',
    'id'     => $this->id,
));
?>
<form action="<?php echo $url; ?>" method="post">
<div>
    <input type="hidden" name="id" value="<?php echo (int) $album->id; ?>" />
    <input type="submit" name="del" value="Yes" />
    <input type="submit" name="del" value="No" />
</div>
</form>
```

In this script, we display a confirmation message to the user and then a form with "Yes" and "No" buttons. In the action, we checked specifically for the "Yes" value when doing the deletion.

10.4 Ensuring that the home page displays the list of albums

One final point. At the moment, the home page, http://zf2-tutorial.localhost/ doesn't display the list of albums.

This is due to a route set up in the `Application` module's `module.config.php`. To change it, open `module/Application/config/module.config.php` and find the home route:

```
'home' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route'    => '/',
        'defaults' => array(
            'controller' => 'Application\Controller\Index',
            'action'     => 'index',
```

```
        ),
    ),
),
```

Change the `controller` from `Application\Controller\Index` to `Album\Controller\Album`:

```
'home' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route'    => '/',
        'defaults' => array(
            'controller' => 'Album\Controller\Album', // <-- change here
            'action'     => 'index',
        ),
    ),
),
```

That's it - you now have a fully working application!

CHAPTER

ELEVEN

CONCLUSION

This concludes our brief look at building a simple, but fully functional, MVC application using Zend Framework 2.

In this tutorial we but briefly touched quite a number of different parts of the framework.

The most important part of applications built with Zend Framework 2 are the *modules*, the building blocks of any *MVC ZF2 application*.

To ease the work with dependencies inside our applications, we use the *service manager*.

To be able to map a request to controllers and their actions, we use *routes*.

Data persistence, in most cases, includes using *Zend\Db* to communicate with one of the databases. Input data is filtered and validated with *input filters* and together with *Zend\Form* they provide a strong bridge between the domain model and the view layer.

Zend\View is responsible for the View in the MVC stack, together with a vast amount of *view helpers*.

CHAPTER

TWELVE

ZEND FRAMEWORK TOOL (ZFTOOL)

ZFTool is a utility module for maintaining modular Zend Framework 2 applications. It runs from the command line and can be installed as ZF2 module or as PHAR (see below). This tool gives you the ability to:

- create a ZF2 project, installing a skeleton application;
- create a new module inside an existing ZF2 application;
- get the list of all the modules installed inside an application;
- get the configuration file of a ZF2 application;
- install the ZF2 library choosing a specific version.

To install the ZFTool you can use one of the following methods or you can just download the PHAR package and use it.

12.1 Installation using Composer

1. Open console (command prompt)
2. Go to your application's directory
3. Run *composer require zendframework/zftool:dev-master*

12.2 Manual installation

1. Clone using *git* or download zipball
2. Extract to *vendor/ZFTool* in your ZF2 application
3. Enter the *vendor/ZFTool* folder and execute *zf.php* as reported below.

12.3 Without installation, using the PHAR file

1. You don't need to install ZFTool if you want just use it as a shell command. You can download zftool.phar and use it.

12.4 Usage

In the following usage examples, the **zf.php** command can be replace with **zftool.phar**.

12.4.1 Basic information

```
> zf.php modules [list]           show loaded modules
```

The *modules* option gives you the list of all the modules installed in a ZF2 application.

```
> zf.php version | --version      display current Zend Framework version
```

The *version* option gives you the version number of ZFTool and, if executed from the root folder of a ZF2 application, the version number of the Zend Framework library used by the application.

12.4.2 Project creation

```
> zf.php create project <path>

<path>              The path of the project to be created
```

This command installs the ZendSkeletonApplication in the specified path.

12.4.3 Module creation

```
> zf.php create module <name> [<path>]

<name>              The name of the module to be created
<path>              The path to the root folder of the ZF2 application (optional)
```

This command can be used to create a new module inside an existing ZF2 application. If the path is not provided the ZFTool try to create a new module in the local directory (only if the local folder contains a ZF2 application).

12.4.4 Classmap generator

```
> zf.php classmap generate <directory> <classmap file> [--append|-a] [--overwrite|-w]

<directory>         The directory to scan for PHP classes (use "." to use current directory)
<classmap file>     File name for generated class map file  or - for standard output. If not supplie
                    autoload_classmap.php inside <directory>.
--append | -a       Append to classmap file if it exists
--overwrite | -w    Whether or not to overwrite existing classmap file
```

12.4.5 ZF library installation

```
> zf.php install zf <path> [<version>]

<path>              The directory where to install the ZF2 library
<version>           The version to install, if not specified uses the last available
```

This command install the specified version of the ZF2 library in a path. If the version is omitted it will be used the last stable available. Using this command you can install all the tag version specified in the ZF2 github repository (the name used for the version is obtained removing the *'release-'* string from the tag name; for instance, the tag *'release-2.0.0'* is equivalent to the version number *2.0.0*).

12.4.6 Compile the PHAR file

You can create a .phar file containing the ZFTool project. In order to compile ZFTool in a .phar file you need to execute the following command:

```
> bin/create-phar
```

This command will create a *zftool.phar* file in the bin folder. You can use and ship only this file to execute all the ZFTool functionalities. After the *zftool.phar* creation, we suggest to add the folder bin of ZFTool in your PATH environment. In this way you can execute the *zftool.phar* script wherever you are.

CHAPTER

THIRTEEN

LEARNING DEPENDENCY INJECTION

13.1 Very brief introduction to Di.

Dependency Injection is a concept that has been talked about in numerous places over the web. For the purposes of this quickstart, we'll explain the act of injecting dependencies simply with this below code:

```
$b = new B(new A());
```

Above, A is a dependency of B, and A was **injected** into B. If you are not familiar with the concept of dependency injection, here are a couple of great reads: Matthew Weier O'Phinney's Analogy, Ralph Schindler's Learning DI, or Fabien Potencier's Series on DI.

13.2 Very brief introduction to Di Container.

```
TBD.
```

13.3 Simplest usage case (2 classes, one consumes the other)

In the simplest use case, a developer might have one class (`A`) that is consumed by another class (`B`) through the constructor. By having the dependency injected through the constructor, this requires an object of type `A` be instantiated before an object of type `B` so that `A` can be injected into `B`.

```
namespace My {

    class A
    {
        /* Some useful functionality */
    }

    class B
    {
        protected $a = null;
        public function __construct(A $a)
        {
            $this->a = $a;
        }
    }
}
```

To create B by hand, a developer would follow this work flow, or a similar workflow to this:

```
$b = new B(new A());
```

If this workflow becomes repeated throughout your application multiple times, this creates an opportunity where one might want to DRY up the code. While there are several ways to do this, using a dependency injection container is one of these solutions. With Zend's dependency injection container Zend\Di\Di, the above use case can be taken care of with no configuration (provided all of your autoloading is already configured properly) with the following usage:

```
$di = new Zend\Di\Di;
$b = $di->get('My\B'); // will produce a B object that is consuming an A object
```

Moreover, by using the Di::get() method, you are ensuring that the same exact object is returned on subsequent calls. To force new objects to be created on each and every request, one would use the Di::newInstance() method:

```
$b = $di->newInstance('My\B');
```

Let's assume for a moment that A requires some configuration before it can be created. Our previous use case is expanded to this (we'll throw a 3rd class in for good measure):

```
namespace My {

    class A
    {
        protected $username = null;
        protected $password = null;
        public function __construct($username, $password)
        {
            $this->username = $username;
            $this->password = $password;
        }
    }

    class B
    {
        protected $a = null;
        public function __construct(A $a)
        {
            $this->a = $a;
        }
    }

    class C
    {
        protected $b = null;
        public function __construct(B $b)
        {
            $this->b = $b;
        }
    }

}
```

With the above, we need to ensure that our Di is capable of seeing the A class with a few configuration values (which are generally scalar in nature). To do this, we need to interact with the InstanceManager:

```
$di = new Zend\Di\Di;
$di->getInstanceManager()->setProperty('A', 'username', 'MyUsernameValue');
$di->getInstanceManager()->setProperty('A', 'password', 'MyHardToGuessPassword%$#');
```

Now that our container has values it can use when creating `A`, and our new goal is to have a `C` object that consumes `B` and in turn consumes `A`, the usage scenario is still the same:

```
$c = $di->get('My\C');
// or
$c = $di->newInstance('My\C');
```

Simple enough, but what if we wanted to pass in these parameters at call time? Assuming a default `Di` object (`$di = new Zend\Di\Di()` without any configuration to the `InstanceManager`), we could do the following:

```
$parameters = array(
    'username' => 'MyUsernameValue',
    'password' => 'MyHardToGuessPassword%$#',
);

$c = $di->get('My\C', $parameters);
// or
$c = $di->newInstance('My\C', $parameters);
```

Constructor injection is not the only supported type of injection. The other most popular method of injection is also supported: setter injection. Setter injection allows one to have a usage scenario that is the same as our previous example with the exception, for example, of our `B` class now looking like this:

```
namespace My {
    class B
    {
        protected $a;
        public function setA(A $a)
        {
            $this->a = $a;
        }
    }
}
```

Since the method is prefixed with set, and is followed by a capital letter, the `Di` knows that this method is used for setter injection, and again, the use case `$c = $di->get('C')`, will once again know how to fill the dependencies when needed to create an object of type `C`.

Other methods are being created to determine what the wirings between classes are, such as interface injection and annotation based injection.

13.4 Simplest Usage Case Without Type-hints

If your code does not have type-hints or you are using 3rd party code that does not have type-hints but does practice dependency injection, you can still use the `Di`, but you might find you need to describe your dependencies explicitly. To do this, you will need to interact with one of the definitions that is capable of letting a developer describe, with objects, the map between classes. This particular definition is called the `BuilderDefinition` and can work with, or in place of, the default `RuntimeDefinition`.

Definitions are a part of the `Di` that attempt to describe the relationship between classes so that `Di::newInstance()` and `Di::get()` can know what the dependencies are that need to be filled for a particular class/object. With no configuration, `Di` will use the `RuntimeDefinition` which uses reflection and the type-hints in your code to determine the dependency map. Without type-hints, it will assume that all dependencies are scalar or required configuration parameters.

The `BuilderDefinition`, which can be used in tandem with the `RuntimeDefinition` (technically, it can be used in tandem with any definition by way of the `AggregateDefinition`), allows you to programmatically describe the mappings with objects. Let's say for example, our above `A/B/C` usage scenario, were altered such that class `B` now looks like this:

```
namespace My {
    class B
    {
        protected $a;
        public function setA($a)
        {
            $this->a = $a;
        }
    }
}
```

You'll notice the only change is that setA now does not include any type-hinting information.

```
use Zend\Di\Di;
use Zend\Di\Definition;
use Zend\Di\Definition\Builder;

// Describe this class:
$builder = new Definition\BuilderDefinition;
$builder->addClass(($class = new Builder\PhpClass));

$class->setName('My\B');
$class->addInjectableMethod(($im = new Builder\InjectableMethod));

$im->setName('setA');
$im->addParameter('a', 'My\A');

// Use both our Builder Definition as well as the default
// RuntimeDefinition, builder first
$aDef = new Definition\AggregateDefinition;
$aDef->addDefinition($builder);
$aDef->addDefinition(new Definition\RuntimeDefinition);

// Now make sure the Di understands it
$di = new Di;
$di->setDefinition($aDef);

// and finally, create C
$parameters = array(
    'username' => 'MyUsernameValue',
    'password' => 'MyHardToGuessPassword%$#',
);

$c = $di->get('My\C', $parameters);
```

This above usage scenario provides that whatever the code looks like, you can ensure that it works with the dependency injection container. In an ideal world, all of your code would have the proper type hinting and/or would be using a mapping strategy that reduces the amount of bootstrapping work that needs to be done in order to have a full definition that is capable of instantiating all of the objects you might require.

13.5 Simplest usage case with Compiled Definition

Without going into the gritty details, as you might expect, PHP at its core is not DI friendly. Out-of-the-box, the `Di` uses a `RuntimeDefinition` which does all class map resolution via PHP's `Reflection` extension. Couple that with the fact that PHP does not have a true application layer capable of storing objects in-memory between requests, and you get a recipe that is less performant than similar solutions you'll find in Java and .Net (where there is an application layer with in-memory object storage.)

To mitigate this shortcoming, `Zend\Di` has several features built in capable of pre-compiling the most expensive tasks that surround dependency injection. It is worth noting that the `RuntimeDefinition`, which is used by default, is the **only** definition that does lookups on-demand. The rest of the `Definition` objects are capable of being aggregated and stored to disk in a very performant way.

Ideally, 3rd party code will ship with a pre-compiled `Definition` that will describe the various relationships and parameter/property needs of each class that is to be instantiated. This `Definition` would have been built as part of some deployment or packaging task by this 3rd party. When this is not the case, you can create these `Definitions` via any of the `Definition` types provided with the exception of the `RuntimeDefinition`. Here is a breakdown of the job of each definition type:

- `AggregateDefinition`- Aggregates multiple definitions of various types. When looking for a class, it looks it up in the order the definitions were provided to this aggregate.
- `ArrayDefinition`- This definition takes an array of information and exposes it via the interface provided by `Zend\Di\Definition` suitable for usage by `Di` or an `AggregateDefinition`
- `BuilderDefinition`- Creates a definition based on an object graph consisting of various `Builder\PhpClass` objects and `Builder\InjectionMethod` objects that describe the mapping needs of the target codebase and ...
- `Compiler`- This is not actually a definition, but produces an `ArrayDefinition` based off of a code scanner (`Zend\Code\Scanner\DirectoryScanner` or `Zend\Code\Scanner\FileScanner`).

The following is an example of producing a definition via a `DirectoryScanner`:

```
$compiler = new Zend\Di\Definition\Compiler();
$compiler->addCodeScannerDirectory(
    new Zend\Code\Scanner\ScannerDirectory('path/to/library/My/')
);
$definition = $compiler->compile();
```

This definition can then be directly used by the `Di` (assuming the above `A, B, C` scenario was actually a file per class on disk):

```
$di = new Zend\Di\Di;
$di->setDefinition($definition);
$di->getInstanceManager()->setProperty('My\A', 'username', 'foo');
$di->getInstanceManager()->setProperty('My\A', 'password', 'bar');
$c = $di->get('My\C');
```

One strategy for persisting these compiled definitions would be the following:

```
if (!file_exists(__DIR__ . '/di-definition.php') && $isProduction) {
    $compiler = new Zend\Di\Definition\Compiler();
    $compiler->addCodeScannerDirectory(
        new Zend\Code\Scanner\ScannerDirectory('path/to/library/My/')
    );
    $definition = $compiler->compile();
    file_put_contents(
        __DIR__ . '/di-definition.php',
```

```
        '<?php return ' . var_export($definition->toArray(), true) . ';'
    );
} else {
    $definition = new Zend\Di\Definition\ArrayDefinition(
        include __DIR__ . '/di-definition.php'
    );
}

// $definition can now be used; in a production system it will be written
// to disk.
```

Since `Zend\Code\Scanner` does not include files, the classes contained within are not loaded into memory. Instead, `Zend\Code\Scanner` uses tokenization to determine the structure of your files. This makes this suitable to use this solution during development and within the same request as any one of your application's dispatched actions.

13.6 Creating a precompiled definition for others to use

If you are a 3rd party code developer, it makes sense to produce a `Definition` file that describes your code so that others can utilize this `Definition` without having to `Reflect` it via the `RuntimeDefinition`, or create it via the `Compiler`. To do this, use the same technique as above. Instead of writing the resulting array to disk, you would write the information into a definition directly, by way of `Zend\Code\Generator`:

```
// First, compile the information
$compiler = new Zend\Di\Definition\CompilerDefinition();
$compiler->addDirectoryScanner(
    new Zend\Code\Scanner\DirectoryScanner(__DIR__ . '/My/')
);
$compiler->compile();
$definition = $compiler->toArrayDefinition();

// Now, create a Definition class for this information
$codeGenerator = new Zend\Code\Generator\FileGenerator();
$codeGenerator->setClass(($class = new Zend\Code\Generator\ClassGenerator()));
$class->setNamespaceName('My');
$class->setName('DiDefinition');
$class->setExtendedClass('\Zend\Di\Definition\ArrayDefinition');
$class->addMethod(
    '__construct',
    array(),
    \Zend\Code\Generator\MethodGenerator::FLAG_PUBLIC,
    'parent::__construct(' . var_export($definition->toArray(), true) . ');'
);
file_put_contents(__DIR__ . '/My/DiDefinition.php', $codeGenerator->generate());
```

13.7 Using Multiple Definitions From Multiple Sources

In all actuality, you will be using code from multiple places, some Zend Framework code, some other 3rd party code, and of course, your own code that makes up your application. Here is a method for consuming definitions from multiple places:

```
use Zend\Di\Di;
use Zend\Di\Definition;
use Zend\Di\Definition\Builder;
```

```

$di = new Di;
$diDefAggregate = new Definition\Aggregate();

// first add in provided Definitions, for example
$diDefAggregate->addDefinition(new ThirdParty\Dbal\DiDefinition());
$diDefAggregate->addDefinition(new Zend\Controller\DiDefinition());

// for code that does not have TypeHints
$builder = new Definition\BuilderDefinition();
$builder->addClass(($class = Builder\PhpClass));
$class->addInjectionMethod(
    ($injectMethod = new Builder\InjectionMethod())
);
$injectMethod->setName('injectImplementation');
$injectMethod->addParameter(
'implementation', 'Class\For\Specific\Implementation'
);

// now, your application code
$compiler = new Definition\Compiler()
$compiler->addCodeScannerDirectory(
    new Zend\Code\Scanner\DirectoryScanner(__DIR__ . '/App/')
);
$appDefinition = $compiler->compile();
$diDefAggregate->addDefinition($appDefinition);

// now, pass in properties
$im = $di->getInstanceManager();

// this could come from Zend\Config\Config::toArray
$propertiesFromConfig = array(
    'ThirdParty\Dbal\DbAdapter' => array(
        'username' => 'someUsername',
        'password' => 'somePassword'
    ),
    'Zend\Controller\Helper\ContentType' => array(
        'default' => 'xhtml5'
    ),
);
$im->setProperties($propertiesFromConfig);
```

13.8 Generating Service Locators

In production, you want things to be as fast as possible. The Dependency Injection Container, while engineered for speed, still must do a fair bit of work resolving parameters and dependencies at runtime. What if you could speed things up and remove those lookups?

The `Zend\Di\ServiceLocator\Generator` component can do just that. It takes a configured DI instance, and generates a service locator class for you from it. That class will manage instances for you, as well as provide hard-coded, lazy-loading instantiation of instances.

The method `getCodeGenerator()` returns an instance of `Zend\CodeGenerator\Php\PhpFile`, from which you can then write a class file with the new Service Locator. Methods on the `Generator` class allow you to specify the namespace and class for the generated Service Locator.

As an example, consider the following:

```
use Zend\Di\ServiceLocator\Generator;

// $di is a fully configured DI instance
$generator = new Generator($di);

$generator->setNamespace('Application')
          ->setContainerClass('Context');
$file = $generator->getCodeGenerator();
$file->setFilename(__DIR__ . '/../Application/Context.php');
$file->write();
```

The above code will write to `../Application/Context.php`, and that file will contain the class `Application\Context`. That file might look like the following:

```
<?php

namespace Application;

use Zend\Di\ServiceLocator;

class Context extends ServiceLocator
{

    public function get($name, array $params = array())
    {
        switch ($name) {
            case 'composed':
            case 'My\ComposedClass':
                return $this->getMyComposedClass();

            case 'struct':
            case 'My\Struct':
                return $this->getMyStruct();

            default:
                return parent::get($name, $params);
        }
    }

    public function getComposedClass()
    {
        if (isset($this->services['My\ComposedClass'])) {
            return $this->services['My\ComposedClass'];
        }

        $object = new \My\ComposedClass();
        $this->services['My\ComposedClass'] = $object;
        return $object;
    }
    public function getMyStruct()
    {
        if (isset($this->services['My\Struct'])) {
            return $this->services['My\Struct'];
        }

        $object = new \My\Struct();
        $this->services['My\Struct'] = $object;
        return $object;
```

```
    }

    public function getComposed()
    {
        return $this->get('My\ComposedClass');
    }

    public function getStruct()
    {
        return $this->get('My\Struct');
    }
}
```

To use this class, you simply consume it as you would a DI container:

```
$container = new Application\Context;

$struct = $container->get('struct'); // My\Struct instance
```

One note about this functionality in its current incarnation. Configuration is per-environment only at this time. This means that you will need to generate a container per execution environment. Our recommendation is that you do so, and then in your environment, specify the container class to use.

CHAPTER

FOURTEEN

UNIT TESTING A ZEND FRAMEWORK 2 APPLICATION

A solid unit test suite is essential for ongoing development in large projects, especially those with many people involved. Going back and manually testing every individual component of an application after every change is impractical. Your unit tests will help alleviate that by automatically testing your application's components and alerting you when something is not working the same way it was when you wrote your tests.

This tutorial is written in the hopes of showing how to test different parts of a Zend Framework 2 MVC application. As such, this tutorial will use the application written in the *getting started user guide*. It is in no way a guide to unit testing in general, but is here only to help overcome the initial hurdles in writing unit tests for ZF2 applications.

It is recommended to have at least a basic understanding of unit tests, assertions and mocks.

As the Zend Framework 2 API uses PHPUnit, so will this tutorial. This tutorial assumes that you already have PHPUnit installed. The version of PHPUnit used should be 3.7.*

14.1 Setting up the tests directory

As Zend Framework 2 applications are built from modules that should be standalone blocks of an application, we don't test the application in it's entirety, but module by module.

We will show how to set up the minimum requirements to test a module, the `Album` module we wrote in the user guide, and which then can be used as a base for testing any other module.

Start by creating a directory called `test` in `zf2-tutorial\module\Album` with the following subdirectories:

```
zf2-tutorial/
    /module
        /Album
            /test
                /AlbumTest
                    /Controller
```

The structure of the `test` directory matches exactly with that of the module's source files, and it will allow you to keep your tests well-organized and easy to find.

14.2 Bootstrapping your tests

Next, create a file called `phpunit.xml` under `zf2-tutorial/module/Album/test`:

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit bootstrap="Bootstrap.php" colors="true">
    <testsuites>
        <testsuite name="zf2tutorial">
            <directory>./AlbumTest</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

And a file called `Bootstrap.php`, also under `zf2-tutorial/module/Album/test`:

```
<?php

namespace AlbumTest;

use Zend\Loader\AutoloaderFactory;
use Zend\Mvc\Service\ServiceManagerConfig;
use Zend\ServiceManager\ServiceManager;
use RuntimeException;

error_reporting(E_ALL | E_STRICT);
chdir(__DIR__);

/**
 * Test bootstrap, for setting up autoloading
 */
class Bootstrap
{
    protected static $serviceManager;

    public static function init()
    {
        $zf2ModulePaths = array(dirname(dirname(__DIR__)));
        if (($path = static::findParentPath('vendor'))) {
            $zf2ModulePaths[] = $path;
        }
        if (($path = static::findParentPath('module')) !== $zf2ModulePaths[0]) {
            $zf2ModulePaths[] = $path;
        }

        static::initAutoloader();

        // use ModuleManager to load this module and it's dependencies
        $config = array(
            'module_listener_options' => array(
                'module_paths' => $zf2ModulePaths,
            ),
            'modules' => array(
                'Album'
            )
        );

        $serviceManager = new ServiceManager(new ServiceManagerConfig());
        $serviceManager->setService('ApplicationConfig', $config);
        $serviceManager->get('ModuleManager')->loadModules();
        static::$serviceManager = $serviceManager;
    }
```

```
47
48      public static function getServiceManager()
49      {
50          return static::$serviceManager;
51      }
52
53      protected static function initAutoloader()
54      {
55          $vendorPath = static::findParentPath('vendor');
56
57          $zf2Path = getenv('ZF2_PATH');
58          if (!$zf2Path) {
59              if (defined('ZF2_PATH')) {
60                  $zf2Path = ZF2_PATH;
61              } elseif (is_dir($vendorPath . '/ZF2/library')) {
62                  $zf2Path = $vendorPath . '/ZF2/library';
63              } elseif (is_dir($vendorPath . '/zendframework/zendframework/library')) {
64                  $zf2Path = $vendorPath . '/zendframework/zendframework/library';
65              }
66          }
67
68          if (!$zf2Path) {
69              throw new RuntimeException('Unable to load ZF2. Run `php composer.phar install` or define
70          }
71
72          include $zf2Path . '/Zend/Loader/AutoloaderFactory.php';
73          AutoloaderFactory::factory(array(
74              'Zend\Loader\StandardAutoloader' => array(
75                  'autoregister_zf' => true,
76                  'namespaces' => array(
77                      __NAMESPACE__ => __DIR__ . '/' . __NAMESPACE__,
78                  ),
79              ),
80          ));
81      }
82
83      protected static function findParentPath($path)
84      {
85          $dir = __DIR__;
86          $previousDir = '.';
87          while (!is_dir($dir . '/' . $path)) {
88              $dir = dirname($dir);
89              if ($previousDir === $dir) return false;
90              $previousDir = $dir;
91          }
92          return $dir . '/' . $path;
93      }
94  }
95
96  Bootstrap::init();
```

The contents of this bootstrap file can be daunting at first sight, but all it really does is ensuring that all the necessary files are autoloadable for our tests. The most important lines is line 38 on which we say what modules we want to load for our test. In this case we are only loading the `Album` module as it has no dependencies against other modules.

Now, if you navigate to the `zf2-tutorial/module/Album/test/` directory, and run `phpunit`, you should get a similar output to this:

```
PHPUnit 3.7.13 by Sebastian Bergmann.

Configuration read from /var/www/zf2-tutorial/module/Album/test/phpunit.xml

Time: 0 seconds, Memory: 1.75Mb

No tests executed!
```

Even though no tests were executed, we at least know that the autoloader found the ZF2 files, otherwise it would throw a `RuntimeException`, defined on line 69 of our bootstrap file.

14.3 Your first controller test

Testing controllers is never an easy task, but Zend Framework 2 comes with the `Zend\Test` component which should make testing much less cumbersome.

First, create `IndexControllerTest.php` under `zf2-tutorial/module/Album/test/AlbumTest/Controller` with the following contents:

```php
<?php

namespace AlbumTest\Controller;

use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;

class AlbumControllerTest extends AbstractHttpControllerTestCase
{
    public function setUp()
    {
        $this->setApplicationConfig(
            include '/var/www/zf2-tutorial/config/application.config.php'
        );
        parent::setUp();
    }
}
```

The `AbstractHttpControllerTestCase` class we extend here helps us setting up the application itself, helps with dispatching and other tasks that happen during a request, as well offers methods for asserting request params, response headers, redirects and more. See *Zend\Test* documentation for more.

One thing that is needed is to set the application config with the `setApplicationConfig` method.

Now, add the following function to the `AlbumControllerTest` class:

```php
public function testIndexActionCanBeAccessed()
{
    $this->dispatch('/album');
    $this->assertResponseStatusCode(200);

    $this->assertModuleName('Album');
    $this->assertControllerName('Album\Controller\Album');
    $this->assertControllerClass('AlbumController');
    $this->assertMatchedRouteName('album');
}
```

This test case dispatches the `/album` URL, asserts that the response code is 200, and that we ended up in the desired module and controller.

Note: For asserting the *controller name* we are using the controller name we defined in our routing configuration for the Album module. In our example this should be defined on line 19 of the `module.config.php` file in the Album module.

14.4 A failing test case

Finally, `cd` to `zf2-tutorial/module/Album/test/` and run `phpunit`. Uh-oh! The test failed!

```
PHPUnit 3.7.13 by Sebastian Bergmann.

Configuration read from /var/www/zf2-tutorial/module/Album/test/phpunit.xml

F

Time: 0 seconds, Memory: 8.50Mb

There was 1 failure:

1) AlbumTest\Controller\AlbumControllerTest::testIndexActionCanBeAccessed
Failed asserting response code "200", actual status code is "500"

/var/www/zf2-tutorial/vendor/ZF2/library/Zend/Test/PHPUnit/Controller/AbstractControllerTestCase.php
/var/www/zf2-tutorial/module/Album/test/AlbumTest/Controller/AlbumControllerTest.php:22

FAILURES!
Tests: 1, Assertions: 0, Failures: 1.
```

The failure message doesn't tell us much, apart from that the expected status code is not 200, but 500. To get a bit more information when something goes wrong in a test case, we set the protected `$traceError` member to `true`. Add the following just above the `setUp` method in our `AlbumControllerTest` class:

```
protected $traceError = true;
```

Running the `phpunit` command again and we should see some more information about what went wrong in our test. The main error message we are interested in should read something like:

```
Zend\ServiceManager\Exception\ServiceNotFoundException: Zend\ServiceManager\ServiceManager::get
was unable to fetch or create an instance for Zend\Db\Adapter\Adapter
```

From this error message it is clear that not all our dependencies are available in the service manager. Let us take a look how can we fix this.

14.5 Configuring the service manager for the tests

The error says that the service manager can not create an instance of a database adapter for us. The database adapter is indirectly used by our `Album\Model\AlbumTable` to fetch the list of albums from the database.

The first thought would be to create an instance of an adapter, pass it to the service manager and let the code run from there as is. Problem with this approach is that we would end up with our test cases doing actually queries against the database. To keep our tests fast, and to reduce the number of possible failure points in our tests, this should be avoided.

The second thought would be then to create a mock of the database adapter, and prevent the actual database calls by mocking them out. This is a much better approach, but creating the adapter mock is tedious (but no doubt we will have to create it at one point).

The best thing to do would be to mock out our `Album\Model\AlbumTable` class which retrieves the list of albums from the database. Remember, we are now testing our controller, so we can mock out the actual call to `fetchAll` and replace the return values with dummy values. At this point, we are not interested in how does `fetchAll` retrieve the albums, but only that it gets called and that it returns an array of albums, so that is why we can get away with this mocking. When we will test `AlbumTable` itself, then we will write the actual tests for the `fetchAll` method.

Here is how we can accomplish this, by modifying the `testIndexActionCanBeAccessed` test method as follows:

```
1  public function testIndexActionCanBeAccessed()
2  {
3      $albumTableMock = $this->getMockBuilder('Album\Model\AlbumTable')
4                              ->disableOriginalConstructor()
5                              ->getMock();
6
7      $albumTableMock->expects($this->once())
8                     ->method('fetchAll')
9                     ->will($this->returnValue(array()));
10
11     $serviceManager = $this->getApplicationServiceLocator();
12     $serviceManager->setAllowOverride(true);
13     $serviceManager->setService('Album\Model\AlbumTable', $albumTableMock);
14
15     $this->dispatch('/album');
16     $this->assertResponseStatusCode(200);
17
18     $this->assertModuleName('Album');
19     $this->assertControllerName('Album\Controller\Album');
20     $this->assertControllerClass('AlbumController');
21     $this->assertMatchedRouteName('album');
22 }
```

By default, the Service Manager does not allow us to replace existing services. As the `Album\Model\AlbumTable` was already set, we are allowing for overrides (line 12), and then replacing the real instance of the *AlbumTable* with a mock. The mock is created so that it will return just an empty array when the `fetchAll` method is called. This allows us to test for what we care about in this test, and that is that by dispatching to the `/album` URL we get to the *Album* module's *AlbumController*.

Running the `phpunit` command at this point, we will get the following output as the tests now pass:

```
PHPUnit 3.7.13 by Sebastian Bergmann.

Configuration read from /var/www/zf2-tutorial/module/Album/test/phpunit.xml

.

Time: 0 seconds, Memory: 9.00Mb

OK (1 test, 6 assertions)
```

14.6 Testing actions with POST

One of the most common actions happening in controllers is submitting a form with some POST data. Testing this is surprisingly easy:

```
public function testAddActionRedirectsAfterValidPost()
{
    $albumTableMock = $this->getMockBuilder('Album\Model\AlbumTable')
                           ->disableOriginalConstructor()
                           ->getMock();

    $albumTableMock->expects($this->once())
                   ->method('saveAlbum')
                   ->will($this->returnValue(null));

    $serviceManager = $this->getApplicationServiceLocator();
    $serviceManager->setAllowOverride(true);
    $serviceManager->setService('Album\Model\AlbumTable', $albumTableMock);

    $postData = array('title' => 'Led Zeppelin III', 'artist' => 'Led Zeppelin');
    $this->dispatch('/album/add', 'POST', $postData);
    $this->assertResponseStatusCode(302);

    $this->assertRedirectTo('/album');
}
```

Here we test that when we make a POST request against the `/album/add` URL, the `Album\Model\AlbumTable`'s `saveAlbum` will be called and after that we will be redirected back to the `/album` URL.

Running `phpunit` gives us the following output:

```
PHPUnit 3.7.13 by Sebastian Bergmann.

Configuration read from /home/robert/www/zf2-tutorial/module/Album/test/phpunit.xml

..

Time: 0 seconds, Memory: 10.75Mb

OK (2 tests, 9 assertions)
```

Testing the `editAction` and `deleteAction` methods can be easily done in a manner similar as shown for the `addAction`.

14.7 Testing model entities

Now that we know how to test our controllers, let us move to an other important part of our application - the model entity.

Here we want to test that the initial state of the entity is what we expect it to be, that we can convert the model's parameters to and from an array, and that it has all the input filters we need.

Create the file `AlbumTest.php` in `module/Album/test/AlbumTest/Model` directory with the following contents:

```php
<?php
namespace AlbumTest\Model;

use Album\Model\Album;
use PHPUnit_Framework_TestCase;

class AlbumTest extends PHPUnit_Framework_TestCase
{
    public function testAlbumInitialState()
    {
        $album = new Album();

        $this->assertNull($album->artist, '"artist" should initially be null');
        $this->assertNull($album->id, '"id" should initially be null');
        $this->assertNull($album->title, '"title" should initially be null');
    }

    public function testExchangeArraySetsPropertiesCorrectly()
    {
        $album = new Album();
        $data  = array('artist' => 'some artist',
                       'id'     => 123,
                       'title'  => 'some title');

        $album->exchangeArray($data);

        $this->assertSame($data['artist'], $album->artist, '"artist" was not set correctly');
        $this->assertSame($data['id'], $album->id, '"id" was not set correctly');
        $this->assertSame($data['title'], $album->title, '"title" was not set correctly');
    }

    public function testExchangeArraySetsPropertiesToNullIfKeysAreNotPresent()
    {
        $album = new Album();

        $album->exchangeArray(array('artist' => 'some artist',
                                    'id'     => 123,
                                    'title'  => 'some title'));
        $album->exchangeArray(array());

        $this->assertNull($album->artist, '"artist" should have defaulted to null');
        $this->assertNull($album->id, '"id" should have defaulted to null');
        $this->assertNull($album->title, '"title" should have defaulted to null');
    }

    public function testGetArrayCopyReturnsAnArrayWithPropertyValues()
    {
        $album = new Album();
        $data  = array('artist' => 'some artist',
                       'id'     => 123,
                       'title'  => 'some title');

        $album->exchangeArray($data);
        $copyArray = $album->getArrayCopy();

        $this->assertSame($data['artist'], $copyArray['artist'], '"artist" was not set correctly');
        $this->assertSame($data['id'], $copyArray['id'], '"id" was not set correctly');
        $this->assertSame($data['title'], $copyArray['title'], '"title" was not set correctly');
```

```
    }

    public function testInputFiltersAreSetCorrectly()
    {
        $album = new Album();

        $inputFilter = $album->getInputFilter();

        $this->assertSame(3, $inputFilter->count());
        $this->assertTrue($inputFilter->has('artist'));
        $this->assertTrue($inputFilter->has('id'));
        $this->assertTrue($inputFilter->has('title'));
    }
}
```

We are testing for 5 things:

1. Are all of the Album's properties initially set to NULL?
2. Will the Album's properties be set correctly when we call `exchangeArray()`?
3. Will a default value of NULL be used for properties whose keys are not present in the `$data` array?
4. Can we get an array copy of our model?
5. Do all elements have input filters present?

If we run `phpunit` again, we will get the following output, confirming that our model is indeed correct:

```
PHPUnit 3.7.13 by Sebastian Bergmann.

Configuration read from /var/www/zf2-tutorial/module/Album/test/phpunit.xml

.......

Time: 0 seconds, Memory: 11.00Mb

OK (7 tests, 25 assertions)
```

14.8 Testing model tables

The final step in this unit testing tutorial for Zend Framework 2 applications is writing tests for our model tables.

This test assures that we can get a list of albums, or one album by it's ID, and that we can save and delete albums from the database.

To avoid actual interaction with the database itself, we will replace certain parts with *mocks*.

Create a file `AlbumTableTest.php` in `module/Album/test/AlbumTest/Model` with the following contents:

```
<?php
namespace AlbumTest\Model;

use Album\Model\AlbumTable;
use Album\Model\Album;
use Zend\Db\ResultSet\ResultSet;
use PHPUnit_Framework_TestCase;
```

```
class AlbumTableTest extends PHPUnit_Framework_TestCase
{
    public function testFetchAllReturnsAllAlbums()
    {
        $resultSet = new ResultSet();
        $mockTableGateway = $this->getMock('Zend\Db\TableGateway\TableGateway',
                                           array('select'), array(), '', false);
        $mockTableGateway->expects($this->once())
                         ->method('select')
                         ->with()
                         ->will($this->returnValue($resultSet));

        $albumTable = new AlbumTable($mockTableGateway);

        $this->assertSame($resultSet, $albumTable->fetchAll());
    }
}
```

Since we are testing the `AlbumTable` here and not the `TableGateway` class (which has already been tested in Zend Framework), we just want to make sure that our `AlbumTable` class is interacting with the `TableGateway` class the way that we expect it to. Above, we're testing to see if the `fetchAll()` method of `AlbumTable` will call the `select()` method of the `$tableGateway` property with no parameters. If it does, it should return a `ResultSet` object. Finally, we expect that this same `ResultSet` object will be returned to the calling method. This test should run fine, so now we can add the rest of the test methods:

```
public function testCanRetrieveAnAlbumByItsId()
{
    $album = new Album();
    $album->exchangeArray(array('id'     => 123,
                                'artist' => 'The Military Wives',
                                'title'  => 'In My Dreams'));

    $resultSet = new ResultSet();
    $resultSet->setArrayObjectPrototype(new Album());
    $resultSet->initialize(array($album));

    $mockTableGateway = $this->getMock('Zend\Db\TableGateway\TableGateway', array('select'), array(),
    $mockTableGateway->expects($this->once())
                     ->method('select')
                     ->with(array('id' => 123))
                     ->will($this->returnValue($resultSet));

    $albumTable = new AlbumTable($mockTableGateway);

    $this->assertSame($album, $albumTable->getAlbum(123));
}

public function testCanDeleteAnAlbumByItsId()
{
    $mockTableGateway = $this->getMock('Zend\Db\TableGateway\TableGateway', array('delete'), array(),
    $mockTableGateway->expects($this->once())
                     ->method('delete')
                     ->with(array('id' => 123));

    $albumTable = new AlbumTable($mockTableGateway);
    $albumTable->deleteAlbum(123);
}
```

```
public function testSaveAlbumWillInsertNewAlbumsIfTheyDontAlreadyHaveAnId()
{
    $albumData = array('artist' => 'The Military Wives', 'title' => 'In My Dreams');
    $album     = new Album();
    $album->exchangeArray($albumData);

    $mockTableGateway = $this->getMock('Zend\Db\TableGateway\TableGateway', array('insert'), array(),
    $mockTableGateway->expects($this->once())
                     ->method('insert')
                     ->with($albumData);

    $albumTable = new AlbumTable($mockTableGateway);
    $albumTable->saveAlbum($album);
}

public function testSaveAlbumWillUpdateExistingAlbumsIfTheyAlreadyHaveAnId()
{
    $albumData = array('id' => 123, 'artist' => 'The Military Wives', 'title' => 'In My Dreams');
    $album     = new Album();
    $album->exchangeArray($albumData);

    $resultSet = new ResultSet();
    $resultSet->setArrayObjectPrototype(new Album());
    $resultSet->initialize(array($album));

    $mockTableGateway = $this->getMock('Zend\Db\TableGateway\TableGateway',
                                       array('select', 'update'), array(), '', false);
    $mockTableGateway->expects($this->once())
                     ->method('select')
                     ->with(array('id' => 123))
                     ->will($this->returnValue($resultSet));
    $mockTableGateway->expects($this->once())
                     ->method('update')
                     ->with(array('artist' => 'The Military Wives', 'title' => 'In My Dreams'),
                            array('id' => 123));

    $albumTable = new AlbumTable($mockTableGateway);
    $albumTable->saveAlbum($album);
}

public function testExceptionIsThrownWhenGettingNonExistentAlbum()
{
    $resultSet = new ResultSet();
    $resultSet->setArrayObjectPrototype(new Album());
    $resultSet->initialize(array());

    $mockTableGateway = $this->getMock('Zend\Db\TableGateway\TableGateway', array('select'), array(),
    $mockTableGateway->expects($this->once())
                     ->method('select')
                     ->with(array('id' => 123))
                     ->will($this->returnValue($resultSet));

    $albumTable = new AlbumTable($mockTableGateway);

    try {
        $albumTable->getAlbum(123);
    }
    catch (\Exception $e) {
```

```
        $this->assertSame('Could not find row 123', $e->getMessage());
        return;
    }

    $this->fail('Expected exception was not thrown');
}
```

These tests are nothing complicated and they should be self explanatory. In each test we are injecting a mock table gateway into our `AlbumTable` and set our expectations accordingly.

We are testing that:

1. We can retrieve an individual album by its ID.
2. We can delete albums.
3. We can save new album.
4. We can update existing albums.
5. We will encounter an exception if we're trying to retrieve an album that doesn't exist.

Running `phpunit` command for one last time, we get the output as follows:

```
PHPUnit 3.7.13 by Sebastian Bergmann.

Configuration read from /var/www/zf2-tutorial/module/Album/test/phpunit.xml

.............

Time: 0 seconds, Memory: 11.50Mb

OK (13 tests, 34 assertions)
```

14.9 Conclusion

In this short tutorial we gave a few examples how different parts of a Zend Framework 2 MVC application can be tested. We covered *setting up* the environment for testing, how to test *controllers and actions*, how to approach *failing test cases*, how to configure *the service manager*, as well as how to test *model entities* and *model tables*.

This tutorial is by no means a definitive guide to writing unit tests, just a small stepping stone helping you develop applications of higher quality.

CHAPTER

FIFTEEN

USING THE EVENTMANAGER

This tutorial explores the various features of `Zend\EventManager`.

15.1 Terminology

- An **Event** is a named action.
- A **Listener** is any PHP callback that reacts to an *event*.
- An **EventManager** *aggregates* listeners for one or more named events, and *triggers* events.

Typically, an *event* will be modeled as an object, containing metadata surrounding when and how it was triggered, including the event name, what object triggered the event (the "target"), and what parameters were provided. Events are *named*, which allows a single *listener* to branch logic based on the event.

15.2 Getting started

The minimal things necessary to start using events are:

- An `EventManager` instance
- One or more listeners on one or more events
- A call to `trigger()` an event

The simplest example looks something like this:

```
use Zend\EventManager\EventManager;

$events = new EventManager();
$events->attach('do', function ($e) {
    $event = $e->getName();
    $params = $e->getParams();
    printf(
        'Handled event "%s", with parameters %s',
        $event,
        json_encode($params)
    );
});

$params = array('foo' => 'bar', 'baz' => 'bat');
$events->trigger('do', null, $params);
```

The above will result in the following:

```
Handled event "do", with parameters {"foo":"bar","baz":"bat"}
```

Note: Throughout this tutorial, we use closures as listeners. However, any valid PHP callback can be attached as a listeners: PHP function names, static class methods, object instance methods, functors, or closures. We use closures within this post simply for illustration and simplicity.

If you were paying attention to the example, you will have noted the `null` argument. Why is it there?

Typically, you will compose an `EventManager` within a class, to allow triggering actions within methods. The middle argument to `trigger()` is the "target", and in the case described, would be the current object instance. This gives event listeners access to the calling object, which can often be useful.

```
use Zend\EventManager\EventManager;
use Zend\EventManager\EventManagerAwareInterface;
use Zend\EventManager\EventManagerInterface;

class Example implements EventManagerAwareInterface
{
    protected $events;

    public function setEventManager(EventManagerInterface $events)
    {
        $events->setIdentifiers(array(
            __CLASS__,
            get_class($this)
        ));
        $this->events = $events;
    }

    public function getEventManager()
    {
        if (!$this->events) {
            $this->setEventManager(new EventManager());
        }
        return $this->events;
    }

    public function do($foo, $baz)
    {
        $params = compact('foo', 'baz');
        $this->getEventManager()->trigger(__FUNCTION__, $this, $params);
    }

}

$example = new Example();

$example->getEventManager()->attach('do', function($e) {
    $event  = $e->getName();
    $target = get_class($e->getTarget()); // "Example"
    $params = $e->getParams();
    printf(
        'Handled event "%s" on target "%s", with parameters %s',
        $event,
        $target,
        json_encode($params)
```

```
    );
});

$example->do('bar', 'bat');
```

The above is basically the same as the first example. The main difference is that we're now using that middle argument in order to pass the target, the instance of `Example`, on to the listeners. Our listener is now retrieving that (`$e->getTarget()`), and doing something with it.

If you're reading this critically, you should have a new question: What is the call to `setIdentifiers()` for?

15.3 Shared managers

One aspect that the `EventManager` implementation provides is an ability to compose a `SharedEventManagerInterface` implementation.

`Zend\EventManager\\SharedEventManagerInterface` describes an object that aggregates listeners for events attached to objects with specific *identifiers*. It does not trigger events itself. Instead, an `EventManager` instance that composes a `SharedEventManager` will query the `SharedEventManager` for listeners on identifiers it's interested in, and trigger those listeners as well.

How does this work, exactly?

Consider the following:

```
use Zend\EventManager\SharedEventManager;

$sharedEvents = new SharedEventManager();
$sharedEvents->attach('Example', 'do', function ($e) {
    $event  = $e->getName();
    $target = get_class($e->getTarget()); // "Example"
    $params = $e->getParams();
    printf(
        'Handled event "%s" on target "%s", with parameters %s',
        $event,
        $target,
        json_encode($params)
    );
});
```

This looks almost identical to the previous example; the key difference is that there is an additional argument at the *start* of the list, `'Example'`. This code is basically saying, "Listen to the 'do' event of the 'Example' target, and, when notified, execute this callback."

This is where the `setIdentifiers()` argument of `EventManager` comes into play. The method allows passing a string, or an array of strings, defining the name or names of the context or targets the given instance will be interested in. If an array is given, then any listener on any of the targets given will be notified.

So, getting back to our example, let's assume that the above shared listener is registered, and also that the `Example` class is defined as above. We can then execute the following:

```
$example = new Example();
$example->getEventManager()->setSharedManager($sharedEvents);
$example->do('bar', 'bat');
```

and expect the following to be `echo`'d:

```
Handled event "do" on target "Example", with parameters {"foo":"bar","baz":"bat"}
```

Now, let's say we extended `Example` as follows:

```
class SubExample extends Example
{
}
```

One interesting aspect of our `setEventManager()` method is that we defined it to listen both on `__CLASS__` and `get_class($this)`. This means that calling `do()` on our `SubExample` class would also trigger the shared listener! It also means that, if desired, we could attach to specifically `SubExample`, and listeners attached to only the `Example` target would not be triggered.

Finally, the names used as contexts or targets need not be class names; they can be some name that only has meaning in your application if desired. As an example, you could have a set of classes that respond to "log" or "cache" – and listeners on these would be notified by any of them.

Note: We recommend using class names, interface names, and/or abstract class names for identifiers. This makes determining what events are available easier, as well as finding which listeners might be attaching to those events. Interfaces make a particularly good use case, as they allow attaching to a group of related classes a single operation.

At any point, if you do not want to notify shared listeners, pass a `null` value to `setSharedManager()`:

and they will be ignored. If at any point, you want to enable them again, pass the `SharedEventManager` instance:

CHAPTER

SIXTEEN

WILDCARDS

So far, with both a normal `EventManager` instance and with the `SharedEventManager` instance, we've seen the usage of singular strings representing the event and target names to which we want to attach. What if you want to attach a listener to multiple events or targets?

The answer is to supply an array of events or targets, or a wildcard, `*`.

Consider the following examples:

```
// Multiple named events:
$events->attach(
    array('foo', 'bar', 'baz'), // events
    $listener
);

// All events via wildcard:
$events->attach(
    '*', // all events
    $listener
);

// Multiple named targets:
$sharedEvents->attach(
    array('Foo', 'Bar', 'Baz'), // targets
    'doSomething', // named event
    $listener
);

// All targets via wildcard
$sharedEvents->attach(
    '*', // all targets
    'doSomething', // named event
    $listener
);

// Mix and match: multiple named events on multiple named targets:
$sharedEvents->attach(
    array('Foo', 'Bar', 'Baz'), // targets
    array('foo', 'bar', 'baz'), // events
    $listener
);

// Mix and match: all events on multiple named targets:
$sharedEvents->attach(
    array('Foo', 'Bar', 'Baz'), // targets
```

```
    '*', // events
    $listener
);

// Mix and match: multiple named events on all targets:
$sharedEvents->attach(
    '*', // targets
    array('foo', 'bar', 'baz'), // events
    $listener
);

// Mix and match: all events on all targets:
$sharedEvents->attach(
    '*', // targets
    '*', // events
    $listener
);
```

The ability to specify multiple targets and/or events when attaching can slim down your code immensely.

CHAPTER

SEVENTEEN

LISTENER AGGREGATES

Another approach to listening to multiple events is via a concept of listener aggregates, represented by `Zend\EventManager\ListenerAggregateInterface`. Via this approach, a single class can listen to multiple events, attaching one or more instance methods as listeners.

This interface defines two methods, `attach(EventManagerInterface $events)` and `detach(EventManagerInterface $events)`. Basically, you pass an `EventManager` instance to one and/or the other, and then it's up to the implementing class to determine what to do.

As an example:

```
use Zend\EventManager\EventInterface;
use Zend\EventManager\EventManagerInterface;
use Zend\EventManager\ListenerAggregateInterface;
use Zend\Log\Logger;

class LogEvents implements ListenerAggregateInterface
{
    protected $listeners = array();
    protected $log;

    public function __construct(Logger $log)
    {
        $this->log = $log;
    }

    public function attach(EventManagerInterface $events)
    {
        $this->listeners[] = $events->attach('do', array($this, 'log'));
        $this->listeners[] = $events->attach('doSomethingElse', array($this, 'log'));
    }

    public function detach(EventCollection $events)
    {
        foreach ($this->listeners as $index => $listener) {
            if ($events->detach($listener)) {
                unset($this->listeners[$index];
            }
        }
    }

    public function log(EventInterface $e)
    {
        $event  = $e->getName();
        $params = $e->getParams();
```

```
        $this->log->info(sprintf('%s: %s', $event, json_encode($params)));
    }
}
```

You can attach this using either `attach()` or `attachAggregate()`:

Any events the aggregate attaches to will then be notified when triggered.

Why bother? For a couple of reasons:

- Aggregates allow you to have stateful listeners. The above example demonstrates this via the composition of the logger; another example would be tracking configuration options.
- Aggregates make detaching listeners easier. When you call `attach()` normally, you receive a `Zend\Stdlib\CallbackHandler` instance; the only way to `detach()` a listener is to pass that instance back – which means if you want to detach later, you need to keep that instance somewhare. Aggregates typically do this for you – as you can see in the example above.

17.1 Introspecting results

Sometimes you'll want to know what your listeners returned. One thing to remember is that you may have multiple listeners on the same event; the interface for results must be consistent regardless of the number of listeners.

The `EventManager` implementation by default returns a `Zend\EventManager\ResponseCollection` instance. This class extends PHP's `SplStack`, allowing you to loop through responses in reverse order (since the last one executed is likely the one you're most interested in). It also implements the following methods:

- `first()` will retrieve the first result received
- `last()` will retrieve the last result received
- `contains($value)` allows you to test all values to see if a given one was received, and returns simply a boolean `true` if found, and `false` if not.

Typically, you should not worry about the return values from events, as the object triggering the event shouldn't really have much insight into what listeners are attached. However, sometimes you may want to short-circuit execution if interesting results are obtained.

17.2 Short-ciruiting listener execution

You may want to short-ciruit execution if a particular result is obtained, or if a listener determines that something is wrong, or that it can return something quicker than the target.

As examples, one rationale for adding an `EventManager` is as a caching mechanism. You can trigger one event early in the method, returning if a cache is found, and trigger another event late in the method, seeding the cache.

The `EventManager` component offers two ways to handle this. The first is to pass a callback as the last argument to `trigger()`; if that callback returns a boolean `true`, execution is halted.

Here's an example:

```
public function someExpensiveCall($criteria1, $criteria2)
{
    $params  = compact('criteria1', 'criteria2');
    $results = $this->getEventManager()->trigger(
        __FUNCTION__,
        $this,
```

```
        $params,
        function ($r) {
            return ($r instanceof SomeResultClass);
        }
    );
    if ($results->stopped()) {
        return $results->last();
    }

    // ... do some work ...
}
```

With this paradigm, we know that the likely reason of execution halting is due to the last result meeting the test callback criteria; as such, we simply return that last result.

The other way to halt execution is within a listener, acting on the `Event` object it receives. In this case, the listener calls `stopPropagation(true)`, and the `EventManager` will then return without notifying any additional listeners.

```
$events->attach('do', function ($e) {
    $e->stopPropagation();
    return new SomeResultClass();
});
```

This, of course, raises some ambiguity when using the trigger paradigm, as you can no longer be certain that the last result meets the criteria it's searching on. As such, we recommend that you standardize on one approach or the other.

17.3 Keeping it in order

On occasion, you may be concerned about the order in which listeners execute. As an example, you may want to do any logging early, to ensure that if short-circuiting occurs, you've logged; or if implementing a cache, you may want to return early if a cache hit is found, and execute late when saving to a cache.

Each of `EventManager::attach()` and `SharedEentManager::attach()` accept one additional argument, a *priority*. By default, if this is omitted, listeners get a priority of 1, and are executed in the order in which they are attached. However, if you provide a priority value, you can influence order of execution.

- Higher priority values execute *earlier*.
- Lower (negative) priority values execute *later*.

To borrow an example from earlier:

```
$priority = 100;
$events->attach('Example', 'do', function($e) {
    $event  = $e->getName();
    $target = get_class($e->getTarget()); // "Example"
    $params = $e->getParams();
    printf(
        'Handled event "%s" on target "%s", with parameters %s',
        $event,
        $target,
        json_encode($params)
    );
}, $priority);
```

This would execute with high priority, meaning it would execute early. If we changed `$priority` to `-100`, it would execute with low priority, executing late.

While you can't necessarily know all the listeners attached, chances are you can make adequate guesses when necessary in order to set appropriate priority values. We advise avoiding setting a priority value unless absolutely necessary.

17.4 Custom event objects

Hopefully some of you have been wondering, "where and when is the Event object created"? In all of the examples above, it's created based on the arguments passed to `trigger()` – the event name, target, and parameters. Sometimes, however, you may want greater control over the object.

As an example, one thing that looks like a code smell is when you have code like this:

```
$routeMatch = $e->getParam('route-match', false);
if (!$routeMatch) {
    // Oh noes! we cannot do our work! whatever shall we do?!?!?!
}
```

The problems with this are several. First, relying on string keys is going to very quickly run into problems – typos when setting or retrieving the argument can lead to hard to debug situations. Second, we now have a documentation issue; how do we document expected arguments? how do we document what we're shoving into the event? Third, as a side effect, we can't use IDE or editor hinting support – string keys give these tools nothing to work with.

Similarly, consider how you might represent a computational result of a method when triggering an event. As an example:

```
// in the method:
$params['__RESULT'] = $computedResult;
$events->trigger(__FUNCTION__ . '.post', $this, $params);

// in the listener:
$result = $e->getParam('__RESULT__');
if (!$result) {
    // Oh noes! we cannot do our work! whatever shall we do?!?!?!
}
```

Sure, that key may be unique, but it suffers from a lot of the same issues.

So, the solution is to create custom events. As an example, we have a custom `MvcEvent` in the ZF2 MVC layer. This event composes the application instance, the router, the route match object, request and response objects, the view model, and also a result. We end up with code like this in our listeners:

```
$response = $e->getResponse();
$result   = $e->getResult();
if (is_string($result)) {
    $content = $view->render('layout.phtml', array('content' => $result));
    $response->setContent($content);
}
```

But how do we use this custom event? Simple: `trigger()` can accept an event object instead of any of the event name, target, or params arguments.

```
$event = new CustomEvent();
$event->setSomeKey($value);

// Injected with event name and target:
$events->trigger('foo', $this, $event);

// Injected with event name:
$event->setTarget($this);
```

```
$events->trigger('foo', $event);

// Fully encapsulates all necessary properties:
$event->setName('foo');
$event->setTarget($this);
$events->trigger($event);

// Passing a callback following the event object works for
// short-circuiting, too.
$results = $events->trigger('foo', $this, $event, $callback);
```

This is a really powerful technique for domain-specific event systems, and definitely worth experimenting with.

17.5 Putting it together: Implementing a simple caching system

In previous sections, I indicated that short-circuiting is a way to potentially implement a caching solution. Let's create a full example.

First, let's define a method that could use caching. You'll note that in most of the examples, I've used `__FUNCTION__` as the event name; this is a good practice, as it makes it simple to create a macro for triggering events, as well as helps to keep event names unique (as they're usually within the context of the triggering class). However, in the case of a caching example, this would lead to identical events being triggered. As such, I recommend postfixing the event name with semantic names: "do.pre", "do.post", "do.error", etc. I'll use that convention in this example.

Additionally, you'll notice that the `$params` I pass to the event is usually the list of parameters passed to the method. This is because those are often not stored in the object, and also to ensure the listeners have the exact same context as the calling method. But it raises an interesting problem in this example: what name do we give the result of the method? One standard that has emerged is the use of `__RESULT__`, as double-underscored variables are typically reserved for the sytem.

Here's what the method will look like:

```
public function someExpensiveCall($criteria1, $criteria2)
{
    $params  = compact('criteria1', 'criteria2');
    $results = $this->getEventManager()->trigger(
        __FUNCTION__ . '.pre',
        $this,
        $params,
        function ($r) {
            return ($r instanceof SomeResultClass);
        }
    );
    if ($results->stopped()) {
        return $results->last();
    }

    // ... do some work ...

    $params['__RESULT__'] = $calculatedResult;
    $this->events()->trigger(__FUNCTION__ . '.post', $this, $params);
    return $calculatedResult;
}
```

Now, to provide some caching listeners. We'll need to attach to each of the "someExpensiveCall.pre" and "someExpensiveCall.post" methods. In the former case, if a cache hit is detected, we return it, and move on. In the latter, we store the value in the cache.

We'll assume `$cache` is defined, and follows the paradigms of `Zend\Cache`. We'll want to return early if a hit is detected, and execute late when saving a cache (in case the result is modified by another listener). As such, we'll set the "someExpensiveCall.pre" listener to execute with priority `100`, and the "someExpensiveCall.post" listener to execute with priority `-100`.

```
$events->attach('someExpensiveCall.pre', function($e) use ($cache) {
    $params = $e->getParams();
    $key    = md5(json_encode($params));
    $hit    = $cache->load($key);
    return $hit;
}, 100);

$events->attach('someExpensiveCall.post', function($e) use ($cache) {
    $params = $e->getParams();
    $result = $params['__RESULT__'];
    unset($params['__RESULT__']);
    $key    = md5(json_encode($params));
    $cache->save($result, $key);
}, -100);
```

Note: The above could have been done within a `ListenerAggregate`, which would have allowed keeping the `$cache` instance as a stateful property, instead of importing it into closures.

Another approach would be to move the body of the method to a listener as well, which would allow using the priority system in order to implement caching. That would look like this:

```
public function setEventManager(EventManagerInterface $events)
{
    $this->events = $events;
    $events->setIdentifiers(array(__CLASS__, get_class($this)));
    $events->attach('someExpensiveCall', array($this, 'doSomeExpensiveCall'));
}

public function someExpensiveCall($criteria1, $criteria2)
{
    $params  = compact('criteria1', 'criteria2');
    $results = $this->getEventManager()->trigger(
        __FUNCTION__,
        $this,
        $params,
        function ($r) {
            return ($r instanceof SomeResultClass);
        }
    );
    return $results->last();
}

public function doSomeExpensiveCall($e)
{
    // ... do some work ...
    $e->setParam('__RESULT__', $calculatedResult);
    return $calculatedResult;
}
```

The listeners would then attach to the "someExpensiveCall" event, with the cache lookup listener listening at high priority, and the cache storage listener listening at low (negative) priority.

Sure, we could probably simply add caching to the object itself - but this approach allows the same handlers to be

attached to multiple events, or to attach multiple listeners to the same events (e.g. an argument validator, a logger and a cache manager). The point is that if you design your object with events in mind, you can easily make it more flexible and extensible, without requiring developers to actually extend it – they can simply attach listeners.

17.6 Conclusion

The `EventManager` is a powerful component. It drives the workflow of the MVC layer, and is used in countless components to provide hook points for developers to manipulate the workflow. It can be put to any number of uses inside your own code, and is an important part of your Zend Framework toolbox.

CHAPTER

EIGHTEEN

USING ZEND\NAVIGATION IN YOUR ALBUM MODULE

In this tutorial we will use `Zend\Navigation` to add a navigation menu to the black bar at the top of the screen, and add breadcrumbs above the main site content.

18.1 Preparation

In a real world application, the album browser would be only a portion of a working website. Usually the user would land on a homepage first, and be able to view albums by using a standard navigation menu. So that we have a site that is more realistic than just the albums feature, lets make the standard skeleton welcome page our homepage, with the */album* route still showing our album module. In order to make this change, we need to undo some work we did earlier. Currently, navigating to the root of your app (/) routes you to the `AlbumController`'s default action. Let's undo this route change so we have two discrete entry points to the app, a home page, and an albums area.

`module/Application/config/module.config.php`

```
'home' => array(
   'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route'    => '/',
        'defaults' => array(
            'controller' => 'Application\Controller\Album', // <-- change back here
            'action'     => 'index',
        ),
    ),
),
```

This change means that if you go to the home page of your application (http://zf2-tutorial.localhost/), you see the default skeleton application introduction. Your list of albums is still available at the /album route.

18.2 Setting Up Zend\Navigation

Firstly, we need to tell our application which `NavigationFactory` to use when using the bundled navigation view helpers. Thankfully, ZF2 comes with a default factory that will suit our needs just fine. To tell ZF2 to use this default factory, we simply add a `navigation` key to the service manager. Its best to do this in the `Application` module, because, like the translation data, this is specific to the entire application, and not just to our album pages:

`module/Application/config/module.config.php`

```
'service_manager' => array(
    'factories' => array(
        'translator' => 'Zend\I18n\Translator\TranslatorServiceFactory',
        'navigation' => 'Zend\Navigation\Service\DefaultNavigationFactory', \\ <-- add this
    ),
),
```

18.3 Configuring our Site Map

Next up, we need `Zend\Navigation` to understand the hierarchy of our site. Thankfully, if we add a `navigation` key to our merged config, the navigation factory will automagically create the container and pages needed to use the view helpers. Let's do this in the `Application` module:

`module/Application/config/module.config.php`

```
return array(
...
'navigation' => array(
    'default' => array(
        array(
            'label' => 'Home',
            'route' => 'home',
        ),
        array(
            'label' => 'Album',
            'route' => 'album',
            'pages' => array(
                array(
                    'label' => 'Add',
                    'route' => 'album',
                    'action' => 'add',
                ),
                array(
                    'label' => 'Edit',
                    'route' => 'album',
                    'action' => 'edit',
                ),
                array(
                    'label' => 'Delete',
                    'route' => 'album',
                    'action' => 'delete',
                ),
            ),
        ),
    ),
),
...
);
```

This configuration maps out the pages we've defined in our controller, with labels linking to the given route names. You can define highly complex hierarchical sites here with pages and sub-pages linking to route names, controller/action pairs or external uris. For more information see the docs here.

18.4 Adding the Menu View Helper

Now that we have the navigation helper configured by our service manager and merged config, we can easily add the menu to the title bar to our layout by using the `menu` view helper:

`module/Application/view/layout/layout.phtml`

```
...
<a class="brand"
       href="<?php echo $this->url('home') ?>"><?php echo $this->translate('Skeleton Application') ?
<?php // <-- Add this !!
echo $this->navigation('navigation')->menu();
?>
...
```

The navigation helper is built in to Zend Framework 2, and uses the service manager configuration we've already defined to configure itself automatically. Refreshing your application you will see a working (but ugly) menu, with just a few tweaks however, we can make it look awesome:

`module/Application/view/layout/layout.phtml`

```
<a class="brand"
   href="<?php echo $this->url('home') ?>"><?php echo $this->translate('Skeleton Application') ?></a>
<?php // <-- Update this !!
echo $this->navigation('navigation')
    ->menu()
    ->setMinDepth(0)
    ->setMaxDepth(0)
    ->setUlClass('nav')
    ->render();
?>
```

Here we tell the renderer to give the root UL the class of 'nav' so that Twitter Bootstrap styles the menu correctly, and only render the first level of any given page. If you view your application in your browser, you will now see a nicely styled menu appear in the title bar. The great thing about `Zend\Navigation` is that it integrates with ZF2's route so can tell which page you are currently viewing. Because of this, it sets the active page to have a class of `active` in the menu. Twitter Bootstrap uses this to highlight your current page accordingly.

18.5 Adding Breadcrumbs

Adding breadcrumbs is initially just as simple. In our `layout.phtml` we want to add breadcrumbs above the main content pane, so our foolish user knows exactly where they are in our complex website. Inside the container div, before we output the content from the view, let's add a simple breadcrumb:

`module/Application/view/layout/layout.phtml`

```
...
<div class="container">
    <?php echo $this->navigation('navigation')->breadcrumbs()->setMinDepth(0); // <-- Add this!! ?>
    <?php echo $this->content; ?>
</div>
...
```

This adds a simple but functional breadcrumb to every page (we simply tell it to render from a depth of 0 so we see all level of pages) but we can do better than that! Because Bootstrap has a styled breadcrumb as part of it's base CSS, so let's add a partial that outputs the UL in bootstrap happy CSS. We'll create it in the `view` directory of the `Application` module (this partial is application wide, rather than album specific):

module/Application/view/partial/breadcrumb.phtml

```
<ul class="breadcrumb">
    <?php
    // iterate through the pages
    foreach ($this->pages as $key => $page):
        ?>
        <li>
            <?php
            // if this isn't the last page, add a link and the separator
            if ($key < count($this->pages) - 1):
                ?>
                <a href="<?php echo $page->getHref(); ?>"><?php echo $page->getLabel(); ?></a>
                <span class="divider">/</span>
            <?php // otherwise, just output the name
            else:
            ?>
                <?php echo $page->getLabel(); ?>
            <?php endif; ?>
        </li>
        <?php endforeach; ?>
</ul>
```

Notice how the partial is passed a `Zend\View\Model\ViewModel` instance with the `pages` property set to an array of pages to render. Now all we have to do is tell the breadcrumb helper to use the partial we have just written:

module/Application/view/layout/layout.phtml

```
...
<div class="container">
    <?php echo $this->navigation('navigation') // <-- Update this!!
    ->breadcrumbs()
    ->setMinDepth(0)
    ->setPartial(array('partial/breadcrumb.phtml', 'Album'));
    ?>
    <?php echo $this->content; ?>
</div>
...
```

Refreshing the page now gives us a lovely styled set of breadcrumbs on each page.

CHAPTER

NINETEEN

USING ZEND\PAGINATOR IN YOUR ALBUM MODULE

In this tutorial we will use `Zend\Paginator` to add a handy pagination controller to the bottom of the album list.

Currently, we only have a handful of albums to display, so showing everything on one page is not a problem. However, how will the album list look when we have 100 albums or more in our database? The standard solution to this problem is to split the data up into a number of pages, and allow the user to navigate around these pages using a pagination control. Just type "Zend Framework" into Google, and you can see their pagination control at the bottom of the page:

19.1 Preparation

In order for us to have lots of albums in our database, you'll need to run the following SQL insert statement to insert the current 150 top iTunes albums (at the time of writing!):

```
INSERT INTO `album` (`artist`, `title`) VALUES ('David Bowie', 'The Next Day (Deluxe Version)'), ('Ba
```

This gives us a handy extra 150 rows to play with. If you now visit your album list at `/album`, you'll see a huge long list of 150+ albums, its ugly.

19.2 Modifying the AlbumTable

In order to let ZF2 handle our database queries automatically for us, we will be using the `Zend\Paginator\Adapter\DbSelect` paginator adapter. This will automatically manipulate and run a `Zend\Db\Sql\Select` object to include the correct `LIMIT` and `WHERE` clauses, so that it returns only the right amount of data needed to display the given page. Let's modify the `fetchAll` method of the `AlbumTable` model, so that it can optionally return a paginator object:

`module/Album/src/Album/Model/AlbumTable.php`

```
<?php
namespace Album\Model;

use Zend\Db\ResultSet\ResultSet;
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\Sql\Select;
use Zend\Paginator\Adapter\DbSelect;
use Zend\Paginator\Paginator;
```

```
class AlbumTable
{
    ...
    public function fetchAll($paginated=false)
    {
        if($paginated) {
            // create a new Select object for the table album
            $select = new Select('album');
            // create a new result set based on the Album entity
            $resultSetPrototype = new ResultSet();
            $resultSetPrototype->setArrayObjectPrototype(new Album());
            // create a new pagination adapter object
            $paginatorAdapter = new DbSelect(
                // our configured select object
                $select,
                // the adapter to run it against
                $this->tableGateway->getAdapter(),
                // the result set to hydrate
                $resultSetPrototype
            );
            $paginator = new Paginator($paginatorAdapter);
            return $paginator;
        }
        $resultSet = $this->tableGateway->select();
        return $resultSet;
    }
    ...
```

This will return a fully configured `Paginator` object. We've already told the `DbSelect` adapter to use our created `Select` object, to use the adapter that the `TableGateway` object uses, and also how to hydrate the result into a `Album` entity in the same fashion as the `TableGateway` does. This means that our executed and returned paginator results will return `Album` objects in exactly the same fashion as the non-paginated results.

19.3 Modifying the AlbumController

Next, we need to tell the album controller to return a `Pagination` object instead of a `ResultSet`. Both these objects can by iterated over to return hydrated `Album` objects, so we won't need to make many changes to the view script:

`module/Album/src/Album/Model/AlbumTable.php`

```
...
public function indexAction()
{
    // grab the paginator from the AlbumTable
    $paginator = $this->getAlbumTable()->fetchAll(true);
    // set the current page to what has been passed in query string, or to 1 if none set
    $paginator->setCurrentPageNumber((int)$this->params()->fromQuery('page', 1));
    // set the number of items per page to 10
    $paginator->setItemCountPerPage(10);

    return new ViewModel(array(
        'paginator' => $paginator
    ));
}
...
```

Here we are getting the configured `Paginator` object from the `AlbumTable`, and then telling it to use the page that is optionally passed in the querystring `page` parameter. We are also telling the paginator we want to display 10 objects per page.

19.4 Updating the View Script

Now, let's just tell the view script to iterate over the `pagination` view variable, rather than the `albums` variable:

`module/Album/view/album/album/index.phtml`

```
<table class="table">
    <tr>
        <th>Title</th>
        <th>Artist</th>
        <th> </th>
    </tr>
    <?php foreach ($this->paginator as $album) : // <-- change here! ?>
        <tr>
            <td><?php echo $this->escapeHtml($album->title);?></td>
            <td><?php echo $this->escapeHtml($album->artist);?></td>
            <td>
                <a href="<?php echo $this->url('album',
                    array('action' => 'edit', 'id' => $album->id));?>">Edit</a>
                <a href="<?php echo $this->url('album',
                    array('action' => 'delete', 'id' => $album->id));?>">Delete</a>
            </td>
        </tr>
    <?php endforeach; ?>
</table>
```

Checking the `/album` route on your website should now give you a list of just 10 albums, but with no method to navigate through the pages. Let's correct that now...

19.5 Creating the Pagination Control Partial

Much like we created a custom breadcrumbs partial to render our breadcrumb in the last tutorial, we need to create a custom pagination control partial to render our pagination control just the way we want it. Again, because we are using Twitter Bootstrap, this should be as simple as outputting correctly formatted html to get a pretty control. Let's create the partail in the `module/Application/view/partial/` folder, so that we can use the control in all our modules:

`module/Application/view/partial/paginator.phtml`

```
<?php if ($this->pageCount): ?>
    <div class="pagination pagination-centered">
        <ul>
            <!-- Previous page link -->
            <?php if (isset($this->previous)): ?>
                <li>
                    <a href="<?php echo $this->url($this->route); ?>?page=<?php echo $this->previous
                        <<
                    </a>
                </li>
            <?php else: ?>
                <li class="disabled">
```

```
                <a href="#">
                    <<
                </a>
            </li>
        <?php endif; ?>
        <!-- Numbered page links -->
        <?php foreach ($this->pagesInRange as $page): ?>
            <?php if ($page != $this->current): ?>
                <li>
                    <a href="<?php echo $this->url($this->route);?>?page=<?php echo $page; ?>">
                        <?php echo $page; ?>
                    </a>
                </li>
            <?php else: ?>
                <li class="active">
                    <a href="#"><?php echo $page; ?></a>
                </li>
            <?php endif; ?>
        <?php endforeach; ?>

        <!-- Next page link -->
        <?php if (isset($this->next)): ?>
            <li>
                <a href="<?php echo $this->url($this->route); ?>?page=<?php echo $this->next; ?>"
                    >>
                </a>
            </li>
        <?php else: ?>
            <li class="disabled">
                <a href="#">
                    >>
                </a>
            </li>
        <?php endif; ?>
    </ul>
</div>
<?php endif; ?>
```

All this partial does is to create a pagination control with links to the correct pages (if there is more than one page in the pagination object). It will render a previous page link (and mark it disabled if you are at the first page), then render a list of intermediate pages (that are passed to the partial based on the rendering style – we'll set in the view helper in the next step). Finally, it will create a next page link (and disable it if you're at the end). Notice how we pass the page number via the `page` querystring parameter which we have already told our controller to use to display the current page.

CHAPTER

TWENTY

USING THE PAGINATIONCONTROL VIEW HELPER

The only thing left for us to do so that we can page through the alumbs is to use the `PaginationControl` view helper to display our pagination control. This is nicely straightforward as we have already done all the ground work needed to display the control:

```
module/Album/view/album/album/index.phtml
...
<?php // add at the end of the file after the table
echo $this->paginationControl(
    // the paginator object
    $this->paginator,
    // the scrolling style; see http://zf2.readthedocs.org/en/release-2.1.4/modules/zend.paginator.us
    'sliding',
    // the partial to use to render the control
    array('partial/paginator.phtml', 'Album'),
    // the route to link to when a user clicks a control link
    array(
        'route' => 'album'
    )
);
?>
```

All we need to do here is to echo the *paginationControl* helper, and tell it to use our paginator object, sliding scrolling style, our paginator partial, and which route to use for clicks. Refreshing your application should give you a lovely bootstrap styled pagination control!

CHAPTER

TWENTYONE

SETTING UP A DATABASE ADAPTER

21.1 Introduction

In most cases, e.g. in your controllers, your database adapter can be fetched directly from the service manager. Some classes however, like `Zend\Validator\DbRecordExists` isn't aware of the service manager, but still needs an adapter to function.

There are many different ways to provide this functionality to your application. Below are a few examples.

21.2 Basic setup

Normally you will setup your database adapter using a factory in the service manager in your configuration. It might look something like this:

```
// config/autoload/database.local.php

return array(
   'db' => array(
      'driver'         => 'Pdo',
      'dsn'            => 'mysql:dbname=zf2tutorial;host=localhost',
   ),
   'service_manager' => array(
      'factories' => array(
         'Zend\Db\Adapter\Adapter' => 'Zend\Db\Adapter\AdapterServiceFactory',
      ),
   ),
);
```

The adapter can then be accessed in any ServiceLocatorAware classes.

```
public function getAdapter()
{
   if (!$this->adapter) {
      $sm = $this->getServiceLocator();
      $this->adapter = $sm->get('Zend\Db\Adapter\Adapter');
   }
   return $this->adapter;
}
```

More information on adapter options can be found in the docs for *Zend\Db\Adapter*.

21.3 Setting a static adapter

In order to utilize this adapter in non-ServiceLocatorAware classes, you can use `Zend\Db\TableGateway\Feature\GlobalAdapterFeature::setStaticAdapter()` to set a static adapter:

```
// config/autoload/database.local.php

return array(
   'db' => array(
      'driver'         => 'Pdo',
      'dsn'            => 'mysql:dbname=zf2tutorial;host=localhost',
   ),
   'service_manager' => array(
      'factories' => array(
         'Zend\Db\Adapter\Adapter' => function ($serviceManager) {
            $adapterFactory = new Zend\Db\Adapter\AdapterServiceFactory();
               $adapter = $adapterFactory->createService($serviceManager);

               \Zend\Db\TableGateway\Feature\GlobalAdapterFeature::setStaticAdapter($adapter);

               return $adapter;
         }
      ),
   ),
);
```

The adapter can then later be fetched using `Zend\Db\TableGateway\Feature\GlobalAdapterFeature::getStaticAd` for use in e.g. `Zend\Validator\DbRecordExists`:

```
$validator = new Zend\Validator\Db\RecordExists(
   array(
      'table'   => 'users',
      'field'   => 'emailaddress',
      'adapter' => \Zend\Db\TableGateway\Feature\GlobalAdapterFeature::getStaticAdapter();
   )
);
```

CHAPTER

TWENTYTWO

MIGRATION FROM ZEND FRAMEWORK 1

This guide is intended to provide tools and strategies for migrating from Zend Framework 1 to Zend Framework 2. There is no single solution that will work for every project, nor any tools to automate the process.

In this guide, we will cover the following:

- Tools for namespacing your code.
- Tools for consuming Zend Framework 2 within your Zend Framework 1 application.
- Strategies for running Zend Framework 2 and Zend Framework 1 in parallel.
- Strategies for making your code easier to migrate, focussing primarily on clean separation of your domain logic and the MVC layer.
- Strategies for migrating the MVC layer.
- Strategies for migrating your domain layer.

CHAPTER

TWENTYTHREE

NAMESPACING OLD CLASSES

ZF2's minimal version is PHP 5.3. The most notable feature of PHP 5.3 is the addition of namespaces, which ZF2 fully embraces. Moreover, new projects built on ZF2 also fully embrace PHP namespaces. The addition of namespaces to PHP has greatly improved the readability of long class names and has helped better organize code into modules and components. This transition has also given birth to some naming best practices that help developers organize their code bases consisting of classes, components, and modules in a consistent and clean fashion.

Converting an older code base that follows the original PEAR/ZF underscore separated class naming convention into a properly namespaced codebase is one of the easier strategies to employ in both modernizing your code base as well as getting ready to ZF2-ify your ZF1 application.

We've created a tool to help in this endeavor, it is located here:

https://github.com/zendframework/Namespacer

This tool will take a wholesale approach to converting older code like the following:

```
class My_Long_NestedComponent_ClassName
{
    // methods that use other classes
}
```

into:

```
namespace My\Long\NestedComponent;

use Other\Classes;
use Something\ElseConsumed;

class ClassName
{
    // methods with classes converted to short name from use statement.
}
```

Some IDEs have this capability to some degree. That said, a good approach might be to use the command line `Namespacer` to do a full sweep of your codebase, then use the IDE to make more specific naming changes that might makes more sense to your application.

23.1 Namespacing a ZF1 Application

The above `Namespacer` is a generalized tool. It does not understand the structure and naming conventions of a ZF1 application. As such, you'll need to address the problem of converting your classes according to their role, and which classes you find you can convert without affecting the way the framework interoperates with your code.

For example, in ZF1, the naming convention of application and module layer classes does not directly match up with same well-defined library class/file conventions of the PEAR/ZF namings. For a standard ZF1 application, in the `application/` directory, controller classes are not prefixed, yet model and form classes are prefixed with `Application_`. Moreover, they exist inside of lowercased directories, such as `models` or `forms`, and their file to class name segment matching picks up only after the first segment. As an example, you might have this directory structure with the class names on the right:

```
application/
    -- Bootstrap.php
    -- configs
    |   -- application.ini
    |   -- application.ini.dist
    -- controllers
    |   -- IndexController.php         [class IndexController]
    |   -- PurchaseOrderController.php [class PurchaseOrderController]
    -- forms
    |   -- PurchaseOrder
    |       -- Payment.php             [class Application_Form_PurchaseOrder_Payment]
    -- layouts
    |   -- scripts
    |       -- main.phtml
    |       -- subpage.phtml
    -- models
    |   -- DbTable
    |   |   -- Invoice.php             [Application_Model_DbTable_Invoice]
    |   -- Invoice.php                 [Application_Model_Invoice]
    |   -- InvoiceRepository.php       [Application_Model_InvoiceRepository]
    |   -- Payment
    |   |   -- Paypal
    |   |       --  DirectPayment.php  [Application_Model_Payment_Paypal_DirectPayment]
    |   -- PurchaseOrder.php           [Application_Model_PurchaseOrder]
    -- views
        -- scripts
            -- error
            |   -- error.phtml
            -- index
            |   -- index.phtml
            -- purchase-order
                -- index.phtml
                -- purchaser.phtml
```

It would not be a good strategy to attempt to do a wholesale namespacing of this kind of project for a number of reasons:

1. ZF1 has special, context-aware autoloaders that will assist loading a class of a particular context from a special location on disk. For example, ZF1 understands controllers will be located in the `controllers` directory and will not be prefixed unless they are inside of a named module's `controllers` directory.
2. Attempting to apply namespacing to controller classes would generally render a ZF1 application useless. ZF1, beyond loading files from disk, assumes controllers will have a very specific naming convention so that they can be invoked by the framework upon routing and dispatching.
3. Beyond dispatching, ZF1 uses the class name to identify and map the proper view script to automatically execute. By naming the controller something non-standard, views will no longer this this 1:1 mapping of controllers by name to controller action named view scripts.

A better solution would be to start by namespacing the parts of your ZF1 application that have fewer tie-ins with the ZF1 architecture. The place to start with this is models and forms.

Since models and forms do not touch controller and view classes (which make heavy use of ZF1 classes by way of

inheritance), model and form classes might not have the same level of coupling.

23.2 HOWTO Namespace Your Models

First, ensure your classes are under version control. The namespacer tool will make modification to classes in place. You can then use your version control system as a diffing utility afterwards .

To run the tool, download the phar. Optionally you can place the `namespacer.phar` into a directory in your `PATH`.

Namespacing is a 2 part process:

1. Create a map of all the old files, new files, old classes and new classes.
2. Make the transformations according to the map file.

Change into your `models/` directory and execute the map function:

```
namespacer.phar map --mapfile model-map.php --source models/
```

This will produce a file called `model-map.php` with entries like this:

```php
<?php return array (
    array (
      'root_directory' => '/realpath/to/project/application/models',
      'original_class' => 'Application_Model_Invoice',
      'original_file' => '/realpath/to/project/application/models/Invoice.php',
      'new_namespace' => 'Application\\Model',
      'new_class' => 'Invoice',
      'new_file' => '/realpath/to/project/application/models/Application/Model/Invoice.php',
    ),
    ...
);
```

This gives you an opportunity to manually edit the transformations if you so desire. While you can modify this file, you also might find it to be easier to go with the default transformations, and do the remaining changes with your IDE's refactoring utility.

Once you are happy with the map file, run the transformations:

```
namespacer.phar transform --mapfile model-map.php
```

At this point, you can use your version control system's `status` command to see how the directory has transformed. As an example, in a sample project of mine, `git` reports the following:

```
renamed:  models/DbTable/Invoice.php -> models/Application/Model/DbTable/Invoice.php
new file: models/Application/Model/DbTable/Transaction.php
renamed:  models/Invoice.php -> models/Application/Model/Invoice.php
renamed:  models/Payment/Paypal/DirectPayment.php -> models/Application/Model/Payment/Paypal/DirectPa
renamed:  models/PurchaseOrder.php -> models/Application/Model/PurchaseOrder.php
renamed:  models/PurchaseOrderRepository.php -> models/Application/Model/PurchaseOrderRepository.php
new file: models/Application/Model/PurchaseOrderService.php
renamed:  models/Purchaser.php -> models/Application/Model/Purchaser.php
renamed:  models/Ticket.php -> models/Application/Model/Ticket.php
renamed:  models/Transaction.php -> models/Application/Model/Transaction.php
renamed:  models/TransactionRepository.php -> models/Application/Model/TransactionRepository.php
deleted:  models/DbTable/Transaction.php
deleted:  models/PurchaseOrderService.php
```

You'll notice that the resulting files have treated the `models/` directory as the autoloader root directory. That means that from this root, class files follow the strict PEAR/ZF2 classfile naming convention. The contents of one of the files will look like this:

```
namespace Application\Model;

use Application\Model\PurchaseOrder;
use Application\Model\Transaction;
use Zend_Filter_Alnum;

class Invoice
{

    protected $tickets;
    protected $transaction;

    ...
}
```

Things to notice here:

- A namespace has been created for this class.
- The namespacer has created PHP `use` statements for classes known in the map file.
- Unknown classes are also included (for example, `Zend` classes) in `use` statements.

By keeping the old ZF1 classes, your models should continue to work if they consume ZF1 classes. This will allow you to, at your own pace, transition your codebase to ZF2.

This same procedure can largely be adapted to forms and independent library code as well.

CHAPTER

TWENTYFOUR

RUNNING ZEND FRAMEWORK 2 AND ZEND FRAMEWORK 1 IN PARALLEL

From a technical point of view it is absolutely possible to run ZF2 in parallel with ZF1 because there is no conflict between the classnames due to the fact that ZF2 uses namespaces and ZF1 does not. Running ZF1 and ZF2 in parallel can be used as a migration strategy in projects where it is not possible, or not convenient, to migrate an entire application from ZF1 to ZF2. For instance, you could implement any new features of the application using ZF2, while maintaining original ZF1 features.

Let's examine some scenarios on how to execute ZF1 and ZF2 together.

24.1 Use ZF2 in a ZF1 project

Suppose we have an existing ZF1 application and we want to start using ZF2; how could we do that?

Because ZF2 uses namespaced classes, you can run it in parallel with ZF1 without naming conflicts. In order to do this, you will need to add some code to autoload ZF2 from within your ZF1 project. Add these lines of code in your `public/index.php`, before the instantiation of `$application`:

```
define('ZF2_PATH', '/path/to/zf2/library');
require_once ZF2_PATH . '/Zend/Loader/StandardAutoloader.php';
$loader = new Zend\Loader\StandardAutoloader(array(
    'autoregister_zf' => true,
));
$loader->register();
```

We used the `StandardAutoloader` class from ZF2. Using this autoloader, classes with the initial namespace `Zend` will be loaded using the `ZF2_PATH`, and any ZF1 classes will continue to be loaded via the mechanisms present in ZF1.

Of course, this is not a real integration of ZF2 inside ZF1; it only provides the ability to consume ZF2 classes within your ZF1 application. For instance, you cannot use the MVC architecture of ZF2 because you are using the MVC of ZF1.

Evan Coury, a member of the ZF community review team, has produced a nice module for ZF1 (zf-2-for-1) that allows you to use ZF2 features inside an existing ZF1 application. This module offers some basic integrations like the usage of ZF2 view helpers in the ZF1 view layer (i.e. `$this->zf2->get('formRow')`).

24.2 Use ZF1 in a ZF2 project

You can add ZF1 to your ZF2 application via Composer by adding the "zendframework/zendframework1" package as a requirement.

For instance, if you have a ZF2 application and you want to install ZF 1.12, you need to add the following line in the require section of your `composer.json` file:

```
"require": {
     "php": ">=5.3.3",
     "zendframework/zendframework1": "1.12",
     ...
 }
```

After executing `composer.phar update`, you can start to use ZF1 classes in your ZF2 project. Since all ZF1 classes exist in the global namespace, you will need to refer to them by their full name; as examples, `Zend_Date`, `Zend_Feed_Reader`, etc.

For other strategies on how to use ZF1 in a ZF2 project, you can check out this blog post by Abdul Malik Ikhsan, Zend Framework 2 : Using Zend Framework 1 libraries.

24.3 Run ZF1 and ZF2 together

As we mentioned early, one way to migrate a ZF1 application to ZF2 can be to execute in parallel the different versions of the framework, using ZF2 for the new features, and migrating the ZF1 code step by step. In order to execute in parallel, we need to map different URLs to the different front controllers for ZF1 and ZF2. This goal can be accomplished using the rewriting rules of your web server. From a performance point of view, this is the best solution because it does not involve pre-processing overhead. For each URL we can define a different version of the framework to be used.

For instance, imagine we have a ZF1 application and we want to use ZF2 only for URLs starting with `/album`. We can use the following `.htaccess` file (this information is related to apache; if you are using another web server, read the instructions in the note below):

```
SetEnv APPLICATION_ENV development
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^album(/.*)?$ index_zf2.php [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

`index_zf2.php` is a PHP script that includes as the typical `public/index.php` file of ZF2. Here is the source code for `index_zf2.php`:

```
require_once '../path-to-ZF2-app/public/index.php';
```

We suggest putting the ZF2 application in a separate folder under the same root directory of the ZF1 application. In this way you can continue to maintain the existing ZF1 code and use ZF2 only for the new features. Moreover, if you want to migrate the old code you can do that by URL and switch to the new ZF2 code only when you are ready. This approach can be useful to provide migration guideline without losing development time in a full stack migration.

Note: All web servers support a rewriting mechanism. For instance, if you are using **'Microsoft IIS 7'_**, you can check how to configure the rewriting rules from Rob Allen's post Zend Framework URL Rewriting in IIS7; if you are using nginx, you can check out this StackOverflow question: Zend Framework on nginx.

CHAPTER

TWENTYFIVE

INTRODUCTION TO ZEND\AUTHENTICATION

The `Zend\Authentication` component provides an *API* for authentication and includes concrete authentication adapters for common use case scenarios.

`Zend\Authentication` is concerned only with **authentication** and not with **authorization**. Authentication is loosely defined as determining whether an entity actually is what it purports to be (i.e., identification), based on some set of credentials. Authorization, the process of deciding whether to allow an entity access to, or to perform operations upon, other entities is outside the scope of `Zend\Authentication`. For more information about authorization and access control with Zend Framework, please see the *Zend\Permissions\Acl* or *Zend\Permissions\Rbac* component.

Note: There is no `Zend\Authentication\Authentication` class, instead the class `Zend\Authentication\AuthenticationService` is provided. This class uses underlying authentication adapters and persistent storage backends.

25.1 Adapters

`Zend\Authentication` adapters are used to authenticate against a particular type of authentication service, such as *LDAP*, *RDBMS*, or file-based storage. Different adapters are likely to have vastly different options and behaviors, but some basic things are common among authentication adapters. For example, accepting authentication credentials (including a purported identity), performing queries against the authentication service, and returning results are common to `Zend\Authentication` adapters.

Each `Zend\Authentication` adapter class implements `Zend\Authentication\Adapter\AdapterInterface`. This interface defines one method, `authenticate()`, that an adapter class must implement for performing an authentication query. Each adapter class must be prepared prior to calling `authenticate()`. Such adapter preparation includes setting up credentials (e.g., username and password) and defining values for adapter-specific configuration options, such as database connection settings for a database table adapter.

The following is an example authentication adapter that requires a username and password to be set for authentication. Other details, such as how the authentication service is queried, have been omitted for brevity:

```
use Zend\Authentication\Adapter\AdapterInterface;

class My\Auth\Adapter implements AdapterInterface
{
    /**
     * Sets username and password for authentication
     *
```

```
     * @return void
     */
    public function __construct($username, $password)
    {
        // ...
    }

    /**
     * Performs an authentication attempt
     *
     * @return \Zend\Authentication\Result
     * @throws \Zend\Authentication\Adapter\Exception\ExceptionInterface
     *               If authentication cannot be performed
     */
    public function authenticate()
    {
        // ...
    }
}
```

As indicated in its docblock, `authenticate()` must return an instance of `Zend\Authentication\Result` (or of a class derived from `Zend\Authentication\Result`). If for some reason performing an authentication query is impossible, `authenticate()` should throw an exception that derives from `Zend\Authentication\Adapter\Exception\ExceptionInterface`.

25.2 Results

`Zend\Authentication` adapters return an instance of `Zend\Authentication\Result` with `authenticate()` in order to represent the results of an authentication attempt. Adapters populate the `Zend\Authentication\Result` object upon construction, so that the following four methods provide a basic set of user-facing operations that are common to the results of `Zend\Authentication` adapters:

- `isValid()` - returns `TRUE` if and only if the result represents a successful authentication attempt
- `getCode()` - returns a `Zend\Authentication\Result` constant identifier for determining the type of authentication failure or whether success has occurred. This may be used in situations where the developer wishes to distinguish among several authentication result types. This allows developers to maintain detailed authentication result statistics, for example. Another use of this feature is to provide specific, customized messages to users for usability reasons, though developers are encouraged to consider the risks of providing such detailed reasons to users, instead of a general authentication failure message. For more information, see the notes below.
- `getIdentity()` - returns the identity of the authentication attempt
- `getMessages()` - returns an array of messages regarding a failed authentication attempt

A developer may wish to branch based on the type of authentication result in order to perform more specific operations. Some operations developers might find useful are locking accounts after too many unsuccessful password attempts, flagging an IP address after too many nonexistent identities are attempted, and providing specific, customized authentication result messages to the user. The following result codes are available:

```
use Zend\Authentication\Result;

Result::SUCCESS
Result::FAILURE
Result::FAILURE_IDENTITY_NOT_FOUND
Result::FAILURE_IDENTITY_AMBIGUOUS
```

```
Result::FAILURE_CREDENTIAL_INVALID
Result::FAILURE_UNCATEGORIZED
```

The following example illustrates how a developer may branch on the result code:

```
// inside of AuthController / loginAction
$result = $this->auth->authenticate($adapter);

switch ($result->getCode()) {

    case Result::FAILURE_IDENTITY_NOT_FOUND:
        /** do stuff for nonexistent identity **/
        break;

    case Result::FAILURE_CREDENTIAL_INVALID:
        /** do stuff for invalid credential **/
        break;

    case Result::SUCCESS:
        /** do stuff for successful authentication **/
        break;

    default:
        /** do stuff for other failure **/
        break;
}
```

25.3 Identity Persistence

Authenticating a request that includes authentication credentials is useful per se, but it is also important to support maintaining the authenticated identity without having to present the authentication credentials with each request.

HTTP is a stateless protocol, however, and techniques such as cookies and sessions have been developed in order to facilitate maintaining state across multiple requests in server-side web applications.

25.3.1 Default Persistence in the PHP Session

By default, `Zend\Authentication` provides persistent storage of the identity from a successful authentication attempt using the *PHP* session. Upon a successful authentication attempt, `Zend\Authentication\AuthenticationService::authenticate()` stores the identity from the authentication result into persistent storage. Unless specified otherwise, `Zend\Authentication\AuthenticationService` uses a storage class named `Zend\Authentication\Storage\Session`, which, in turn, uses *Zend\Session*. A custom class may instead be used by providing an object that implements `Zend\Authentication\Storage\StorageInterface` to `Zend\Authentication\AuthenticationService::setStorage()`.

Note: If automatic persistent storage of the identity is not appropriate for a particular use case, then developers may forget using the `Zend\Authentication\AuthenticationService` class altogether, instead using an adapter class directly.

Modifying the Session Namespace

`Zend\Authentication\Storage\Session` uses a session namespace of '`Zend_Auth`'. This namespace may be overridden by passing a different value to the constructor of `Zend\Authentication\Storage\Session`, and this value is internally passed along to the constructor of *Zend\Session\Container*. This should occur before authentication is attempted, since `Zend\Authentication\AuthenticationService::authenticate()` performs the automatic storage of the identity.

```
use Zend\Authentication\AuthenticationService;
use Zend\Authentication\Storage\Session as SessionStorage;

$auth = new AuthenticationService();

// Use 'someNamespace' instead of 'Zend_Auth'
$auth->setStorage(new SessionStorage('someNamespace'));

/**
 * @todo Set up the auth adapter, $authAdapter
 */

// Authenticate, saving the result, and persisting the identity on
// success
$result = $auth->authenticate($authAdapter);
```

25.3.2 Chain Storage

A website may have multiple storage in place. The `Chain` Storage can be used to glue these together.

The `Chain` can for example be configured to first use a `Session` Storage and then use a `OAuth` as a secondary Storage. One could configure this in the following way:

```
 $storage = new Chain;
 $storage->add(new Session);
 $storage->add(new OAuth); // Note: imaginary storage, not part of ZF2
```

Now if the `Chain` Storage is accessed its underlying Storage will get accessed in the order in which they were added to the chain. Thus first the `Session` Storage is used. Now either:

- The `Session` Storage is non-empty and the `Chain` will use its contents.
- The `Sesssion` Storage is empty. Next the `OAuth` Storage is accessed.
  - If this one is also empty the Chain will act as empty.
  - If this one is non-empty the `Chain` will use its contents. However it will also populate all Storage with higher priority. Thus the `Session` Storage will be populated with the contents of the `Oauth` Storage.

The priority of Storage in the Chain can be made explicit via the `Chain::add` method.

```
 $chain->add(new A, 2);
 $chain->add(new B, 10); // First use B
```

25.3.3 Implementing Customized Storage

Sometimes developers may need to use a different identity storage mechanism than that provided by `Zend\Authentication\Storage\Session`. For such cases developers may simply imple-

ment `Zend\Authentication\Storage\StorageInterface` and supply an instance of the class to `Zend\Authentication\AuthenticationService::setStorage()`.

Using a Custom Storage Class

In order to use an identity persistence storage class other than `Zend\Authentication\Storage\Session`, a developer implements `Zend\Authentication\Storage\StorageInterface`:

```
use Zend\Authentication\Storage\StorageInterface;

class My\Storage implements StorageInterface
{
    /**
     * Returns true if and only if storage is empty
     *
     * @throws \Zend\Authentication\Exception\ExceptionInterface
     *               If it is impossible to
     *               determine whether storage is empty
     * @return boolean
     */
    public function isEmpty()
    {
        /**
         * @todo implementation
         */
    }

    /**
     * Returns the contents of storage
     *
     * Behavior is undefined when storage is empty.
     *
     * @throws \Zend\Authentication\Exception\ExceptionInterface
     *               If reading contents from storage is impossible
     * @return mixed
     */

    public function read()
    {
        /**
         * @todo implementation
         */
    }

    /**
     * Writes $contents to storage
     *
     * @param  mixed $contents
     * @throws \Zend\Authentication\Exception\ExceptionInterface
     *               If writing $contents to storage is impossible
     * @return void
     */

    public function write($contents)
    {
        /**
         * @todo implementation
```

```
         */
    }

    /**
     * Clears contents from storage
     *
     * @throws \Zend\Authentication\Exception\ExceptionInterface
     *                If clearing contents from storage is impossible
     * @return void
     */

    public function clear()
    {
        /**
         * @todo implementation
         */
    }
}
```

In order to use this custom storage class, `Zend\Authentication\AuthenticationService::setStorage()` is invoked before an authentication query is attempted:

```
use Zend\Authentication\AuthenticationService;

// Instruct AuthenticationService to use the custom storage class
$auth = new AuthenticationService();

$auth->setStorage(new My\Storage());

/**
 * @todo Set up the auth adapter, $authAdapter
 */

// Authenticate, saving the result, and persisting the identity on
// success
$result = $auth->authenticate($authAdapter);
```

25.4 Usage

There are two provided ways to use `Zend\Authentication` adapters:

- indirectly, through `Zend\Authentication\AuthenticationService::authenticate()`
- directly, through the adapter's `authenticate()` method

The following example illustrates how to use a `Zend\Authentication` adapter indirectly, through the use of the `Zend\Authentication\AuthenticationService` class:

```
use Zend\Authentication\AuthenticationService;

// instantiate the authentication service
$auth = new AuthenticationService();

// Set up the authentication adapter
$authAdapter = new My\Auth\Adapter($username, $password);

// Attempt authentication, saving the result
```

```
$result = $auth->authenticate($authAdapter);

if (!$result->isValid()) {
    // Authentication failed; print the reasons why
    foreach ($result->getMessages() as $message) {
        echo "$message\n";
    }
} else {
    // Authentication succeeded; the identity ($username) is stored
    // in the session
    // $result->getIdentity() === $auth->getIdentity()
    // $result->getIdentity() === $username
}
```

Once authentication has been attempted in a request, as in the above example, it is a simple matter to check whether a successfully authenticated identity exists:

```
use Zend\Authentication\AuthenticationService;

$auth = new AuthenticationService();

/**
 * @todo Set up the auth adapter, $authAdapter
 */

if ($auth->hasIdentity()) {
    // Identity exists; get it
    $identity = $auth->getIdentity();
}
```

To remove an identity from persistent storage, simply use the `clearIdentity()` method. This typically would be used for implementing an application "logout" operation:

```
$auth->clearIdentity();
```

When the automatic use of persistent storage is inappropriate for a particular use case, a developer may simply bypass the use of the `Zend\Authentication\AuthenticationService` class, using an adapter class directly. Direct use of an adapter class involves configuring and preparing an adapter object and then calling its `authenticate()` method. Adapter-specific details are discussed in the documentation for each adapter. The following example directly utilizes `My\Auth\Adapter`:

```
// Set up the authentication adapter
$authAdapter = new My\Auth\Adapter($username, $password);

// Attempt authentication, saving the result
$result = $authAdapter->authenticate();

if (!$result->isValid()) {
    // Authentication failed; print the reasons why
    foreach ($result->getMessages() as $message) {
        echo "$message\n";
    }
} else {
    // Authentication succeeded
    // $result->getIdentity() === $username
}
```

CHAPTER

TWENTYSIX

DATABASE TABLE AUTHENTICATION

26.1 Introduction

`Zend\Authentication\Adapter\DbTable` provides the ability to authenticate against credentials stored in a database table. Because `Zend\Authentication\Adapter\DbTable` requires an instance of `Zend\Db\Adapter\Adapter` to be passed to its constructor, each instance is bound to a particular database connection. Other configuration options may be set through the constructor and through instance methods, one for each option.

The available configuration options include:

- **tableName**: This is the name of the database table that contains the authentication credentials, and against which the database authentication query is performed.
- **identityColumn**: This is the name of the database table column used to represent the identity. The identity column must contain unique values, such as a username or e-mail address.
- **credentialColumn**: This is the name of the database table column used to represent the credential. Under a simple identity and password authentication scheme, the credential value corresponds to the password. See also the `credentialTreatment` option.
- **credentialTreatment**: In many cases, passwords and other sensitive data are encrypted, hashed, encoded, obscured, salted or otherwise treated through some function or algorithm. By specifying a parameterized treatment string with this method, such as '`MD5(?)`' or '`PASSWORD(?)`', a developer may apply such arbitrary *SQL* upon input credential data. Since these functions are specific to the underlying *RDBMS*, check the database manual for the availability of such functions for your database system.

26.2 Basic Usage

As explained in the introduction, the `Zend\Authentication\Adapter\DbTable` constructor requires an instance of `Zend\Db\Adapter\Adapter` that serves as the database connection to which the authentication adapter instance is bound. First, the database connection should be created.

The following code creates an adapter for an in-memory database, creates a simple table schema, and inserts a row against which we can perform an authentication query later. This example requires the *PDO* SQLite extension to be available:

```
use Zend\Db\Adapter\Adapter as DbAdapter;

// Create a SQLite database connection
$dbAdapter = new DbAdapter(array(
                'driver' => 'Pdo_Sqlite',
```

```
                'database' => 'path/to/sqlite.db'
            ));

// Build a simple table creation query
$sqlCreate = 'CREATE TABLE [users] ('
           . '[id] INTEGER  NOT NULL PRIMARY KEY, '
           . '[username] VARCHAR(50) UNIQUE NOT NULL, '
           . '[password] VARCHAR(32) NULL, '
           . '[real_name] VARCHAR(150) NULL)';

// Create the authentication credentials table
$dbAdapter->query($sqlCreate);

// Build a query to insert a row for which authentication may succeed
$sqlInsert = "INSERT INTO users (username, password, real_name) "
           . "VALUES ('my_username', 'my_password', 'My Real Name')";

// Insert the data
$dbAdapter->query($sqlInsert);
```

With the database connection and table data available, an instance of `Zend\Authentication\Adapter\DbTable` may be created. Configuration option values may be passed to the constructor or deferred as parameters to setter methods after instantiation:

```
use Zend\Authentication\Adapter\DbTable as AuthAdapter;

// Configure the instance with constructor parameters...
$authAdapter = new AuthAdapter($dbAdapter,
                               'users',
                               'username',
                               'password'
                               );

// ...or configure the instance with setter methods
$authAdapter = new AuthAdapter($dbAdapter);

$authAdapter
    ->setTableName('users')
    ->setIdentityColumn('username')
    ->setCredentialColumn('password')
;
```

At this point, the authentication adapter instance is ready to accept authentication queries. In order to formulate an authentication query, the input credential values are passed to the adapter prior to calling the `authenticate()` method:

```
// Set the input credential values (e.g., from a login form)
$authAdapter
    ->setIdentity('my_username')
    ->setCredential('my_password')
;

// Perform the authentication query, saving the result
```

In addition to the availability of the `getIdentity()` method upon the authentication result object, `Zend\Authentication\Adapter\DbTable` also supports retrieving the table row upon authentication success:

```
// Print the identity
echo $result->getIdentity() . "\n\n";

// Print the result row
print_r($authAdapter->getResultRowObject());

/* Output:
my_username

Array
(
    [id] => 1
    [username] => my_username
    [password] => my_password
    [real_name] => My Real Name
)
*/
```

Since the table row contains the credential value, it is important to secure the values against unintended access.

When retrieving the result object, we can either specify what columns to return, or what columns to omit:

```
$columnsToReturn = array(
    'id', 'username', 'real_name'
);
print_r($authAdapter->getResultRowObject($columnsToReturn));

/* Output:

Array
(
   [id] => 1
   [username] => my_username
   [real_name] => My Real Name
)
*/

$columnsToOmit = array('password');
print_r($authAdapter->getResultRowObject(null, $columnsToOmit);

/* Output:

Array
(
   [id] => 1
   [username] => my_username
   [real_name] => My Real Name
)
*/
```

26.3 Advanced Usage: Persisting a DbTable Result Object

By default, `Zend\Authentication\Adapter\DbTable` returns the identity supplied back to the auth object upon successful authentication. Another use case scenario, where developers want to store to the persistent storage mechanism of `Zend\Authentication` an identity object containing other useful information, is solved by using the `getResultRowObject()` method to return a **stdClass** object. The following code snippet illustrates its use:

```
// authenticate with Zend\Authentication\Adapter\DbTable
$result = $this->_auth->authenticate($adapter);

if ($result->isValid()) {
    // store the identity as an object where only the username and
    // real_name have been returned
    $storage = $this->_auth->getStorage();
    $storage->write($adapter->getResultRowObject(array(
        'username',
        'real_name',
    )));

    // store the identity as an object where the password column has
    // been omitted
    $storage->write($adapter->getResultRowObject(
        null,
        'password'
    ));

    /* ... */

} else {

    /* ... */

}
```

26.3.1 Advanced Usage By Example

While the primary purpose of the `Zend\Authentication` component (and consequently `Zend\Authentication\Adapter\DbTable`) is primarily **authentication** and not **authorization**, there are a few instances and problems that toe the line between which domain they fit within. Depending on how you've decided to explain your problem, it sometimes makes sense to solve what could look like an authorization problem within the authentication adapter.

With that disclaimer out of the way, `Zend\Authentication\Adapter\DbTable` has some built in mechanisms that can be leveraged for additional checks at authentication time to solve some common user problems.

```
use Zend\Authentication\Adapter\DbTable as AuthAdapter;

// The status field value of an account is not equal to "compromised"
$adapter = new AuthAdapter($db,
                           'users',
                           'username',
                           'password',
                           'MD5(?) AND status != "compromised"'
                           );

// The active field value of an account is equal to "TRUE"
$adapter = new AuthAdapter($db,
                           'users',
                           'username',
                           'password',
                           'MD5(?) AND active = "TRUE"'
                           );
```

Another scenario can be the implementation of a salting mechanism. Salting is a term referring to a technique which

can highly improve your application's security. It's based on the idea that concatenating a random string to every password makes it impossible to accomplish a successful brute force attack on the database using pre-computed hash values from a dictionary.

Therefore, we need to modify our table to store our salt string:

```
$sqlAlter = "ALTER TABLE [users] "
          . "ADD COLUMN [password_salt] "
          . "AFTER [password]";
```

Here's a simple way to generate a salt string for every user at registration:

```
$dynamicSalt = '';
for ($i = 0; $i < 50; $i++) {
    $dynamicSalt .= chr(rand(33, 126));
}
```

And now let's build the adapter:

```
$adapter = new AuthAdapter($db,
                           'users',
                           'username',
                           'password',
                           "MD5(CONCAT('staticSalt', ?, password_salt))"
                          );
```

Note: You can improve security even more by using a static salt value hard coded into your application. In the case that your database is compromised (e. g. by an *SQL* injection attack) but your web server is intact your data is still unusable for the attacker.

Another alternative is to use the `getDbSelect()` method of the `Zend\Authentication\Adapter\DbTable` after the adapter has been constructed. This method will return the `Zend\Db\Sql\Select` object instance it will use to complete the `authenticate()` routine. It is important to note that this method will always return the same object regardless if `authenticate()` has been called or not. This object **will not** have any of the identity or credential information in it as those values are placed into the select object at `authenticate()` time.

An example of a situation where one might want to use the `getDbSelect()` method would check the status of a user, in other words to see if that user's account is enabled.

```
// Continuing with the example from above
$adapter = new AuthAdapter($db,
                           'users',
                           'username',
                           'password',
                           'MD5(?)'
                           );

// get select object (by reference)
$select = $adapter->getDbSelect();
$select->where('active = "TRUE"');

// authenticate, this ensures that users.active = TRUE
$adapter->authenticate();
```

CHAPTER

TWENTYSEVEN

DIGEST AUTHENTICATION

27.1 Introduction

Digest authentication is a method of *HTTP* authentication that improves upon Basic authentication by providing a way to authenticate without having to transmit the password in clear text across the network.

This adapter allows authentication against text files containing lines having the basic elements of Digest authentication:

- username, such as "**joe.user**"
- realm, such as "**Administrative Area**"
- *MD5* hash of the username, realm, and password, separated by colons

The above elements are separated by colons, as in the following example (in which the password is "**somePassword**"):

```
someUser:Some Realm:fde17b91c3a510ecbaf7dbd37f59d4f8
```

27.2 Specifics

The digest authentication adapter, `Zend\Authentication\Adapter\Digest`, requires several input parameters:

- filename - Filename against which authentication queries are performed
- realm - Digest authentication realm
- username - Digest authentication user
- password - Password for the user of the realm

These parameters must be set prior to calling `authenticate()`.

27.3 Identity

The digest authentication adapter returns a `Zend\Authentication\Result` object, which has been populated with the identity as an array having keys of **realm** and **username**. The respective array values associated with these keys correspond to the values set before `authenticate()` is called.

```
use Zend\Authentication\Adapter\Digest as AuthAdapter;

$adapter = new AuthAdapter($filename,
```

```
                            $realm,
                            $username,
                            $password);

$result = $adapter->authenticate();

$identity = $result->getIdentity();

print_r($identity);

/*
Array
(
    [realm] => Some Realm
    [username] => someUser
)
*/
```

CHAPTER

TWENTYEIGHT

HTTP AUTHENTICATION ADAPTER

28.1 Introduction

`Zend\Authentication\Adapter\Http` provides a mostly-compliant implementation of RFC-2617, Basic and Digest *HTTP* Authentication. Digest authentication is a method of *HTTP* authentication that improves upon Basic authentication by providing a way to authenticate without having to transmit the password in clear text across the network.

Major Features:

- Supports both Basic and Digest authentication.
- Issues challenges in all supported schemes, so client can respond with any scheme it supports.
- Supports proxy authentication.
- Includes support for authenticating against text files and provides an interface for authenticating against other sources, such as databases.

There are a few notable features of *RFC-2617* that are not implemented yet:

- Nonce tracking, which would allow for "stale" support, and increased replay attack protection.
- Authentication with integrity checking, or "auth-int".
- Authentication-Info *HTTP* header.

28.2 Design Overview

This adapter consists of two sub-components, the *HTTP* authentication class itself, and the so-called "Resolvers." The *HTTP* authentication class encapsulates the logic for carrying out both Basic and Digest authentication. It uses a Resolver to look up a client's identity in some data store (text file by default), and retrieve the credentials from the data store. The "resolved" credentials are then compared to the values submitted by the client to determine whether authentication is successful.

28.3 Configuration Options

The `Zend\Authentication\Adapter\Http` class requires a configuration array passed to its constructor. There are several configuration options available, and some are required:

Table 28.1: Configuration Options

Option Name	Required	Description
accept_schemes	Yes	Determines which authentication schemes the adapter will accept from the client. Must be a space-separated list containing 'basic' and/or 'digest'.
realm	Yes	Sets the authentication realm; usernames should be unique within a given realm.
digest_domains	Yes, when accept_schemes contains digest	Space-separated list of URIs for which the same authentication information is valid. The URIs need not all point to the same server.
nonce_timeout	Yes, when accept_schemes contains digest	Sets the number of seconds for which the nonce is valid. See notes below.
use_opaque	No	Specifies whether to send the opaque value in the header. True by default.
algorithm	No	Specified the algorithm. Defaults to MD5, the only supported option (for now).
proxy_auth	No	Disabled by default. Enable to perform Proxy authentication, instead of normal origin server authentication.

Note: The current implementation of the `nonce_timeout` has some interesting side effects. This setting is supposed to determine the valid lifetime of a given nonce, or effectively how long a client's authentication information is accepted. Currently, if it's set to 3600 (for example), it will cause the adapter to prompt the client for new credentials every hour, on the hour. This will be resolved in a future release, once nonce tracking and stale support are implemented.

28.4 Resolvers

The resolver's job is to take a username and realm, and return some kind of credential value. Basic authentication expects to receive the Base64 encoded version of the user's password. Digest authentication expects to receive a hash of the user's username, the realm, and their password (each separated by colons). Currently, the only supported hash algorithm is *MD5*.

`Zend\Authentication\Adapter\Http` relies on objects implementing `Zend\Authentication\Adapter\Http\ResolverInterface`. A text file resolver class is included with this adapter, but any other kind of resolver can be created simply by implementing the resolver interface.

28.4.1 File Resolver

The file resolver is a very simple class. It has a single property specifying a filename, which can also be passed to the constructor. Its `resolve()` method walks through the text file, searching for a line with a matching username and realm. The text file format similar to Apache htpasswd files:

```
<username>:<realm>:<credentials>\n
```

Each line consists of three fields - username, realm, and credentials - each separated by a colon. The credentials field is opaque to the file resolver; it simply returns that value as-is to the caller. Therefore, this same file format serves both Basic and Digest authentication. In Basic authentication, the credentials field should be written in clear text. In Digest authentication, it should be the *MD5* hash described above.

There are two equally easy ways to create a File resolver:

```
use Zend\Authentication\Adapter\Http\FileResolver;
$path     = 'files/passwd.txt';
$resolver = new FileResolver($path);
```

or

```
$path     = 'files/passwd.txt';
$resolver = new FileResolver();
$resolver->setFile($path);
```

If the given path is empty or not readable, an exception is thrown.

28.5 Basic Usage

First, set up an array with the required configuration values:

```
$config = array(
    'accept_schemes' => 'basic digest',
    'realm'          => 'My Web Site',
    'digest_domains' => '/members_only /my_account',
    'nonce_timeout'  => 3600,
);
```

This array will cause the adapter to accept either Basic or Digest authentication, and will require authenticated access to all the areas of the site under `/members_only` and `/my_account`. The realm value is usually displayed by the browser in the password dialog box. The `nonce_timeout`, of course, behaves as described above.

Next, create the `Zend\Authentication\Adapter\Http` object:

```
$adapter = new Zend\Authentication\Adapter\Http($config);
```

Since we're supporting both Basic and Digest authentication, we need two different resolver objects. Note that this could just as easily be two different classes:

```
use Zend\Authentication\Adapter\Http\FileResolver;

$basicResolver = new FileResolver();
$basicResolver->setFile('files/basicPasswd.txt');

$digestResolver = new FileResolver();
$digestResolver->setFile('files/digestPasswd.txt');

$adapter->setBasicResolver($basicResolver);
$adapter->setDigestResolver($digestResolver);
```

Finally, we perform the authentication. The adapter needs a reference to both the Request and Response objects in order to do its job:

```
assert($request instanceof Zend\Http\Request);
assert($response instanceof Zend\Http\Response);

$adapter->setRequest($request);
$adapter->setResponse($response);

$result = $adapter->authenticate();
if (!$result->isValid()) {
    // Bad username/password, or canceled password prompt
}
```

CHAPTER

TWENTYNINE

LDAP AUTHENTICATION

29.1 Introduction

`Zend\Authentication\Adapter\Ldap` supports web application authentication with *LDAP* services. Its features include username and domain name canonicalization, multi-domain authentication, and failover capabilities. It has been tested to work with Microsoft Active Directory and OpenLDAP, but it should also work with other *LDAP* service providers.

This documentation includes a guide on using `Zend\Authentication\Adapter\Ldap`, an exploration of its *API*, an outline of the various available options, diagnostic information for troubleshooting authentication problems, and example options for both Active Directory and OpenLDAP servers.

29.2 Usage

To incorporate `Zend\Authentication\Adapter\Ldap` authentication into your application quickly, even if you're not using `Zend\Mvc`, the meat of your code should look something like the following:

```
use Zend\Authentication\AuthenticationService;
use Zend\Authentication\Adapter\Ldap as AuthAdapter;
use Zend\Config\Reader\Ini as ConfigReader;
use Zend\Config\Config;
use Zend\Log\Logger;
use Zend\Log\Writer\Stream as LogWriter;
use Zend\Log\Filter\Priority as LogFilter;

$username = $this->getRequest()->getPost('username');
$password = $this->getRequest()->getPost('password');


$auth = new AuthenticationService();

$configReader = new ConfigReader();
$configData = $configReader->fromFile('./ldap-config.ini');
$config = new Config($configData, true);

$log_path = $config->production->ldap->log_path;
$options = $config->production->ldap->toArray();
unset($options['log_path']);

$adapter = new AuthAdapter($options,
                           $username,
```

```
                            $password);

$result = $auth->authenticate($adapter);

if ($log_path) {
    $messages = $result->getMessages();

    $logger = new Logger;
    $writer = new LogWriter($log_path);

    $logger->addWriter($writer);

    $filter = new LogFilter(Logger::DEBUG);
    $writer->addFilter($filter);

    foreach ($messages as $i => $message) {
        if ($i-- > 1) { // $messages[2] and up are log messages
            $message = str_replace("\n", "\n  ", $message);
            $logger->debug("Ldap: $i: $message");
        }
    }
}
```

Of course, the logging code is optional, but it is highly recommended that you use a logger. `Zend\Authentication\Adapter\Ldap` will record just about every bit of information anyone could want in `$messages` (more below), which is a nice feature in itself for something that has a history of being notoriously difficult to debug.

The `Zend\Config\Reader\Ini` code is used above to load the adapter options. It is also optional. A regular array would work equally well. The following is an example `ldap-config.ini` file that has options for two separate servers. With multiple sets of server options the adapter will try each, in order, until the credentials are successfully authenticated. The names of the servers (e.g., 'server1' and 'server2') are largely arbitrary. For details regarding the options array, see the **Server Options** section below. Note that `Zend\Config\Reader\Ini` requires that any values with "equals" characters (=) will need to be quoted (like the DNs shown below).

```
[production]

ldap.log_path = /tmp/ldap.log

; Typical options for OpenLDAP
ldap.server1.host = s0.foo.net
ldap.server1.accountDomainName = foo.net
ldap.server1.accountDomainNameShort = FOO
ldap.server1.accountCanonicalForm = 3
ldap.server1.username = "CN=user1,DC=foo,DC=net"
ldap.server1.password = pass1
ldap.server1.baseDn = "OU=Sales,DC=foo,DC=net"
ldap.server1.bindRequiresDn = true

; Typical options for Active Directory
ldap.server2.host = dc1.w.net
ldap.server2.useStartTls = true
ldap.server2.accountDomainName = w.net
ldap.server2.accountDomainNameShort = W
ldap.server2.accountCanonicalForm = 3
ldap.server2.baseDn = "CN=Users,DC=w,DC=net"
```

The above configuration will instruct `Zend\Authentication\Adapter\Ldap` to attempt to authenticate users

with the OpenLDAP server `s0.foo.net` first. If the authentication fails for any reason, the AD server `dc1.w.net` will be tried.

With servers in different domains, this configuration illustrates multi-domain authentication. You can also have multiple servers in the same domain to provide redundancy.

Note that in this case, even though OpenLDAP has no need for the short NetBIOS style domain name used by Windows, we provide it here for name canonicalization purposes (described in the **Username Canonicalization** section below).

29.3 The API

The `Zend\Authentication\Adapter\Ldap` constructor accepts three parameters.

The `$options` parameter is required and must be an array containing one or more sets of options. Note that it is **an array of arrays** of *Zend\Ldap\Ldap* options. Even if you will be using only one *LDAP* server, the options must still be within another array.

Below is print_r() output of an example options parameter containing two sets of server options for *LDAP* servers `s0.foo.net` and `dc1.w.net` (the same options as the above *INI* representation):

```
Array
(
    [server2] => Array
        (
            [host] => dc1.w.net
            [useStartTls] => 1
            [accountDomainName] => w.net
            [accountDomainNameShort] => W
            [accountCanonicalForm] => 3
            [baseDn] => CN=Users,DC=w,DC=net
        )

    [server1] => Array
        (
            [host] => s0.foo.net
            [accountDomainName] => foo.net
            [accountDomainNameShort] => FOO
            [accountCanonicalForm] => 3
            [username] => CN=user1,DC=foo,DC=net
            [password] => pass1
            [baseDn] => OU=Sales,DC=foo,DC=net
            [bindRequiresDn] => 1
        )

)
```

The information provided in each set of options above is different mainly because AD does not require a username be in DN form when binding (see the `bindRequiresDn` option in the **Server Options** section below), which means we can omit a number of options associated with retrieving the DN for a username being authenticated.

Note: What is a Distinguished Name?

A DN or "distinguished name" is a string that represents the path to an object within the *LDAP* directory. Each comma-separated component is an attribute and value representing a node. The components are evaluated in reverse. For example, the user account **CN=Bob Carter,CN=Users,DC=w,DC=net** is located directly within the

CN=Users,DC=w,DC=net container. This structure is best explored with an *LDAP* browser like the *ADSI* Edit *MMC* snap-in for Active Directory or phpLDAPadmin.

The names of servers (e.g. 'server1' and 'server2' shown above) are largely arbitrary, but for the sake of using `Zend\Config\Reader\Ini`, the identifiers should be present (as opposed to being numeric indexes) and should not contain any special characters used by the associated file formats (e.g. the '**.**'*INI* property separator, '**&**' for *XML* entity references, etc).

With multiple sets of server options, the adapter can authenticate users in multiple domains and provide failover so that if one server is not available, another will be queried.

Note: The Gory Details: What Happens in the Authenticate Method?

When the `authenticate()` method is called, the adapter iterates over each set of server options, sets them on the internal `Zend\Ldap\Ldap` instance, and calls the `Zend\Ldap\Ldap::bind()` method with the username and password being authenticated. The `Zend\Ldap\Ldap` class checks to see if the username is qualified with a domain (e.g., has a domain component like `alice@foo.net` or `FOO\alice`). If a domain is present, but does not match either of the server's domain names (`foo.net` or *FOO*), a special exception is thrown and caught by `Zend\Authentication\Adapter\Ldap` that causes that server to be ignored and the next set of server options is selected. If a domain **does** match, or if the user did not supply a qualified username, `Zend\Ldap\Ldap` proceeds to try to bind with the supplied credentials. if the bind is not successful, `Zend\Ldap\Ldap` throws a `Zend\Ldap\Exception\LdapException` which is caught by `Zend\Authentication\Adapter\Ldap` and the next set of server options is tried. If the bind is successful, the iteration stops, and the adapter's `authenticate()` method returns a successful result. If all server options have been tried without success, the authentication fails, and `authenticate()` returns a failure result with error messages from the last iteration.

The username and password parameters of the `Zend\Authentication\Adapter\Ldap` constructor represent the credentials being authenticated (i.e., the credentials supplied by the user through your *HTML* login form). Alternatively, they may also be set with the `setUsername()` and `setPassword()` methods.

29.4 Server Options

Each set of server options **in the context of Zend\Authentication\Adapter\Ldap** consists of the following options, which are passed, largely unmodified, to `Zend\Ldap\Ldap::setOptions()`:

Table 29.1: Server Options

Name	Description
host	The hostname of LDAP server that these options represent. This option is required.
port	The port on which the LDAP server is listening. If useSsl is TRUE, the default port value is 636. If useSsl is FALSE, the default port value is 389.
useStartTls	Whether or not the LDAP client should use TLS (aka SSLv2) encrypted transport. A value of TRUE is strongly favored in production environments to prevent passwords from be transmitted in clear text. The default value is FALSE, as servers frequently require that a certificate be installed separately after installation. The useSsl and useStartTls options are mutually exclusive. The useStartTls option should be favored over useSsl but not all servers support this newer mechanism.
useSsl	Whether or not the LDAP client should use SSL encrypted transport. The useSsl and useStartTls options are mutually exclusive, but useStartTls should be favored if the server and LDAP client library support it. This value also changes the default port value (see port description above).
username	The DN of the account used to perform account DN lookups. LDAP servers that require the username to be in DN form when performing the "bind" require this option. Meaning, if bindRequiresDn is TRUE, this option is required. This account does not need to be a privileged account; an account with read-only access to objects under the baseDn is all that is necessary (and preferred based on the Principle of Least Privilege).
password	The password of the account used to perform account DN lookups. If this option is not supplied, the LDAP client will attempt an "anonymous bind" when performing account DN lookups.
bindRequiresDn	Some LDAP servers require that the username used to bind be in DN form like CN=Alice Baker,OU=Sales,DC=foo,DC=net (basically all servers except AD). If this option is TRUE, this instructs Zend\Ldap\Ldap to automatically retrieve the DN corresponding to the username being authenticated, if it is not already in DN form, and then re-bind with the proper DN. The default value is FALSE. Currently only Microsoft Active Directory Server (ADS) is known not to require usernames to be in DN form when binding, and therefore this option may be FALSE with AD (and it should be, as retrieving the DN requires an extra round trip to the server). Otherwise, this option must be set to TRUE (e.g. for OpenLDAP). This option also controls the default accountFilterFormat used when searching for accounts. See the accountFilterFormat option.
baseDn	The DN under which all accounts being authenticated are located. This option is required. if you are uncertain about the correct baseDn value, it should be sufficient to derive it from the user's DNS domain using DC= components. For example, if the user's principal name is alice@foo.net, a baseDn of DC=foo,DC=net should work. A more precise location (e.g., OU=Sales,DC=foo,DC=net) will be more efficient, however.
accountCanonicalForm	A value of 2, 3 or 4 indicating the form to which account names should be canonicalized after successful authentication. Values are as follows: 2 for traditional username style names (e.g., alice), 3 for backslash-style names (e.g., FOO\alice) or 4 for principal style usernames (e.g., alice@foo.net). The default value is 4 (e.g., alice@foo.net). For example, with a value of 3, the identity returned by Zend\Authentication\Result::getIdentity() (and Zend\Authentication\AuthenticationService::getIdentity(), if Zend\Authentication\AuthenticationService was used) will always be FOO\alice, regardless of what form Alice supplied, whether it be alice, alice@foo.net, FOO\alice, FoO\aLicE, foo.net\alice, etc. See the Account Name Canonicalization section in the Zend\Ldap\Ldap documentation for details. Note that when using multiple sets of server options it is recommended, but not required, that the same accountCanonicalForm be used with all server options so that the resulting usernames are always canonicalized to the same form (e.g., if you canonicalize to EXAMPLE\username with an AD server but to username@example.com with an OpenLDAP server, that may be awkward for the application's high-level logic).
accountDomainName	The FQDN domain name for which the target LDAP server is an authority (e.g., example.com). This option is used to canonicalize names so that the username supplied by the user can be converted as necessary for binding. It is also used to determine if the server is an authority for the supplied username (e.g., if accountDomainName is foo.net and the user supplies bob@bar.net, the server will not be queried, and a failure will result). This option is not required, but if it is not supplied, usernames in principal name form (e.g., alice@foo.net) are not supported. It is strongly recommended that you supply this option, as there are many use-cases that require generating the principal name form.

Note: If you enable **useStartTls = TRUE** or **useSsl = TRUE** you may find that the *LDAP* client generates an error claiming that it cannot validate the server's certificate. Assuming the *PHP LDAP* extension is ultimately linked to the OpenLDAP client libraries, to resolve this issue you can set "`TLS_REQCERT never`" in the OpenLDAP client `ldap.conf` (and restart the web server) to indicate to the OpenLDAP client library that you trust the server. Alternatively, if you are concerned that the server could be spoofed, you can export the *LDAP* server's root certificate and put it on the web server so that the OpenLDAP client can validate the server's identity.

29.5 Collecting Debugging Messages

`Zend\Authentication\Adapter\Ldap` collects debugging information within its `authenticate()` method. This information is stored in the `Zend\Authentication\Result` object as messages. The array returned by `Zend\Authentication\Result::getMessages()` is described as follows

Table 29.2: Debugging Messages

Messages Array Index	Description
Index 0	A generic, user-friendly message that is suitable for displaying to users (e.g., "Invalid credentials"). If the authentication is successful, this string is empty.
Index 1	A more detailed error message that is not suitable to be displayed to users but should be logged for the benefit of server operators. If the authentication is successful, this string is empty.
Indexes 2 and higher	All log messages in order starting at index 2.

In practice, index 0 should be displayed to the user (e.g., using the FlashMessenger helper), index 1 should be logged and, if debugging information is being collected, indexes 2 and higher could be logged as well (although the final message always includes the string from index 1).

29.6 Common Options for Specific Servers

29.6.1 Options for Active Directory

For *ADS*, the following options are noteworthy:

Table 29.3: Options for Active Directory

Name	Additional Notes
host	As with all servers, this option is required.
useStartTls	For the sake of security, this should be TRUE if the server has the necessary certificate installed.
useSsl	Possibly used as an alternative to useStartTls (see above).
baseDn	As with all servers, this option is required. By default AD places all user accounts under the Users container (e.g., CN=Users,DC=foo,DC=net), but the default is not common in larger organizations. Ask your AD administrator what the best DN for accounts for your application would be.
accountCanonicalForm	You almost certainly want this to be 3 for backslash style names (e.g., FOO\alice), which are most familiar to Windows users. You should not use the unqualified form 2 (e.g., alice), as this may grant access to your application to users with the same username in other trusted domains (e.g., BAR\alice and FOO\alice will be treated as the same user). (See also note below.)
accountDomainName	This is required with AD unless accountCanonicalForm 2 is used, which, again, is discouraged.
accountDomainNameShort	The NetBIOS name of the domain that users are in and for which the AD server is an authority. This is required if the backslash style accountCanonicalForm is used.

Note: Technically there should be no danger of accidental cross-domain authentication with the current `Zend\Authentication\Adapter\Ldap` implementation, since server domains are explicitly checked, but this may not be true of a future implementation that discovers the domain at runtime, or if an alternative adapter is used (e.g., Kerberos). In general, account name ambiguity is known to be the source of security issues, so always try to use qualified account names.

29.6.2 Options for OpenLDAP

For OpenLDAP or a generic *LDAP* server using a typical posixAccount style schema, the following options are noteworthy:

Table 29.4: Options for OpenLDAP

Name	Additional Notes
host	As with all servers, this option is required.
useStartTls	For the sake of security, this should be TRUE if the server has the necessary certificate installed.
useSsl	Possibly used as an alternative to useStartTls (see above).
username	Required and must be a DN, as OpenLDAP requires that usernames be in DN form when performing a bind. Try to use an unprivileged account.
password	The password corresponding to the username above, but this may be omitted if the LDAP server permits an anonymous binding to query user accounts.
bindRequiresDn	Required and must be TRUE, as OpenLDAP requires that usernames be in DN form when performing a bind.
baseDn	As with all servers, this option is required and indicates the DN under which all accounts being authenticated are located.
accountCanonicalForm	Optional, but the default value is 4 (principal style names like alice@foo.net), which may not be ideal if your users are used to backslash style names (e.g., FOO\alice). For backslash style names use value 3.
accountDomainName	Required unless you're using accountCanonicalForm 2, which is not recommended.
accountDomainNameShort	If AD is not also being used, this value is not required. Otherwise, if accountCanonicalForm 3 is used, this option is required and should be a short name that corresponds adequately to the accountDomainName (e.g., if your accountDomainName is foo.net, a good accountDomainNameShort value might be FOO).

CHAPTER

THIRTY

AUTHENTICATION VALIDATOR

30.1 Introduction

`Zend\Authentication\Validator\Authentication` provides the ability to utilize a validator for an InputFilter in the instance of a Form or for single use where you simply want a true/false value and being able to introspect the error.

The available configuration options include:

- **adapter**: This is an instance of `Zend\Authentication\Adapter`.
- **identity**: This is the identity or name of the identity in the passed in context.
- **credential**: This is the credential or the name of the credential in the passed in context.
- **service**: This is an instance of `Zend\Authentication\AuthenticationService`

30.2 Basic Usage

```
use Zend\Authentication\AuthenticationService;
use Zend\Authentication\Validator\Authentication as AuthenticationValidator;

$service = new AuthenticationService();
$adapter = new My\Authentication\Adapter();
$validator = new AuthenticationValidator(
    'service' => $service,
    'adapter' => $adapter,
);

$validator->setCredential('myCredentialContext');
$validator->isValid('myIdentity', array(
     'myCredentialContext' => 'myCredential',
));
```

CHAPTER

THIRTYONE

INTRODUCTION TO ZEND\BARCODE

`Zend\Barcode\Barcode` provides a generic way to generate barcodes. The `Zend\Barcode` component is divided into two subcomponents: barcode objects and renderers. Objects allow you to create barcodes independently of the renderer. Renderer allow you to draw barcodes based on the support required.

CHAPTER

THIRTYTWO

BARCODE CREATION USING ZEND\BARCODE\BARCODE CLASS

32.1 Using Zend\Barcode\Barcode::factory

`Zend\Barcode\Barcode` uses a factory method to create an instance of a renderer that extends `Zend\Barcode\Renderer\AbstractRenderer`. The factory method accepts five arguments.

- The name of the barcode format (e.g., "code39") or a Traversable object (required)
- The name of the renderer (e.g., "image") (required)
- Options to pass to the barcode object (an array or a Traversable object) (optional)
- Options to pass to the renderer object (an array or a Traversable object) (optional)
- Boolean to indicate whether or not to automatically render errors. If an exception occurs, the provided barcode object will be replaced with an Error representation (optional default `TRUE`)

Getting a Renderer with Zend\Barcode\Barcode::factory()

`Zend\Barcode\Barcode::factory()` instantiates barcode classes and renderers and ties them together. In this first example, we will use the **Code39** barcode type together with the **Image** renderer.

```
use Zend\Barcode\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();
$renderer = Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

Using Zend\Barcode\Barcode::factory() with Zend\Config\Config objects

You may pass a `Zend\Config\Config` object to the factory in order to create the necessary objects. The following example is functionally equivalent to the previous.

```
use Zend\Config\Config;
use Zend\Barcode\Barcode;

// Using only one Zend\Config\Config object
$config = new Config(array(
    'barcode'        => 'code39',
    'barcodeParams'  => array('text' => 'ZEND-FRAMEWORK'),
    'renderer'       => 'image',
    'rendererParams' => array('imageType' => 'gif'),
));

$renderer = Barcode::factory($config);
```

32.2 Drawing a barcode

When you **draw** the barcode, you retrieve the resource in which the barcode is drawn. To draw a barcode, you can call the `draw()` of the renderer, or simply use the proxy method provided by `Zend\Barcode\Barcode`.

Drawing a barcode with the renderer object

```
use Zend\Barcode\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
$imageResource = Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
)->draw();
```

Drawing a barcode with Zend\Barcode\Barcode::draw()

```
use Zend\Barcode\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
$imageResource = Barcode::draw(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

32.3 Rendering a barcode

When you render a barcode, you draw the barcode, you send the headers and you send the resource (e.g. to a browser). To render a barcode, you can call the `render()` method of the renderer or simply use the proxy method provided by `Zend\Barcode\Barcode`.

Rendering a barcode with the renderer object

```
use Zend\Barcode\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
// send the headers and the image
Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
)->render();
```

This will generate this barcode:

Rendering a barcode with Zend\Barcode\Barcode::render()

```
use Zend\Barcode\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
// send the headers and the image
Barcode::render(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

This will generate the same barcode as the previous example.

CHAPTER

THIRTYTHREE

ZEND\BARCODE\BARCODE OBJECTS

Barcode objects allow you to generate barcodes independently of the rendering support. After generation, you can retrieve the barcode as an array of drawing instructions that you can provide to a renderer.

Objects have a large number of options. Most of them are common to all objects. These options can be set in three ways:

- As an array or a Traversable object passed to the constructor.
- As an array passed to the `setOptions()` method.
- Via individual setters for each configuration type.

Different ways to parameterize a barcode object

```
use Zend\Barcode\Object;

$options = array('text' => 'ZEND-FRAMEWORK', 'barHeight' => 40);

// Case 1: constructor
$barcode = new Object\Code39($options);

// Case 2: setOptions()
$barcode = new Object\Code39();
$barcode->setOptions($options);

// Case 3: individual setters
$barcode = new Object\Code39();
$barcode->setText('ZEND-FRAMEWORK')
        ->setBarHeight(40);
```

33.1 Common Options

In the following list, the values have no units; we will use the term "unit." For example, the default value of the "thin bar" is "1 unit". The real units depend on the rendering support (see *the renderers documentation* for more information). Setters are each named by uppercasing the initial letter of the option and prefixing the name with "set" (e.g. "barHeight" becomes "setBarHeight"). All options have a corresponding getter prefixed with "get" (e.g. "getBarHeight"). Available options are:

Table 33.1: Common Options

Option	Data Type	Default Value	Description
barcodeNamespace	String	Zend\Barcode\Object	Namespace of the barcode; for example, if you need to extend the embedding objects
barHeight	Integer	50	Height of the bars
barThickWidth	Integer	3	Width of the thick bar
barThinWidth	Integer	1	Width of the thin bar
factor	Integer	1	Factor by which to multiply bar widths and font sizes (barHeight, barThinWidth, barThickWidth and fontSize)
foreColor	Integer	0x000000 (black)	Color of the bar and the text. Could be provided as an integer or as a HTML value (e.g. "#333333")
backgroundColor	Integer or String	0xFFFFFF (white)	Color of the background. Could be provided as an integer or as a HTML value (e.g. "#333333")
orientation	Float	0	Orientation of the barcode
font	String or Integer	NULL	Font path to a TTF font or a number between 1 and 5 if using image generation with GD (internal fonts)
fontSize	Float	10	Size of the font (not applicable with numeric fonts)
withBorder	Boolean	FALSE	Draw a border around the barcode and the quiet zones
withQuietZones	Boolean	TRUE	Leave a quiet zone before and after the barcode
drawText	Boolean	TRUE	Set if the text is displayed below the barcode
stretchText	Boolean	FALSE	Specify if the text is stretched all along the barcode
withChecksum	Boolean	FALSE	Indicate whether or not the checksum is automatically added to the barcode
withChecksumInText	Boolean	FALSE	Indicate whether or not the checksum is displayed in the textual representation
text	String	NULL	The text to represent as a barcode

33.1.1 Particular case of static setBarcodeFont()

You can set a common font for all your objects by using the static method `Zend\Barcode\Barcode::setBarcodeFont()`. This value can be always be overridden for individual objects by using the `setFont()` method.

```
use Zend\Barcode\Barcode;

// In your bootstrap:
Barcode::setBarcodeFont('my_font.ttf');

// Later in your code:
Barcode::render(
    'code39',
    'pdf',
    array('text' => 'ZEND-FRAMEWORK')
); // will use 'my_font.ttf'

// or:
Barcode::render(
    'code39',
    'image',
```

```
    array(
        'text' => 'ZEND-FRAMEWORK',
        'font' => 3
    )
); // will use the 3rd GD internal font
```

33.2 Common Additional Getters

Table 33.2: Common Getters

Getter	Data Type	Description
getType()	String	Return the name of the barcode class without the namespace (e.g. Zend\Barcode\Object\Code39 returns simply "code39")
getRawText()	String	Return the original text provided to the object
getTextToDisplay()	String	Return the text to display, including, if activated, the checksum value
getQuietZone()	Integer	Return the size of the space needed before and after the barcode without any drawing
getInstructions()	Array	Return drawing instructions as an array.
getHeight($recalculate = false)	Integer	Return the height of the barcode calculated after possible rotation
getWidth($recalculate = false)	Integer	Return the width of the barcode calculated after possible rotation
getOffsetTop($recalculate = false)	Integer	Return the position of the top of the barcode calculated after possible rotation
getOffsetLeft($recalculate = false)	Integer	Return the position of the left of the barcode calculated after possible rotation

CHAPTER

THIRTYFOUR

DESCRIPTION OF SHIPPED BARCODES

You will find below detailed information about all barcode types shipped by default with Zend Framework.

34.1 Zend\Barcode\Object\Error

This barcode is a special case. It is internally used to automatically render an exception caught by the `Zend\Barcode` component.

34.2 Zend\Barcode\Object\Code128

- **Name:** Code 128
- **Allowed characters:** the complete ASCII-character set
- **Checksum:** optional (modulo 103)
- **Length:** variable

There are no particular options for this barcode.

34.3 Zend\Barcode\Object\Codabar

- **Name:** Codabar (or Code 2 of 7)
- **Allowed characters:** '0123456789-$:/.+' with 'ABCD' as start and stop characters
- **Checksum:** none
- **Length:** variable

There are no particular options for this barcode.

34.4 Zend\Barcode\Object\Code25

- **Name:** Code 25 (or Code 2 of 5 or Code 25 Industrial)
- **Allowed characters:** '0123456789'
- **Checksum:** optional (modulo 10)
- **Length:** variable

There are no particular options for this barcode.

34.5 Zend\Barcode\Object\Code25interleaved

This barcode extends `Zend\Barcode\Object\Code25` (Code 2 of 5), and has the same particulars and options, and adds the following:

- **Name:** Code 2 of 5 Interleaved
- **Allowed characters:** '0123456789'
- **Checksum:** optional (modulo 10)
- **Length:** variable (always even number of characters)

Available options include:

Table 34.1: Zend\Barcode\Object\Code25interleaved Options

Option	Data Type	Default Value	Description
withBearerBars	Boolean	FALSE	Draw a thick bar at the top and the bottom of the barcode.

Note: If the number of characters is not even, `Zend\Barcode\Object\Code25interleaved` will automatically prepend the missing zero to the barcode text.

34.6 Zend\Barcode\Object\Ean2

This barcode extends `Zend\Barcode\Object\Ean5` (*EAN* 5), and has the same particulars and options, and adds the following:

- **Name:** *EAN*-2
- **Allowed characters:** '0123456789'
- **Checksum:** only use internally but not displayed
- **Length:** 2 characters

There are no particular options for this barcode.

Note: If the number of characters is lower than 2, `Zend\Barcode\Object\Ean2` will automatically prepend the missing zero to the barcode text.

34.7 Zend\Barcode\Object\Ean5

This barcode extends `Zend\Barcode\Object\Ean13` (*EAN* 13), and has the same particulars and options, and adds the following:

- **Name:** *EAN*-5
- **Allowed characters:**'0123456789'
- **Checksum:** only use internally but not displayed
- **Length:** 5 characters

There are no particular options for this barcode.

Note: If the number of characters is lower than 5, `Zend\Barcode\Object\Ean5` will automatically prepend the missing zero to the barcode text.

34.8 Zend\Barcode\Object\Ean8

This barcode extends `Zend\Barcode\Object\Ean13` (*EAN* 13), and has the same particulars and options, and adds the following:

- **Name:** *EAN*-8
- **Allowed characters:**'0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 8 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 8, `Zend\Barcode\Object\Ean8` will automatically prepend the missing zero to the barcode text.

34.9 Zend\Barcode\Object\Ean13

- **Name:** *EAN*-13
- **Allowed characters:**'0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 13 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 13, `Zend\Barcode\Object\Ean13` will automatically prepend the missing zero to the barcode text.

The option `withQuietZones` has no effect with this barcode.

34.10 Zend\Barcode\Object\Code39

- **Name:** Code 39
- **Allowed characters:** '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ -.$/+%'
- **Checksum:** optional (modulo 43)
- **Length:** variable

Note: `Zend\Barcode\Object\Code39` will automatically add the start and stop characters ('*') for you.

There are no particular options for this barcode.

34.11 Zend\Barcode\Object\Identcode

This barcode extends `Zend\Barcode\Object\Code25interleaved` (Code 2 of 5 Interleaved), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** Identcode (Deutsche Post Identcode)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10 different from Code25)
- **Length:** 12 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 12, `Zend\Barcode\Object\Identcode` will automatically prepend missing zeros to the barcode text.

34.12 Zend\Barcode\Object\Itf14

This barcode extends `Zend\Barcode\Object\Code25interleaved` (Code 2 of 5 Interleaved), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** *ITF*-14
- **Allowed characters:** '0123456789'

- **Checksum:** mandatory (modulo 10)
- **Length:** 14 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 14, `Zend\Barcode\Object\Itf14` will automatically prepend missing zeros to the barcode text.

34.13 Zend\Barcode\Object\Leitcode

This barcode extends `Zend\Barcode\Object\Identcode` (Deutsche Post Identcode), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** Leitcode (Deutsche Post Leitcode)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10 different from Code25)
- **Length:** 14 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 14, `Zend\Barcode\Object\Leitcode` will automatically prepend missing zeros to the barcode text.

34.14 Zend\Barcode\Object\Planet

- **Name:** Planet (PostaL Alpha Numeric Encoding Technique)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 12 or 14 characters (including checksum)

There are no particular options for this barcode.

34.15 Zend\Barcode\Object\Postnet

- **Name:** Postnet (POSTal Numeric Encoding Technique)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 6, 7, 10 or 12 characters (including checksum)

There are no particular options for this barcode.

34.16 Zend\Barcode\Object\Royalmail

- **Name:** Royal Mail or *RM4SCC* (Royal Mail 4-State Customer Code)
- **Allowed characters:** '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'
- **Checksum:** mandatory
- **Length:** variable

There are no particular options for this barcode.

34.17 Zend\Barcode\Object\Upca

This barcode extends `Zend\Barcode\Object\Ean13` (*EAN*-13), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** *UPC*-A (Universal Product Code)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 12 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 12, `Zend\Barcode\Object\Upca` will automatically prepend missing zeros to the barcode text.

The option `withQuietZones` has no effect with this barcode.

34.18 Zend\Barcode\Object\Upce

This barcode extends `Zend\Barcode\Object\Upca` (*UPC*-A), and inherits some of its capabilities; it also has a few particulars of its own. The first character of the text to encode is the system (0 or 1).

- **Name:** *UPC*-E (Universal Product Code)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 8 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 8, `Zend\Barcode\Object\Upce` will automatically prepend missing zeros to the barcode text.

Note: If the first character of the text to encode is not 0 or 1, `Zend\Barcode\Object\Upce` will automatically replace it by 0.

The option `withQuietZones` has no effect with this barcode.

CHAPTER

THIRTYFIVE

ZEND\BARCODE RENDERERS

Renderers have some common options. These options can be set in three ways:

- As an array or a Traversable object passed to the constructor.
- As an array passed to the `setOptions()` method.
- As discrete values passed to individual setters.

Different ways to parameterize a renderer object

```
use Zend\Barcode\Renderer;

$options = array('topOffset' => 10);

// Case 1
$renderer = new Renderer\Pdf($options);

// Case 2
$renderer = new Renderer\Pdf();
$renderer->setOptions($options);

// Case 3
$renderer = new Renderer\Pdf();
$renderer->setTopOffset(10);
```

35.1 Common Options

In the following list, the values have no unit; we will use the term "unit." For example, the default value of the "thin bar" is "1 unit." The real units depend on the rendering support. The individual setters are obtained by uppercasing the initial letter of the option and prefixing the name with "set" (e.g. "barHeight" => "setBarHeight"). All options have a correspondent getter prefixed with "get" (e.g. "getBarHeight"). Available options are:

Table 35.1: Common Options

Option	Data Type	Default Value	Description
rendererNamespace	String	Zend\Barcode\Renderer	Namespace of the renderer; for example, if you need to extend the renderers
horizontalPosition	String	"left"	Can be "left", "center" or "right". Can be useful with PDF or if the setWidth() method is used with an image renderer.
verticalPosition	String	"top"	Can be "top", "middle" or "bottom". Can be useful with PDF or if the setHeight() method is used with an image renderer.
leftOffset	Integer	0	Top position of the barcode inside the renderer. If used, this value will override the "horizontalPosition" option.
topOffset	Integer	0	Top position of the barcode inside the renderer. If used, this value will override the "verticalPosition" option.
automaticRenderError	Boolean	FALSE	Whether or not to automatically render errors. If an exception occurs, the provided barcode object will be replaced with an Error representation. Note that some errors (or exceptions) can not be rendered.
moduleSize	Float	1	Size of a rendering module in the support.
barcode	Zend\Barcode\Object	NULL	The barcode object to render.

An additional getter exists: `getType()`. It returns the name of the renderer class without the namespace (e.g. `Zend\Barcode\Renderer\Image` returns "image").

35.2 Zend\Barcode\Renderer\Image

The Image renderer will draw the instruction list of the barcode object in an image resource. The component requires the GD extension. The default width of a module is 1 pixel.

Available options are:

Table 35.2: Zend\Barcode\Renderer\Image Options

Option	Data Type	Default Value	Description
height	Integer	0	Allow you to specify the height of the result image. If "0", the height will be calculated by the barcode object.
width	Integer	0	Allow you to specify the width of the result image. If "0", the width will be calculated by the barcode object.
imageType	String	"png"	Specify the image format. Can be "png", "jpeg", "jpg" or "gif".

35.3 Zend\Barcode\Renderer\Pdf

The *PDF* renderer will draw the instruction list of the barcode object in a *PDF* document. The default width of a module is 0.5 point.

There are no particular options for this renderer.

CHAPTER

THIRTYSIX

ZEND\CACHE\STORAGE\ADAPTER

36.1 Overview

Storage adapters are wrappers for real storage resources such as memory and the filesystem, using the well known adapter pattern.

They come with tons of methods to read, write and modify stored items and to get information about stored items and the storage.

All adapters implement the interface `Zend\Cache\Storage\StorageInterface` and most extend `Zend\Cache\Storage\Adapter\AbstractAdapter`, which comes with basic logic.

Configuration is handled by either `Zend\Cache\Storage\Adapter\AdapterOptions`, or an adapter-specific options class if it exists. You may pass the options instance to the class at instantiation or via the `setOptions()` method, or alternately pass an associative array of options in either place (internally, these are then passed to an options class instance). Alternately, you can pass either the options instance or associative array to the `Zend\Cache\StorageFactory::factory` method.

Note: Many methods throw exceptions

Because many caching operations throw an exception on error, you need to catch them manually or you can use the plug-in `Zend\Cache\Storage\Plugin\ExceptionHandler` with `throw_exceptions` set to `false` to automatically catch them. You can also define an `exception_callback` to log exceptions.

36.2 Quick Start

Caching adapters can either be created from the provided `Zend\Cache\StorageFactory` factory, or by simply instantiating one of the `Zend\Cache\Storage\Adapter\*` classes.

To make life easier, the `Zend\Cache\StorageFactory` comes with a `factory` method to create an adapter and create/add all requested plugins at once.

```
use Zend\Cache\StorageFactory;

// Via factory:
$cache = StorageFactory::factory(array(
    'adapter' => 'apc',
    'plugins' => array(
        'exception_handler' => array('throw_exceptions' => false),
    ),
));
```

```

// Alternately:
$cache  = StorageFactory::adapterFactory('apc');
$plugin = StorageFactory::pluginFactory('exception_handler', array(
    'throw_exceptions' => false,
));
$cache->addPlugin($plugin);

// Or manually:
$cache  = new Zend\Cache\Storage\Adapter\Apc();
$plugin = new Zend\Cache\Storage\Plugin\ExceptionHandler(array(
    'throw_exceptions' => false,
));
$cache->addPlugin($plugin);
```

36.3 Basic Configuration Options

Basic configuration is handled by either `Zend\Cache\Storage\Adapter\AdapterOptions`, or an adapter-specific options class if it exists. You may pass the options instance to the class at instantiation or via the `setOptions()` method, or alternately pass an associative array of options in either place (internally, these are then passed to an options class instance). Alternately, you can pass either the options instance or associative array to the `Zend\Cache\StorageFactory::factory` method.

The following configuration options are defined by `Zend\Cache\Storage\Adapter\AdapterOptions` and are available for every supported adapter. Adapter-specific configuration options are described on adapter level below.

Option	Data Type	Default Value	Description
ttl	`integer`	0	Time to live
namespace	`string`	"zfcache"	The "namespace" in which cache items will live
key_pattern	`null``|``string`	`null`	Pattern against which to validate cache keys
readable	`boolean`	`true`	Enable/Disable reading data from cache
writable	`boolean`	`true`	Enable/Disable writing data to cache

36.4 The StorageInterface

The `Zend\Cache\Storage\StorageInterface` is the basic interface implemented by all storage adapters.

getItem(*string $key*, *boolean & $success = null*, *mixed & $casToken = null*)

Load an item with the given $key.

If item exists set parameter $success to `true`, set parameter $casToken and returns `mixed` value of item.

If item can't load set parameter $success to `false` and returns `null`.

Return type mixed

getItems(*array $keys*)

Load all items given by $keys returning key-value pairs.

Return type array

hasItem(*string $key*)

Test if an item exists.

Return type boolean

hasItems (*array $keys*)
Test multiple items.

Return type string[]

getMetadata (*string $key*)
Get metadata of an item.

Return type array|boolean

getMetadatas (*array $keys*)
Get multiple metadata.

Return type array

setItem (*string $key*, *mixed $value*)
Store an item.

Return type boolean

setItems (*array $keyValuePairs*)
Store multiple items.

Return type boolean

addItem (*string $key*, *mixed $value*)
Add an item.

Return type boolean

addItems (*array $keyValuePairs*)
Add multiple items.

Return type boolean

replaceItem (*string $key*, *mixed $value*)
Replace an item.

Return type boolean

replaceItems (*array $keyValuePairs*)
Replace multiple items.

Return type boolean

checkAndSetItem (*mixed $token*, *string $key*, *mixed $value*)
Set item only if token matches. It uses the token received from `getItem()` to check if the item has changed before overwriting it.

Return type boolean

touchItem (*string $key*)
Reset lifetime of an item.

Return type boolean

touchItems (*array $keys*)
Reset lifetime of multiple items.

Return type boolean

removeItem (*string $key*)
Remove an item.

Return type boolean

removeItems (*array $keys*)
Remove multiple items.

Return type boolean

incrementItem (*string $key*, *int $value*)
Increment an item.

Return type integer|boolean

incrementItems (*array $keyValuePairs*)
Increment multiple items.

Return type boolean

decrementItem (*string $key*, *int $value*)
Decrement an item.

Return type integer|boolean

decrementItems (*array $keyValuePairs*)
Decrement multiple items.

Return type boolean

getCapabilities ()
Capabilities of this storage.

Return type Zend\Cache\Storage\Capabilities

36.5 The AvailableSpaceCapableInterface

The `Zend\Cache\Storage\AvailableSpaceCapableInterface` implements a method to make it possible getting the current available space of the storage.

getAvailableSpace ()
Get available space in bytes.

Return type integer|float

36.6 The TotalSpaceCapableInterface

The `Zend\Cache\Storage\TotalSpaceCapableInterface` implements a method to make it possible getting the total space of the storage.

getTotalSpace ()
Get total space in bytes.

Return type integer|float

36.7 The ClearByNamespaceInterface

The `Zend\Cache\Storage\ClearByNamespaceInterface` implements a method to clear all items of a given namespace.

clearByNamespace (*string $namespace*)
Remove items of given namespace.

Return type boolean

36.8 The ClearByPrefixInterface

The `Zend\Cache\Storage\ClearByPrefixInterface` implements a method to clear all items of a given prefix (within the current configured namespace).

clearByPrefix(*string $prefix*)
Remove items matching given prefix.

Return type boolean

36.9 The ClearExpiredInterface

The `Zend\Cache\Storage\ClearExpiredInterface` implements a method to clear all expired items (within the current configured namespace).

clearExpired()
Remove expired items.

Return type boolean

36.10 The FlushableInterface

The `Zend\Cache\Storage\FlushableInterface` implements a method to flush the complete storage.

flush()
Flush the whole storage.

Return type boolean

36.11 The IterableInterface

The `Zend\Cache\Storage\IterableInterface` implements a method to get an iterator to iterate over items of the storage. It extends `IteratorAggregate` so it's possible to directly iterate over the storage using `foreach`.

getIterator()
Get an Iterator.

Return type Zend\Cache\Storage\IteratorInterface

36.12 The OptimizableInterface

The `Zend\Cache\Storage\OptimizableInterface` implements a method to run optimization processes on the storage.

optimize()
Optimize the storage.

Return type boolean

36.13 The TaggableInterface

The `Zend\Cache\Storage\TaggableInterface` implements methods to mark items with one or more tags and to clean items matching tags.

setTags (*string $key*, *string[] $tags*)
Set tags to an item by given key. (An empty array will remove all tags)

Return type boolean

getTags (*string $key*)
Get tags of an item by given key.

Return type string[]|false

clearByTags (*string[] $tags*, *boolean $disjunction = false*)
Remove items matching given tags.

If $disjunction is `true` only one of the given tags must match else all given tags must match.

Return type boolean

36.14 The Apc Adapter

The `Zend\Cache\Storage\Adapter\Apc` adapter stores cache items in shared memory through the required PHP extension APC (Alternative PHP Cache).

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\ClearByNamespaceInterface`
- `Zend\Cache\Storage\ClearByPrefixInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\IterableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.1: Capabilities

Capability	Value
supportedDatatypes	`null`, `boolean`, `integer`, `double`, `string`, `array` (serialized), `object` (serialized)
supportedMetadata	internal_key, atime, ctime, mtime, rtime, size, hits, ttl
minTtl	1
maxTtl	0
staticTtl	`true`
ttlPrecision	1
useRequestTime	<ini value of `apc.use_request_time`>
expiredRead	`false`
maxKeyLength	5182
namespaceIsPrefix	`true`
namespaceSeparator	<Option value of `namespace_separator`>

Table 36.2: Adapter specific options

Name	Data Type	Default Value	Description
namespace_separator	`string`	":"	A separator for the namespace and prefix

36.15 The Dba Adapter

The `Zend\Cache\Storage\Adapter\Dba` adapter stores cache items into dbm like databases using the required PHP extension dba.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\ClearByNamespaceInterface`
- `Zend\Cache\Storage\ClearByPrefixInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\IterableInterface`
- `Zend\Cache\Storage\OptimizableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.3: Capabilities

Capability	Value
supportedDatatypes	`string`, `null` => `string`, `boolean` => `string`, `integer` => `string`, `double` => `string`
supportedMetadata	<none>
minTtl	0
maxKeyLength	0
namespaceIsPrefix	`true`
namespaceSeparator	<Option value of `namespace_separator`>

Table 36.4: Adapter specific options

Name	Data Type	Default Value	Description
namespace_separator	`string`	":"	A separator for the namespace and prefix
pathname	`string`	""	Pathname to the database file
mode	`string`	"c"	The mode to open the database Please read dba_open for more information
handler	`string`	"flatfile"	The name of the handler which shall be used for accessing the database.

Note: This adapter doesn't support automatically expire items

Because of this adapter doesn't support automatically expire items it's very important to clean outdated items by self.

36.16 The Filesystem Adapter

The `Zend\Cache\Storage\Adapter\Filesystem` adapter stores cache items into the filesystem.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\ClearByNamespaceInterface`
- `Zend\Cache\Storage\ClearByPrefixInterface`
- `Zend\Cache\Storage\ClearExpiredInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\IterableInterface`
- `Zend\Cache\Storage\OptimizableInterface`
- `Zend\Cache\Storage\TaggableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.5: Capabilities

Capability	Value
supportedDatatypes	`string`, `null` => `string`, `boolean` => `string`, `integer` => `string`, `double` => `string`
supportedMetadata	mtime, filespec, atime, ctime
minTtl	1
maxTtl	0
staticTtl	`false`
ttlPrecision	1
useRequestTime	`false`
expiredRead	`true`
maxKeyLength	251
namespaceIsPrefix	`true`
namespaceSeparator	<Option value of `namespace_separator`>

Table 36.6: Adapter specific options

Name	Data Type	Default Value	Description
namespace_separator	`string`	":"	A separator for the namespace and prefix
cache_dir	`string`	""	Directory to store cache files
clear_stat_cache	`boolean`	`true`	Call `clearstatcache()` enabled?
dir_level	`integer`	1	Defines how much sub-directories should be created
dir_permission	`integer` `false`	0700	Set explicit permission on creating new directories
file_locking	`boolean`	`true`	Lock files on writing
file_permission	`integer` `false`	0600	Set explicit permission on creating new files
key_pattern	`string`	`/^[a-z0-9_\+\-]*$/D`	Validate key against pattern
no_atime	`boolean`	`true`	Don't get 'fileatime' as 'atime' on metadata
no_ctime	`boolean`	`true`	Don't get 'filectime' as 'ctime' on metadata
umask	`integer` `false`	`false`	Use umask to set file and directory permissions

36.17 The Memcached Adapter

The `Zend\Cache\Storage\Adapter\Memcached` adapter stores cache items over the memcached protocol. It's using the required PHP extension memcached which is based on Libmemcached.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.7: Capabilities

Capability	Value
supportedDatatypes	`null`, `boolean`, `integer`, `double`, `string`, `array` (serialized), `object` (serialized)
supportedMetadata	<none>
minTtl	1
maxTtl	0
staticTtl	`true`
ttlPrecision	1
useRequestTime	`false`
expiredRead	`false`
maxKeyLength	255
namespaceIsPrefix	`true`
namespaceSeparator	<none>

Table 36.8: Adapter specific options

Name	Data Type	Default Value	Description
servers	`array`	`[]`	List of servers in [] = array(`string` host, `integer` port)
lib_options	`array`	`[]`	Associative array of Libmemcached options were the array key is the option name (without the prefix "OPT_") or the constant value. The array value is the option value Please read this<http://php.net/manual/memcached.setoption.php> for more information

36.18 The Memory Adapter

The `Zend\Cache\Storage\Adapter\Memory` adapter stores cache items into the PHP process using an array.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\ClearByPrefixInterface`
- `Zend\Cache\Storage\ClearExpiredInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\IterableInterface`
- `Zend\Cache\Storage\TaggableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.9: Capabilities

Capability	Value
supportedDatatypes	`string`, `null`, `boolean`, `integer`, `double`, `array`, `object`, `resource`
supportedMetadata	mtime
minTtl	1
maxTtl	<Value of `PHP_INT_MAX`>
staticTtl	`false`
ttlPrecision	0.05
useRequestTime	`false`
expiredRead	`true`
maxKeyLength	0
namespaceIsPrefix	`false`

Table 36.10: Adapter specific options

Name	Data Type	Default Value	Description
memory_limit	`string` `integer`	<50% of ini value `memory_limit`>	Limit of how much memory can PHP allocate to allow store items into this adapter • If the used memory of PHP exceeds this limit an OutOfSpaceException will be thrown. • A number less or equal 0 will disable the memory limit • When a number is used, the value is measured in bytes (Shorthand notation may also be used)

Note: All stored items will be lost after terminating the script.

36.19 The WinCache Adapter

The `Zend\Cache\Storage\Adapter\WinCache` adapter stores cache items into shared memory through the required PHP extension WinCache.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.11: Capabilities

Capability	Value
supportedDatatypes	`null`, `boolean`, `integer`, `double`, `string`, `array` (serialized), `object` (serialized)
supportedMetadata	internal_key, ttl, hits, size
minTtl	1
maxTtl	0
staticTtl	`true`
ttlPrecision	1
useRequestTime	<ini value of `apc.use_request_time`>
expiredRead	`false`
namespaceIsPrefix	`true`
namespaceSeparator	<Option value of `namespace_separator`>

Table 36.12: Adapter specific options

Name	Data Type	Default Value	Description
namespace_separator	`string`	":"	A separator for the namespace and prefix

36.20 The XCache Adapter

The `Zend\Cache\Storage\Adapter\XCache` adapter stores cache items into shared memory through the required PHP extension XCache.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\ClearByNamespaceInterface`
- `Zend\Cache\Storage\ClearByPrefixInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\IterableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.13: Capabilities

Capability	Value
supportedDatatypes	`boolean`, `integer`, `double`, `string`, `array` (serialized), `object` (serialized)
supportedMetadata	internal_key, size, refcount, hits, ctime, atime, hvalue
minTtl	1
maxTtl	<ini value of `xcache.var_maxttl`>
staticTtl	`true`
ttlPrecision	1
useRequestTime	`true`
expiredRead	`false`
maxKeyLength	5182
namespaceIsPrefix	`true`
namespaceSeparator	<Option value of `namespace_separator`>

Table 36.14: Adapter specific options

Name	Data Type	Default Value	Description
namespace_separator	string	":"	A separator for the namespace and prefix
admin_auth	boolean	false	Enable admin authentication by configuration options `admin_user` and `admin_pass` This makes XCache administration functions accessible if `xcache.admin.enable_auth` is enabled without the need of HTTP-Authentication.
admin_user	string	""	The username of `xcache.admin.user`
admin_pass	string	""	The password of `xcache.admin.pass` in plain text

36.21 The ZendServerDisk Adapter

This `Zend\Cache\Storage\Adapter\ZendServerDisk` adapter stores cache items on filesystem through the Zend Server Data Caching API.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\AvailableSpaceCapableInterface`
- `Zend\Cache\Storage\ClearByNamespaceInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.15: Capabilities

Capability	Value
supportedDatatypes	`null`, `boolean`, `integer`, `double`, `string`, `array` (serialized), `object` (serialized)
supportedMetadata	<none>
minTtl	1
maxTtl	0
maxKeyLength	0
staticTtl	`true`
ttlPrecision	1
useRequestTime	`false`
expiredRead	`false`
namespaceIsPrefix	`true`
namespaceSeparator	`::`

36.22 The ZendServerShm Adapter

The `Zend\Cache\Storage\Adapter\ZendServerShm` adapter stores cache items in shared memory through the Zend Server Data Caching API.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\StorageInterface`
- `Zend\Cache\Storage\ClearByNamespaceInterface`
- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 36.16: Capabilities

Capability	Value
supportedDatatypes	`null`, `boolean`, `integer`, `double`, `string`, `array` (serialized), `object` (serialized)
supportedMetadata	<none>
minTtl	1
maxTtl	0
maxKeyLength	0
staticTtl	`true`
ttlPrecision	1
useRequestTime	`false`
expiredRead	`false`
namespaceIsPrefix	`true`
namespaceSeparator	`::`

36.23 Examples

Basic usage

```
$cache   = \Zend\Cache\StorageFactory::factory(array(
    'adapter' => array(
        'name' => 'filesystem'
    ),
    'plugins' => array(
        // Don't throw exceptions on cache errors
        'exception_handler' => array(
            'throw_exceptions' => false
        ),
    )
));
$key    = 'unique-cache-key';
$result = $cache->getItem($key, $success);
if (!$success) {
    $result = doExpansiveStuff();
    $cache->setItem($key, $result);
}
```

Get multiple rows from db

```
// Instantiate the cache instance using a namespace for the same type of items
$cache   = \Zend\Cache\StorageFactory::factory(array(
    'adapter' => array(
        'name'    => 'filesystem'
        // With a namespace we can indicate the same type of items
```

```
        // -> So we can simple use the db id as cache key
        'options' => array(
            'namespace' => 'dbtable'
        ),
    ),
    'plugins' => array(
        // Don't throw exceptions on cache errors
        'exception_handler' => array(
            'throw_exceptions' => false
        ),
        // We store database rows on filesystem so we need to serialize them
        'Serializer'
    )
));

// Load two rows from cache if possible
$ids     = array(1, 2);
$results = $cache->getItems($ids);
if (count($results) < count($ids)) {
    // Load rows from db if loading from cache failed
    $missingIds     = array_diff($ids, array_keys($results));
    $missingResults = array();
    $query          = 'SELECT * FROM dbtable WHERE id IN (' . implode(',', $missingIds) . ')';
    foreach ($pdo->query($query, PDO::FETCH_ASSOC) as $row) {
        $missingResults[ $row['id'] ] = $row;
    }

    // Update cache items of the loaded rows from db
    $cache->setItems($missingResults);

    // merge results from cache and db
    $results = array_merge($results, $missingResults);
}
```

CHAPTER

THIRTYSEVEN

ZEND\CACHE\STORAGE\CAPABILITIES

37.1 Overview

Storage capabilities describes how a storage adapter works and which features it supports.

To get capabilities of a storage adapter, you can use the method `getCapabilities()` of the storage adapter but only the storage adapter and its plugins have permissions to change them.

Because capabilities are mutable, for example, by changing some options, you can subscribe to the "change" event to get notifications; see the examples for details.

If you are writing your own plugin or adapter, you can also change capabilities because you have access to the marker object and can create your own marker to instantiate a new object of `Zend\Cache\Storage\Capabilities`.

37.2 Available Methods

__construct(*Zend\Cache\Storage\StorageInterface $storage*, *stdClass $marker*, *array $capabilities = array()*, *Zend\Cache\Storage\Capabilities|null $baseCapabilities = null*)

Constructor

getSupportedDatatypes()

Get supported datatypes.

Return type array

setSupportedDatatypes(*stdClass $marker*, *array $datatypes*)

Set supported datatypes.

Return type Zend\Cache\Storage\Capabilities

getSupportedMetadata()

Get supported metadata.

Return type array

setSupportedMetadata(*stdClass $marker*, *string $metadata*)

Set supported metadata.

Return type Zend\Cache\Storage\Capabilities

getMinTtl()

Get minimum supported time-to-live.

(Returning 0 means items never expire)

Return type integer

setMinTtl(*stdClass $marker*, *int $minTtl*)
Set minimum supported time-to-live.

Return type Zend\Cache\Storage\Capabilities

getMaxTtl()
Get maximum supported time-to-live.

Return type integer

setMaxTtl(*stdClass $marker*, *int $maxTtl*)
Set maximum supported time-to-live.

Return type Zend\Cache\Storage\Capabilities

getStaticTtl()
Is the time-to-live handled static (on write), or dynamic (on read).

Return type boolean

setStaticTtl(*stdClass $marker*, *boolean $flag*)
Set if the time-to-live is handled statically (on write) or dynamically (on read).

Return type Zend\Cache\Storage\Capabilities

getTtlPrecision()
Get time-to-live precision.

Return type float

setTtlPrecision(*stdClass $marker*, *float $ttlPrecision*)
Set time-to-live precision.

Return type Zend\Cache\Storage\Capabilities

getUseRequestTime()
Get the "use request time" flag status.

Return type boolean

setUseRequestTime(*stdClass $marker*, *boolean $flag*)
Set the "use request time" flag.

Return type Zend\Cache\Storage\Capabilities

getExpiredRead()
Get flag indicating if expired items are readable.

Return type boolean

setExpiredRead(*stdClass $marker*, *boolean $flag*)
Set if expired items are readable.

Return type Zend\Cache\Storage\Capabilities

getMaxKeyLength()
Get maximum key length.

Return type integer

setMaxKeyLength(*stdClass $marker*, *int $maxKeyLength*)
Set maximum key length.

Return type Zend\Cache\Storage\Capabilities

getNamespaceIsPrefix()
Get if namespace support is implemented as a key prefix.

Return type boolean

setNamespaceIsPrefix(*stdClass $marker*, *boolean $flag*)
Set if namespace support is implemented as a key prefix.

Return type Zend\Cache\Storage\Capabilities

getNamespaceSeparator()
Get namespace separator if namespace is implemented as a key prefix.

Return type string

setNamespaceSeparator(*stdClass $marker*, *string $separator*)
Set the namespace separator if namespace is implemented as a key prefix.

Return type Zend\Cache\Storage\Capabilities

37.3 Examples

Get storage capabilities and do specific stuff in base of it

```
use Zend\Cache\StorageFactory;

$cache = StorageFactory::adapterFactory('filesystem');
$supportedDatatypes = $cache->getCapabilities()->getSupportedDatatypes();

// now you can run specific stuff in base of supported feature
if ($supportedDatatypes['object']) {
    $cache->set($key, $object);
} else {
    $cache->set($key, serialize($object));
}
```

Listen to change event

```
use Zend\Cache\StorageFactory;

$cache = StorageFactory::adapterFactory('filesystem', array(
    'no_atime' => false,
));

// Catching capability changes
$cache->getEventManager()->attach('capability', function($event) {
    echo count($event->getParams()) . ' capabilities changed';
});

// change option which changes capabilities
$cache->getOptions()->setNoATime(true);
```

CHAPTER

THIRTYEIGHT

ZEND\CACHE\STORAGE\PLUGIN

38.1 Overview

Cache storage plugins are objects to add missing functionality or to influence behavior of a storage adapter.

The plugins listen to events the adapter triggers and can change called method arguments (*.post - events), skipping and directly return a result (using `stopPropagation`), changing the result (with `setResult` of `Zend\Cache\Storage\PostEvent`) and catching exceptions (with `Zend\Cache\Storage\ExceptionEvent`).

38.2 Quick Start

Storage plugins can either be created from `Zend\Cache\StorageFactory` with the `pluginFactory`, or by simply instantiating one of the `Zend\Cache\Storage\Plugin\*`classes.

To make life easier, the `Zend\Cache\StorageFactory` comes with the method `factory` to create an adapter and all given plugins at once.

```
use Zend\Cache\StorageFactory;

// Via factory:
$cache = StorageFactory::factory(array(
    'adapter' => 'filesystem',
    'plugins' => array('serializer'),
));

// Alternately:
$cache  = StorageFactory::adapterFactory('filesystem');
$plugin = StorageFactory::pluginFactory('serializer');
$cache->addPlugin($plugin);

// Or manually:
$cache  = new Zend\Cache\Storage\Adapter\Filesystem();
$plugin = new Zend\Cache\Storage\Plugin\Serializer();
$cache->addPlugin($plugin);
```

38.3 The ClearExpiredByFactor Plugin

The `Zend\Cache\Storage\Adapter\ClearExpiredByFactor` plugin calls the storage method `clearExpired()` randomly (by factor) after every call of `setItem()`, `setItems()`, `addItem()` and `addItems()`.

Table 38.1: Plugin specific options

Name	Data Type	Default Value	Description
clearing_factor	`integer`	0	The automatic clearing factor

Note: ** The ClearExpiredInterface is required **

The storage have to implement the `Zend\Cache\Storage\ClearExpiredInterface` to work with this plugin.

38.4 The ExceptionHandler Plugin

The `Zend\Cache\Storage\Adapter\ExceptionHandler` plugin catches all exceptions thrown on reading or writing to cache and sends the exception to a defined callback function.

It's configurable if the plugin should re-throw the catched exception.

Table 38.2: Plugin specific options

Name	Data Type	Default Value	Description
exception_callback	`callable` `null`	`null`	Callback will be called on an exception and get the exception as argument
throw_exceptions	`boolean`	`true`	Re-throw catched exceptions

38.5 The IgnoreUserAbort Plugin

The `Zend\Cache\Storage\Adapter\IgnoreUserAbort` plugin ignores script terminations by users until write operations to cache finished.

Table 38.3: Plugin specific options

Name	Data Type	Default Value	Description
exit_on_abort	`boolean`	`true`	Terminate script execution if user abort the script

38.6 The OptimizeByFactor Plugin

The `Zend\Cache\Storage\Adapter\OptimizeByFactor` plugin calls the storage method `optimize()` randomly (by factor) after removing items from cache.

Table 38.4: Plugin specific options

Name	Data Type	Default Value	Description
optimizing_factor	`integer`	0	The automatic optimization factor

Note: ** The OptimizableInterface is required **

The storage have to implement the `Zend\Cache\Storage\OptimizableInterface` to work with this plugin.

38.7 The Serializer Plugin

The `Zend\Cache\Storage\Adapter\Serializer` plugin will serialize data on writing to cache and unserialize on reading. So it's possible to store different datatypes into cache storages only support strings.

Table 38.5: Plugin specific options

Name	Data Type	Default Value	Description
serializer	`null` `string` `Zend\Serializer\Adapter\AdapterInterface`	`null`	The serializer to use • If `null` use the default serializer • If `string` instantiate the serializer with `serializer_options`
serializer_options	`array`	`[]`	Array of serializer options used to instantiate the serializer

38.8 Available Methods

setOptions(*Zend\Cache\Storage\Plugin\PluginOptions $options*)

Set options.

Return type Zend\Cache\Storage\Plugin\PluginInterface

getOptions()

Get options.

Return type Zend\Cache\Storage\Plugin\PluginOptions

attach(*Zend\EventManager\EventManagerInterface $events*)

Defined by `Zend\EventManager\ListenerAggregateInterface`, attach one or more listeners.

Return type void

detach(*Zend\EventManager\EventManagerInterface $events*)

Defined by `Zend\EventManager\ListenerAggregateInterface`, detach all previously attached listeners.

Return type void

38.9 Examples

Basics of writing an own storage plugin

```
use Zend\Cache\Storage\Event;
use Zend\Cache\Storage\Plugin\AbstractPlugin;
use Zend\EventManager\EventManagerInterface;

class MyPlugin extends AbstractPlugin
{

    protected $handles = array();

    // This method have to attach all events required by this plugin
    public function attach(EventManagerInterface $events)
    {
        $this->handles[] = $events->attach('getItem.pre', array($this, 'onGetItemPre'));
        $this->handles[] = $events->attach('getItem.post', array($this, 'onGetItemPost'));
        return $this;
    }

    // This method have to attach all events required by this plugin
    public function detach(EventManagerInterface $events)
    {
        foreach ($this->handles as $handle) {
           $events->detach($handle);
        }
        $this->handles = array();
        return $this;
    }

    public function onGetItemPre(Event $event)
    {
        $params = $event->getParams();
        echo sprintf("Method 'getItem' with key '%s' started\n", params['key']);
    }

    public function onGetItemPost(Event $event)
    {
        $params = $event->getParams();
        echo sprintf("Method 'getItem' with key '%s' finished\n", params['key']);
    }
}

// After defining this basic plugin we can instantiate and add it to an adapter instance
$plugin = new MyPlugin();
$cache->addPlugin($plugin);

// Now on calling getItem our basic plugin should print the expected output
$cache->getItem('cache-key');
// Method 'getItem' with key 'cache-key' started
// Method 'getItem' with key 'cache-key' finished
```

CHAPTER

THIRTYNINE

ZEND\CACHE\PATTERN

39.1 Overview

Cache patterns are configurable objects to solve known performance bottlenecks. Each should be used only in the specific situations they are designed to address. For example you can use one of the `CallbackCache`, `ObjectCache` or `ClassCache` patterns to cache method and function calls; to cache output generation, the `OutputCache` pattern could assist.

All cache patterns implement the same interface, `Zend\Cache\Pattern\PatternInterface`, and most extend the abstract class `Zend\Cache\Pattern\AbstractPattern` to implement basic logic.

Configuration is provided via the `Zend\Cache\Pattern\PatternOptions` class, which can simply be instantiated with an associative array of options passed to the constructor. To configure a pattern object, you can set an instance of `Zend\Cache\Pattern\PatternOptions` with `setOptions`, or provide your options (either as an associative array or `PatternOptions` instance) as the second argument to the factory.

It's also possible to use a single instance of `Zend\Cache\Pattern\PatternOptions` and pass it to multiple pattern objects.

39.2 Quick Start

Pattern objects can either be created from the provided `Zend\Cache\PatternFactory` factory, or, by simply instantiating one of the `Zend\Cache\Pattern\*Cache` classes.

```
// Via the factory:
$callbackCache = Zend\Cache\PatternFactory::factory('callback', array(
    'storage' => 'apc',
));
```

```
// OR, the equivalent manual instantiation:
$callbackCache = new Zend\Cache\Pattern\CallbackCache();
$callbackCache->setOptions(new Zend\Cache\Pattern\PatternOptions(array(
    'storage' => 'apc',
)));
```

39.3 Available Methods

The following methods are implemented by `Zend\Cache\Pattern\AbstractPattern`. Please read documentation of specific patterns to get more information.

setOptions(*Zend\Cache\Pattern\PatternOptions $options*)
Set pattern options.

Return type Zend\Cache\Pattern\PatternInterface

getOptions()
Get all pattern options.

Return type Zend\Cache\Pattern\PatternOptions

CHAPTER

FORTY

ZEND\CACHE\PATTERN\CALLBACKCACHE

40.1 Overview

The callback cache pattern caches calls of non specific functions and methods given as a callback.

40.2 Quick Start

For instantiation you can use the `PatternFactory` or do it manual:

```
use Zend\Cache\PatternFactory;
use Zend\Cache\Pattern\PatternOptions;

// Via the factory:
$callbackCache = PatternFactory::factory('callback', array(
    'storage'      => 'apc',
    'cache_output' => true,
));

// OR, the equivalent manual instantiation:
$callbackCache = new \Zend\Cache\Pattern\CallbackCache();
$callbackCache->setOptions(new PatternOptions(array(
    'storage'      => 'apc',
    'cache_output' => true,
)));
```

40.3 Configuration Options

Option	Data Type	Default Value	Description
storage	`string` `array` `Zend\Cache\Storage\StorageInterface`	<none>	The storage to write/read cached data
cache_output	`boolean`	`true`	Cache output of callback

40.4 Available Methods

call (*callable $callback*, *array $args = array()*)
Call the specified callback or get the result from cache.

Return type mixed

__call (*string $function*, *array $args*)
Function call handler.

Return type mixed

generateKey (*callable $callback*, *array $args = array()*)
Generate a unique key in base of a key representing the callback part and a key representing the arguments part.

Return type string

setOptions (*Zend\Cache\Pattern\PatternOptions $options*)
Set pattern options.

Return type Zend\Cache\Pattern\CallbackCache

getOptions ()
Get all pattern options.

Return type Zend\Cache\Pattern\PatternOptions

40.5 Examples

Instantiating the callback cache pattern

```
use Zend\Cache\PatternFactory;

$callbackCache = PatternFactory::factory('callback', array(
    'storage' => 'apc'
));
```

CHAPTER

FORTYONE

ZEND\CACHE\PATTERN\CLASSCACHE

41.1 Overview

The `ClassCache` pattern is an extension to the `CallbackCache` pattern. It has the same methods but instead it generates the internally used callback in base of the configured class name and the given method name.

41.2 Quick Start

Instantiating the class cache pattern

```
use Zend\Cache\PatternFactory;

$classCache = PatternFactory::factory('class', array(
    'class'   => 'MyClass',
    'storage' => 'apc'
));
```

41.3 Configuration Options

Option	Data Type	Default Value	Description
storage	`string` `array` `Zend\Cache\Storage\StorageInterface`	<none>	The storage to write/read cached data
class	`string`	<none>	The class name
cache_output	`boolean`	`true`	Cache output of callback
cache_by_default	`boolean`	`true`	Cache method calls by default
class_cache_methods	`array`	`[]`	List of methods to cache (If `cache_by_default` is disabled)
class_non_cache_methods	`array`	`[]`	List of methods to no-cache (If `cache_by_default` is enabled)

41.4 Available Methods

`call` (*string $method*, *array $args = array()*)

Call the specified method of the configured class.

Return type mixed

__call(*string $method*, *array $args*)

Call the specified method of the configured class.

Return type mixed

__set(*string $name*, *mixed $value*)

Set a static property of the configured class.

Return type void

__get(*string $name*)

Get a static property of the configured class.

Return type mixed

__isset(*string $name*)

Checks if a static property of the configured class exists.

Return type boolean

__unset(*string $name*)

Unset a static property of the configured class.

Return type void

generateKey(*string $method*, *array $args = array()*)

Generate a unique key in base of a key representing the callback part and a key representing the arguments part.

Return type string

setOptions(*Zend\Cache\Pattern\PatternOptions $options*)

Set pattern options.

Return type Zend\Cache\Pattern\ClassCache

getOptions()

Get all pattern options.

Return type Zend\Cache\Pattern\PatternOptions

41.5 Examples

Caching of import feeds

```
$cachedFeedReader = Zend\Cache\PatternFactory::factory('class', array(
    'class'   => 'Zend\Feed\Reader\Reader',
    'storage' => 'apc',

    // The feed reader doesn't output anything
    // so the output don't need to be caught and cached
    'cache_output' => false,
));

```

```
$feed = $cachedFeedReader->call("import", array('http://www.planet-php.net/rdf/'));
// OR
$feed = $cachedFeedReader->import('http://www.planet-php.net/rdf/');
```

CHAPTER

FORTYTWO

ZEND\CACHE\PATTERN\OBJECTCACHE

42.1 Overview

The `ObjectCache` pattern is an extension to the `CallbackCache` pattern. It has the same methods but instead it generates the internally used callback in base of the configured object and the given method name.

42.2 Quick Start

Instantiating the object cache pattern

```
use Zend\Cache\PatternFactory;

$object      = new stdClass();
$objectCache = PatternFactory::factory('object', array(
    'object'  => $object,
    'storage' => 'apc'
));
```

42.3 Configuration Options

Option	Data Type	Default Value	Description
storage	`string` `array` `Zend\Cache\Storage\StorageInterface`	<none>	The storage to write/read cached data
object	`object`	<none>	The object to cache methods calls of
object_key	`null` `string`	<Class name of object>	A hopefully unique key of the object
cache_output	`boolean`	`true`	Cache output of callback
cache_by_default	`boolean`	`true`	Cache method calls by default
object_cache_methods	`array`	`[]`	List of methods to cache (If `cache_by_default` is disabled)
object_non_cache_methods	`array`	`[]`	List of methods to no-cache (If `cache_by_default` is enabled)
object_cache_magic_properties	`boolean`	`false`	Cache calls of magic object properties

42.4 Available Methods

call (*string $method*, *array $args = array()*)

Call the specified method of the configured object.

Return type mixed

__call (*string $method*, *array $args*)

Call the specified method of the configured object.

Return type mixed

__set (*string $name*, *mixed $value*)

Set a property of the configured object.

Return type void

__get (*string $name*)

Get a property of the configured object.

Return type mixed

__isset (*string $name*)

Checks if static property of the configured object exists.

Return type boolean

__unset (*string $name*)

Unset a property of the configured object.

Return type void

generateKey (*string $method*, *array $args = array()*)

Generate a unique key in base of a key representing the callback part and a key representing the arguments part.

Return type string

setOptions (*Zend\Cache\Pattern\PatternOptions $options*)

Set pattern options.

Return type Zend\Cache\Pattern\ObjectCache

getOptions ()

Get all pattern options.

Return type Zend\Cache\Pattern\PatternOptions

42.5 Examples

Caching a filter

```
$filter       = new Zend\Filter\RealPath();
$cachedFilter = Zend\Cache\PatternFactory::factory('object', array(
    'object'     => $filter,
    'object_key' => 'RealpathFilter',
    'storage'    => 'apc',

    // The realpath filter doesn't output anything
    // so the output don't need to be caught and cached
    'cache_output' => false,
));

$path = $cachedFilter->call("filter", array('/www/var/path/../../mypath'));
// OR
$path = $cachedFilter->filter('/www/var/path/../../mypath');
```

CHAPTER

FORTYTHREE

ZEND\CACHE\PATTERN\OUTPUTCACHE

43.1 Overview

The `OutputCache` pattern caches output between calls to `start()` and `end()`.

43.2 Quick Start

Instantiating the output cache pattern

```
use Zend\Cache\PatternFactory;

$outputCache = PatternFactory::factory('output', array(
    'storage' => 'apc'
));
```

43.3 Configuration Options

Option	Data Type	Default Value	Description
storage	`string` `array` `Zend\Cache\Storage\StorageInterface`	<none>	The storage to write/read cached data

43.4 Available Methods

start (*string $key*)

If there is a cached item with the given key display it's data and return `true` else start buffering output until `end()` is called or the script ends and return `false`.

Return type boolean

end()

Stops buffering output, write buffered data to cache using the given key on `start()` and displays the buffer.

Return type boolean

setOptions (*Zend\Cache\Pattern\PatternOptions $options*)

Set pattern options.

Return type Zend\Cache\Pattern\OutputCache

getOptions()

Get all pattern options.

Return type Zend\Cache\Pattern\PatternOptions

43.5 Examples

Caching simple view scripts

```
$outputCache = Zend\Cache\PatternFactory::factory('output', array(
    'storage' => 'apc',
));

$outputCache->start('mySimpleViewScript');
include '/path/to/view/script.phtml';
$outputCache->end();
```

CHAPTER

FORTYFOUR

ZEND\CACHE\PATTERN\CAPTURECACHE

44.1 Overview

The `CaptureCache` pattern is useful to auto-generate static resources in base of a HTTP request. The Webserver needs to be configured to run a PHP script generating the requested resource so further requests for the same resource can be shipped without calling PHP again.

It comes with basic logic to manage generated resources.

44.2 Quick Start

Simplest usage as Apache-404 handler

```
# .htdocs
ErrorDocument 404 /index.php
```

```
// index.php
use Zend\Cache\PatternFactory;
$capture = Zend\Cache\PatternFactory::factory('capture', array(
    'public_dir' => __DIR__,
));

// Start capturing all output excl. headers and write to public directory
$capture->start();

// Don't forget to change HTTP response code
header('Status: 200', true, 200);

// do stuff to dynamically generate output
```

44.3 Configuration Options

Option	Data Type	Default Value	Description
public_dir	`string`	<none>	Location of public directory to write output to
index_filename	`string`	"index.html"	The name of the first file if only a directory was requested
file_locking	`boolean`	`true`	Locking output files on writing
file_permission	`integer` `boolean`	0600 (`false` on win)	Set permissions of generated output files
dir_permission	`integer` `boolean`	0700 (`false` on win)	Set permissions of generated output directories
umask	`integer` `boolean`	`false`	Using umask on generating output files / directories

44.4 Available Methods

`start` (*string|null $pageId = null*)
Start capturing output.

Return type void

`set` (*string $content, string|null $pageId = null*)
Write content to page identity.

Return type void

`get` (*string|null $pageId = null*)
Get content of an already cached page.

Return type string|false

`has` (*string|null $pageId = null*)
Check if a page has been created.

Return type boolean

`remove` (*string|null $pageId = null*)
Remove a page.

Return type boolean

`clearByGlob` (*string $pattern = '**'*)
Clear pages matching glob pattern.

Return type void

`setOptions` (*Zend\Cache\Pattern\PatternOptions $options*)
Set pattern options.

Return type Zend\Cache\Pattern\CaptureCache

`getOptions` ()
Get all pattern options.

Return type Zend\Cache\Pattern\PatternOptions

44.5 Examples

Scaling images in base of request

```
# .htdocs
ErrorDocument 404 /index.php
```

```
// index.php
$captureCache = Zend\Cache\PatternFactory::factory('capture', array(
    'public_dir' => __DIR__,
));

// TODO
```

CHAPTER

FORTYFIVE

INTRODUCTION TO ZEND\CAPTCHA

CAPTCHA stands for "Completely Automated Public Turing test to tell Computers and Humans Apart"; it is used as a challenge-response to ensure that the individual submitting information is a human and not an automated process. Typically, a captcha is used with form submissions where authenticated users are not necessary, but you want to prevent spam submissions.

Captchas can take a variety of forms, including asking logic questions, presenting skewed fonts, and presenting multiple images and asking how they relate. The `Zend\Captcha` component aims to provide a variety of back ends that may be utilized either standalone or in conjunction with the `Zend\Form` component.

CHAPTER

FORTYSIX

CAPTCHA OPERATION

All *CAPTCHA* adapters implement `Zend\Captcha\AdapterInterface`, which looks like the following:

```
namespace Zend\Captcha;

use Zend\Validator\ValidatorInterface;

interface AdapterInterface extends ValidatorInterface
{
    public function generate();

    public function setName($name);

    public function getName();

    // Get helper name used for rendering this captcha type
    public function getHelperName();
}
```

The name setter and getter are used to specify and retrieve the *CAPTCHA* identifier. The most interesting methods are `generate()` and `render()`. `generate()` is used to create the *CAPTCHA* token. This process typically will store the token in the session so that you may compare against it in subsequent requests. `render()` is used to render the information that represents the *CAPTCHA*, be it an image, a figlet, a logic problem, or some other *CAPTCHA*.

A simple use case might look like the following:

```
// Originating request:
$captcha = new Zend\Captcha\Figlet(array(
    'name' => 'foo',
    'wordLen' => 6,
    'timeout' => 300,
));

$id = $captcha->generate();

//this will output a Figlet string
echo $captcha->getFiglet()->render($captcha->getWord());


// On a subsequent request:
// Assume a captcha setup as before, with corresponding form fields, the value of $_POST['foo']
// would be key/value array: id => captcha ID, input => captcha value
if ($captcha->isValid($_POST['foo'], $_POST)) {
    // Validated!
}
```

Note: Under most circumstances, you probably prefer the use of `Zend\Captcha` functionality combined with the power of the `Zend\Form` component. For an example on how to use `Zend\Form\Element\Captcha`, have a look at the *Zend\Form Quick Start*.

CHAPTER

FORTYSEVEN

CAPTCHA ADAPTERS

The following adapters are shipped with Zend Framework by default.

47.1 Zend\Captcha\AbstractWord

`Zend\Captcha\AbstractWord` is an abstract adapter that serves as the base class for most other *CAPTCHA* adapters. It provides mutators for specifying word length, session *TTL* and the session container object to use. `Zend\Captcha\AbstractWord` also encapsulates validation logic.

By default, the word length is 8 characters, the session timeout is 5 minutes, and `Zend\Session\Container` is used for persistence (using the namespace "`Zend\Form\Captcha\<captcha ID>`").

In addition to the methods required by the `Zend\Captcha\AdapterInterface` interface, `Zend\Captcha\AbstractWord` exposes the following methods:

- `setWordLen($length)` and `getWordLen()` allow you to specify the length of the generated "word" in characters, and to retrieve the current value.
- `setTimeout($ttl)` and `getTimeout()` allow you to specify the time-to-live of the session token, and to retrieve the current value. `$ttl` should be specified in seconds.
- `setUseNumbers($numbers)` and `getUseNumbers()` allow you to specify if numbers will be considered as possible characters for the random work or only letters would be used.
- `setSessionClass($class)` and `getSessionClass()` allow you to specify an alternate `Zend\Session\Container` implementation to use to persist the *CAPTCHA* token and to retrieve the current value.
- `getId()` allows you to retrieve the current token identifier.
- `getWord()` allows you to retrieve the generated word to use with the *CAPTCHA*. It will generate the word for you if none has been generated yet.
- `setSession(Zend\Session\Container $session)` allows you to specify a session object to use for persisting the *CAPTCHA* token. `getSession()` allows you to retrieve the current session object.

All word *CAPTCHA*s allow you to pass an array of options or `Traversable` object to the constructor, or, alternately, pass them to `setOptions()`. By default, the **wordLen**, **timeout**, and **sessionClass** keys may all be used. Each concrete implementation may define additional keys or utilize the options in other ways.

Note: `Zend\Captcha\AbstractWord` is an abstract class and may not be instantiated directly.

47.2 Zend\Captcha\Dumb

The `Zend\Captcha\Dumb` adapter is mostly self-descriptive. It provides a random string that must be typed in reverse to validate. As such, it's not a good *CAPTCHA* solution and should only be used for testing. It extends `Zend\Captcha\AbstractWord`.

47.3 Zend\Captcha\Figlet

The `Zend\Captcha\Figlet` adapter utilizes *Zend\Text\Figlet* to present a figlet to the user.

Options passed to the constructor will also be passed to the *Zend\Text\Figlet* object. See the *Zend\Text\Figlet* documentation for details on what configuration options are available.

47.4 Zend\Captcha\Image

The `Zend\Captcha\Image` adapter takes the generated word and renders it as an image, performing various skewing permutations to make it difficult to automatically decipher. It requires the GD extension compiled with TrueType or Freetype support. Currently, the `Zend\Captcha\Image` adapter can only generate *PNG* images.

`Zend\Captcha\Image` extends `Zend\Captcha\AbstractWord`, and additionally exposes the following methods:

- `setExpiration($expiration)` and `getExpiration()` allow you to specify a maximum lifetime the *CAPTCHA* image may reside on the filesystem. This is typically a longer than the session lifetime. Garbage collection is run periodically each time the *CAPTCHA* object is invoked, deleting all images that have expired. Expiration values should be specified in seconds.
- `setGcFreq($gcFreq)` and `getGcFreq()` allow you to specify how frequently garbage collection should run. Garbage collection will run every `1/$gcFreq` calls. The default is 100.
- `setFont($font)` and `getFont()` allow you to specify the font you will use. `$font` should be a fully qualified path to the font file. This value is required; the *CAPTCHA* will throw an exception during generation if the font file has not been specified.
- `setFontSize($fsize)` and `getFontSize()` allow you to specify the font size in pixels for generating the *CAPTCHA*. The default is 24px.
- `setHeight($height)` and `getHeight()` allow you to specify the height in pixels of the generated *CAPTCHA* image. The default is 50px.
- `setWidth($width)` and `getWidth()` allow you to specify the width in pixels of the generated *CAPTCHA* image. The default is 200px.
- `setImgDir($imgDir)` and `getImgDir()` allow you to specify the directory for storing *CAPTCHA* images. The default is "`./images/captcha/`", relative to the bootstrap script.
- `setImgUrl($imgUrl)` and `getImgUrl()` allow you to specify the relative path to a *CAPTCHA* image to use for *HTML* markup. The default is "`/images/captcha/`".
- `setSuffix($suffix)` and `getSuffix()` allow you to specify the filename suffix for the *CAPTCHA* image. The default is "`.png`". Note: changing this value will not change the type of the generated image.
- `setDotNoiseLevel($level)` and `getDotNoiseLevel()`, along with `setLineNoiseLevel($level)` and `getLineNoiseLevel()`, allow you to control how much "noise" in the form of random dots and lines the image would contain. Each unit of `$level` produces one

random dot or line. The default is 100 dots and 5 lines. The noise is added twice - before and after the image distortion transformation.

All of the above options may be passed to the constructor by simply removing the 'set' method prefix and casting the initial letter to lowercase: "suffix", "height", "imgUrl", etc.

47.5 Zend\Captcha\ReCaptcha

The `Zend\Captcha\ReCaptcha` adapter uses *Zend\Service\ReCaptcha\ReCaptcha* to generate and validate *CAPTCHA*s. It exposes the following methods:

- `setPrivKey($key)` and `getPrivKey()` allow you to specify the private key to use for the ReCaptcha service. This must be specified during construction, although it may be overridden at any point.
- `setPubKey($key)` and `getPubKey()` allow you to specify the public key to use with the ReCaptcha service. This must be specified during construction, although it may be overridden at any point.
- `setService(ZendService\ReCaptcha\ReCaptcha $service)` and `getService()` allow you to set and get the ReCaptcha service object.

CHAPTER

FORTYEIGHT

INTRODUCTION TO ZEND\CONFIG

`Zend\Config` is designed to simplify access to configuration data within applications. It provides a nested object property-based user interface for accessing this configuration data within application code. The configuration data may come from a variety of media supporting hierarchical data storage. Currently, `Zend\Config` provides adapters that read and write configuration data stored in .ini, JSON, YAML and XML files.

48.1 Using Zend\Config\Config with a Reader Class

Normally, it is expected that users would use one of the *reader classes* to read a configuration file, but if configuration data are available in a *PHP* array, one may simply pass the data to `Zend\Config\Config`'s constructor in order to utilize a simple object-oriented interface:

```
// An array of configuration data is given
$configArray = array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        )
    )
);

// Create the object-oriented wrapper using the configuration data
$config = new Zend\Config\Config($configArray);

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;
```

As illustrated in the example above, `Zend\Config\Config` provides nested object property syntax to access configuration data passed to its constructor.

Along with the object-oriented access to the data values, `Zend\Config\Config` also has `get()` method that returns the supplied value if the data element doesn't exist in the configuration array. For example:

```
$host = $config->database->get('host', 'localhost');
```

48.2 Using Zend\Config\Config with a PHP Configuration File

It is often desirable to use a purely *PHP*-based configuration file. The following code illustrates how easily this can be accomplished:

```
// config.php
return array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        )
    )
);
```

```
// Consumes the configuration array
$config = new Zend\Config\Config(include 'config.php');

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;
```

CHAPTER

FORTYNINE

THEORY OF OPERATION

Configuration data are made accessible to `Zend\Config\Config`'s constructor with an associative array, which may be multi-dimensional, so data can be organized from general to specific. Concrete adapter classes adapt configuration data from storage to produce the associative array for `Zend\Config\Config`'s constructor. If needed, user scripts may provide such arrays directly to `Zend\Config\Config`'s constructor, without using a reader class.

Each value in the configuration data array becomes a property of the `Zend\Config\Config` object. The key is used as the property name. If a value is itself an array, then the resulting object property is created as a new `Zend\Config\Config` object, loaded with the array data. This occurs recursively, such that a hierarchy of configuration data may be created with any number of levels.

`Zend\Config\Config` implements the Countable and Iterator interfaces in order to facilitate simple access to configuration data. Thus, `Zend\Config\Config` objects support the count() function and *PHP* constructs such as foreach.

By default, configuration data made available through `Zend\Config\Config` are read-only, and an assignment (e.g. `$config->database->host = 'example.com';`) results in a thrown exception. This default behavior may be overridden through the constructor, allowing modification of data values. Also, when modifications are allowed, `Zend\Config\Config` supports unsetting of values (i.e. `unset($config->database->host)`). The `isReadOnly()` method can be used to determine if modifications to a given `Zend\Config\Config` object are allowed and the `setReadOnly()` method can be used to stop any further modifications to a `Zend\Config\Config` object that was created allowing modifications.

Note: Modifying Config does not save changes

It is important not to confuse such in-memory modifications with saving configuration data out to specific storage media. Tools for creating and modifying configuration data for various storage media are out of scope with respect to `Zend\Config\Config`. Third-party open source solutions are readily available for the purpose of creating and modifying configuration data for various storage media.

If you have two `Zend\Config\Config` objects, you can merge them into a single object using the `merge()` function. For example, given `$config` and `$localConfig`, you can merge data from `$localConfig` to `$config` using `$config->merge($localConfig);`. The items in `$localConfig` will override any items with the same name in `$config`.

Note: The `Zend\Config\Config` object that is performing the merge must have been constructed to allow modifications, by passing `TRUE` as the second parameter of the constructor. The `setReadOnly()` method can then be used to prevent any further modifications after the merge is complete.

CHAPTER

FIFTY

ZEND\CONFIG\READER

`Zend\Config\Reader` gives you the ability to read a config file. It works with concrete implementations for different file format. The `Zend\Config\Reader` is only an interface, that define the two methods `fromFile()` and `fromString()`. The concrete implementations of this interface are:

- `Zend\Config\Reader\Ini`
- `Zend\Config\Reader\Xml`
- `Zend\Config\Reader\Json`
- `Zend\Config\Reader\Yaml`

The `fromFile()` and `fromString()` return a PHP array contains the data of the configuration file.

Note: Differences from ZF1

The `Zend\Config\Reader` component no longer supports the following features:

- Inheritance of sections.
- Reading of specific sections.

50.1 Zend\Config\Reader\Ini

`Zend\Config\Reader\Ini` enables developers to store configuration data in a familiar *INI* format and read them in the application by using an array syntax.

`Zend\Config\Reader\Ini` utilizes the parse_ini_file() *PHP* function. Please review this documentation to be aware of its specific behaviors, which propagate to `Zend\Config\Reader\Ini`, such as how the special values of "`TRUE`", "`FALSE`", "yes", "no", and "`NULL`" are handled.

Note: Key Separator

By default, the key separator character is the period character ("**.**"). This can be changed, however, using the `setNestSeparator()` method. For example:

```
$reader = new Zend\Config\Reader\Ini();
$reader->setNestSeparator('-');
```

The following example illustrates a basic use of `Zend\Config\Reader\Ini` for loading configuration data from an *INI* file. In this example there are configuration data for both a production system and for a staging system. Suppose we have the following INI configuration file:

```
webhost                  = 'www.example.com'
database.adapter         = 'pdo_mysql'
database.params.host     = 'db.example.com'
database.params.username = 'dbuser'
database.params.password = 'secret'
database.params.dbname   = 'dbproduction'
```

We can use the `Zend\Config\Reader\Ini` to read this INI file:

```
$reader = new Zend\Config\Reader\Ini();
$data   = $reader->fromFile('/path/to/config.ini');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

The `Zend\Config\Reader\Ini` supports a feature to include the content of a INI file in a specific section of another INI file. For instance, suppose we have an INI file with the database configuration:

```
database.adapter         = 'pdo_mysql'
database.params.host     = 'db.example.com'
database.params.username = 'dbuser'
database.params.password = 'secret'
database.params.dbname   = 'dbproduction'
```

We can include this configuration in another INI file, for instance:

```
webhost  = 'www.example.com'
@include = 'database.ini'
```

If we read this file using the component `Zend\Config\Reader\Ini` we will obtain the same configuration data structure of the previous example.

The `@include = 'file-to-include.ini'` can be used also in a subelement of a value. For instance we can have an INI file like that:

```
adapter         = 'pdo_mysql'
params.host     = 'db.example.com'
params.username = 'dbuser'
params.password = 'secret'
params.dbname   = 'dbproduction'
```

And assign the `@include` as subelement of the database value:

```
webhost           = 'www.example.com'
database.@include = 'database.ini'
```

50.2 Zend\Config\Reader\Xml

`Zend\Config\Reader\Xml` enables developers to read configuration data in a familiar *XML* format and read them in the application by using an array syntax. The root element of the *XML* file or string is irrelevant and may be named arbitrarily.

The following example illustrates a basic use of `Zend\Config\Reader\Xml` for loading configuration data from an *XML* file. Suppose we have the following *XML* configuration file:

```
<?xml version="1.0" encoding="utf-8"?>?>
<config>
    <webhost>www.example.com</webhost>
```

```
    <database>
        <adapter value="pdo_mysql"/>
        <params>
            <host value="db.example.com"/>
            <username value="dbuser"/>
            <password value="secret"/>
            <dbname value="dbproduction"/>
        </params>
    </database>
</config>
```

We can use the `Zend\Config\Reader\Xml` to read this XML file:

```
$reader = new Zend\Config\Reader\Xml();
$data   = $reader->fromFile('/path/to/config.xml');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

`Zend\Config\Reader\Xml` utilizes the XMLReader *PHP* class. Please review this documentation to be aware of its specific behaviors, which propagate to `Zend\Config\Reader\Xml`.

Using `Zend\Config\Reader\Xml` we can include the content of XML files in a specific XML element. This is provided using the standard function XInclude of XML. To use this function you have to add the namespace `xmlns:xi="http://www.w3.org/2001/XInclude"` to the XML file. Suppose we have an XML files that contains only the database configuration:

```
<?xml version="1.0" encoding="utf-8"?>
<config>
    <database>
        <adapter>pdo_mysql</adapter>
        <params>
            <host>db.example.com</host>
            <username>dbuser</username>
            <password>secret</password>
            <dbname>dbproduction</dbname>
        </params>
    </database>
</config>
```

We can include this configuration in another XML file, for instance:

```
<?xml version="1.0" encoding="utf-8"?>
<config xmlns:xi="http://www.w3.org/2001/XInclude">
    <webhost>www.example.com</webhost>
    <xi:include href="database.xml"/>
</config>
```

The syntax to include an XML file in a specific element is `<xi:include href="file-to-include.xml"/>`

50.3 Zend\Config\Reader\Json

`Zend\Config\Reader\Json` enables developers to read configuration data in a *JSON* format and read them in the application by using an array syntax.

The following example illustrates a basic use of `Zend\Config\Reader\Json` for loading configuration data from a *JSON* file. Suppose we have the following *JSON* configuration file:

```
{
  "webhost"  : "www.example.com",
  "database" : {
    "adapter" : "pdo_mysql",
    "params"  : {
      "host"     : "db.example.com",
      "username" : "dbuser",
      "password" : "secret",
      "dbname"   : "dbproduction"
    }
  }
}
```

We can use the `Zend\Config\Reader\Json` to read this JSON file:

```
$reader = new Zend\Config\Reader\Json();
$data   = $reader->fromFile('/path/to/config.json');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

`Zend\Config\Reader\Json` utilizes the *Zend\Json\Json* class.

Using `Zend\Config\Reader\Json` we can include the content of a JSON file in a specific JSON section or element. This is provided using the special syntax `@include`. Suppose we have a JSON file that contains only the database configuration:

```
{
  "database" : {
    "adapter" : "pdo_mysql",
    "params"  : {
      "host"     : "db.example.com",
      "username" : "dbuser",
      "password" : "secret",
      "dbname"   : "dbproduction"
    }
  }
}
```

We can include this configuration in another JSON file, for instance:

```
{
    "webhost"  : "www.example.com",
    "@include" : "database.json"
}
```

50.4 Zend\Config\Reader\Yaml

`Zend\Config\Reader\Yaml` enables developers to read configuration data in a *YAML* format and read them in the application by using an array syntax. In order to use the YAML reader we need to pass a callback to an external PHP library or use the Yaml PECL extension.

The following example illustrates a basic use of `Zend\Config\Reader\Yaml` that use the Yaml PECL extension. Suppose we have the following *YAML* configuration file:

```
webhost: www.example.com
database:
```

```
    adapter: pdo_mysql
    params:
      host:     db.example.com
      username: dbuser
      password: secret
      dbname:   dbproduction
```

We can use the `Zend\Config\Reader\Yaml` to read this YAML file:

```
$reader = new Zend\Config\Reader\Yaml();
$data   = $reader->fromFile('/path/to/config.yaml');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

If you want to use an external YAML reader you have to pass the callback function in the constructor of the class. For instance, if you want to use the Spyc library:

```
// include the Spyc library
require_once ('path/to/spyc.php');

$reader = new Zend\Config\Reader\Yaml(array('Spyc','YAMLLoadString'));
$data   = $reader->fromFile('/path/to/config.yaml');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

You can also instantiate the `Zend\Config\Reader\Yaml` without any parameter and specify the YAML reader in a second moment using the `setYamlDecoder()` method.

Using `Zend\Config\ReaderYaml` we can include the content of a YAML file in a specific YAML section or element. This is provided using the special syntax `@include`. Suppose we have a YAML file that contains only the database configuration:

```
database:
    adapter: pdo_mysql
    params:
      host:     db.example.com
      username: dbuser
      password: secret
      dbname:   dbproduction
```

We can include this configuration in another YAML file, for instance:

```
webhost:  www.example.com
@include: database.yaml
```

CHAPTER

FIFTYONE

ZEND\CONFIG\WRITER

`Zend\Config\Writer` gives you the ability to write config files out of array, `Zend\Config\Config` and any Traversable object. The `Zend\Config\Writer` is an interface that defines two methods: `toFile()` and `toString()`. We have five specific writers that implement this interface:

- `Zend\Config\Writer\Ini`
- `Zend\Config\Writer\Xml`
- `Zend\Config\Writer\PhpArray`
- `Zend\Config\Writer\Json`
- `Zend\Config\Writer\Yaml`

51.1 Zend\Config\Writer\Ini

The *INI* writer has two modes for rendering with regard to sections. By default the top-level configuration is always written into section names. By calling `$writer->setRenderWithoutSectionsFlags(true);` all options are written into the global namespace of the *INI* file and no sections are applied.

As an addition `Zend\Config\Writer\Ini` has an additional option parameter **nestSeparator**, which defines with which character the single nodes are separated. The default is a single dot, like it is accepted by `Zend\Config\Reader\Ini` by default.

When modifying or creating a `Zend\Config\Config` object, there are some things to know. To create or modify a value, you simply say set the parameter of the `Config` object via the parameter accessor (**->**). To create a section in the root or to create a branch, you just create a new array ("`$config->branch = array();`").

Using Zend\Config\Writer\Ini

This example illustrates the basic use of `Zend\Config\Writer\Ini` to create a new config file:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
```

```
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Ini();
echo $writer->toString($config);
```

The result of this code is an INI string contains the following values:

```
[production]
webhost = "www.example.com"
database.params.host = "localhost"
database.params.username = "production"
database.params.password = "secret"
database.params.dbname = "dbproduction"
```

You can use the method `toFile()` to store the INI data in a file.

51.2 Zend\Config\Writer\Xml

The `Zend\Config\Writer\Xml` can be used to generate an XML string or file starting from a `Zend\Config\Config` object.

Using Zend\Config\Writer\Ini

This example illustrates the basic use of `Zend\Config\Writer\Xml` to create a new config file:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Xml();
echo $writer->toString($config);
```

The result of this code is an XML string contains the following data:

```
<?xml version="1.0" encoding="UTF-8"?>
<zend-config>
    <production>
        <webhost>www.example.com</webhost>
        <database>
            <params>
                <host>localhost</host>
                <username>production</username>
                <password>secret</password>
                <dbname>dbproduction</dbname>
            </params>
        </database>
    </production>
</zend-config>
```

You can use the method `toFile()` to store the XML data in a file.

51.3 Zend\Config\Writer\PhpArray

The `Zend\Config\Writer\PhpArray` can be used to generate a PHP code that returns an array representation of an `Zend\Config\Config` object.

Using Zend\Config\Writer\PhpArray

This example illustrates the basic use of `Zend\Config\Writer\PhpArray` to create a new config file:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\PhpArray();
echo $writer->toString($config);
```

The result of this code is a PHP script that returns an array as follow:

```
<?php
return array (
  'production' =>
  array (
    'webhost' => 'www.example.com',
    'database' =>
    array (
      'params' =>
      array (
        'host' => 'localhost',
        'username' => 'production',
        'password' => 'secret',
        'dbname' => 'dbproduction',
      ),
    ),
  ),
);
```

You can use the method `toFile()` to store the PHP script in a file.

51.4 Zend\Config\Writer\Json

The `Zend\Config\Writer\Json` can be used to generate a PHP code that returns the JSON representation of a `Zend\Config\Config` object.

Using Zend\Config\Writer\Json

This example illustrates the basic use of `Zend\Config\Writer\Json` to create a new config file:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Json();
echo $writer->toString($config);
```

The result of this code is a JSON string contains the following values:

```
{ "webhost"  : "www.example.com",
  "database" : {
    "params"  : {
      "host"     : "localhost",
      "username" : "production",
      "password" : "secret",
      "dbname"   : "dbproduction"
    }
  }
}
```

You can use the method `toFile()` to store the JSON data in a file.

The `Zend\Config\Writer\Json` class uses the `Zend\Json\Json` component to convert the data in a JSON format.

51.5 Zend\Config\Writer\Yaml

The `Zend\Config\Writer\Yaml` can be used to generate a PHP code that returns the YAML representation of a `Zend\Config\Config` object. In order to use the YAML writer we need to pass a callback to an external PHP library or use the Yaml PECL extension.

Using Zend\Config\Writer\Yaml

This example illustrates the basic use of `Zend\Config\Writer\Yaml` to create a new config file using the Yaml PECL extension:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
```

```
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Yaml();
echo $writer->toString($config);
```

The result of this code is a YAML string contains the following values:

```
webhost: www.example.com
database:
    params:
      host:     localhost
      username: production
      password: secret
      dbname:   dbproduction
```

You can use the method `toFile()` to store the YAML data in a file.

If you want to use an external YAML writer library you have to pass the callback function in the constructor of the class. For instance, if you want to use the Spyc library:

```
// include the Spyc library
require_once ('path/to/spyc.php');

$writer = new Zend\Config\Writer\Yaml(array('Spyc','YAMLDump'));
echo $writer->toString($config);
```

CHAPTER

FIFTYTWO

ZEND\CONFIG\PROCESSOR

`Zend\Config\Processor` gives you the ability to perform some operations on a `Zend\Config\Config` object. The `Zend\Config\Processor` is an interface that defines two methods: `process()` and `processValue()`. These operations are provided by the following concrete implementations:

- `Zend\Config\Processor\Constant`: manage PHP constant values;
- `Zend\Config\Processor\Filter`: filter the configuration data using `Zend\Filter`;
- `Zend\Config\Processor\Queue`: manage a queue of operations to apply to configuration data;
- `Zend\Config\Processor\Token`: find and replace specific tokens;
- `Zend\Config\Processor\Translator`: translate configuration values in other languages using `Zend\I18n\Translator`;

Below we reported some examples for each type of processor.

52.1 Zend\Config\Processor\Constant

Using Zend\Config\Processor\Constant

This example illustrates the basic use of `Zend\Config\Processor\Constant`:

```
define ('TEST_CONST', 'bar');
// set true to Zend\Config\Config to allow modifications
$config = new Zend\Config\Config(array('foo' => 'TEST_CONST'), true);
$processor = new Zend\Config\Processor\Constant();

echo $config->foo . ',';
$processor->process($config);
echo $config->foo;
```

This example returns the output: `TEST_CONST, bar.`.

52.2 Zend\Config\Processor\Filter

Using Zend\Config\Processor\Filter

This example illustrates the basic use of `Zend\Config\Processor\Filter`:

```
use Zend\Filter\StringToUpper;
use Zend\Config\Processor\Filter as FilterProcessor;
use Zend\Config\Config;

$config = new Config(array ('foo' => 'bar'), true);
$upper = new StringToUpper();

$upperProcessor = new FilterProcessor($upper);

echo $config->foo . ',';
$upperProcessor->process($config);
echo $config->foo;
```

This example returns the output: `bar,BAR`.

52.3 Zend\Config\Processor\Queue

Using Zend\Config\Processor\Queue

This example illustrates the basic use of `Zend\Config\Processor\Queue`:

```
use Zend\Filter\StringToLower;
use Zend\Filter\StringToUpper;
use Zend\Config\Processor\Filter as FilterProcessor;
use Zend\Config\Processor\Queue;
use Zend\Config\Config;

$config = new Config(array ('foo' => 'bar'), true);
$upper  = new StringToUpper();
$lower  = new StringToLower();

$lowerProcessor = new FilterProcessor($lower);
$upperProcessor = new FilterProcessor($upper);

$queue = new Queue();
$queue->insert($upperProcessor);
$queue->insert($lowerProcessor);
$queue->process($config);

echo $config->foo;
```

This example returns the output: `bar`. The filters in the queue are applied with a *FIFO* logic (First In, First Out).

52.4 Zend\Config\Processor\Token

Using Zend\Config\Processor\Token

This example illustrates the basic use of `Zend\Config\Processor\Token`:

```
// set the Config to true to allow modifications
$config = new Config(array('foo' => 'Value is TOKEN'), true);
$processor = new TokenProcessor();

```

```
$processor->addToken('TOKEN', 'bar');
echo $config->foo . ',';
$processor->process($config);
echo $config->foo;
```

This example returns the output: `Value is TOKEN,Value is bar`.

52.5 Zend\Config\Processor\Translator

Using Zend\Config\Processor\Translator

This example illustrates the basic use of `Zend\Config\Processor\Translator`:

```
use Zend\Config\Config;
use Zend\Config\Processor\Translator as TranslatorProcessor;
use Zend\I18n\Translator\Translator;

$config = new Config(array('animal' => 'dog'), true);

/*
 * The following mapping would exist for the translation
 * loader you provide to the translator instance
 * $italian = array(
 *     'dog' => 'cane'
 * );
 */

$translator = new Translator();
// ... configure the translator ...
$processor = new TranslatorProcessor($translator);

echo "English: {$config->animal}, ";
$processor->process($config);
echo "Italian: {$config->animal}";
```

This example returns the output: `English:  dog, Italian:  cane`.

CHAPTER

FIFTYTHREE

THE FACTORY

The factory gives you the ability to load a configuration file to an array or to `Zend\Config\Config` object. The factory has two purposes

- Loading configuration file(s)
- Storing a configuration file

Note: Storing the configuration will be done to *one* file. The factory is not aware of merging two or more configurations and will not store it into multiple files. If you want to store particular configuration sections to a different file you should separate it manually.

53.1 Loading configuration file

The next example illustrates how to load a single configuration file

```
//Load a php file as array
$config = Zend\Config\Factory::fromFile(__DIR__.'/config/my.config.php');

//Load a xml file as Config object
$config = Zend\Config\Factory::fromFile(__DIR__.'/config/my.config.xml', true);
```

For merging multiple configuration files

53.2 Storing configuration file

Sometimes you want to store the configuration to a file. Also this is really easy to do

CHAPTER

FIFTYFOUR

INTRODUCTION TO ZEND\CONSOLE

Zend Framework 2 features built-in console support.

When a `Zend\Application` is run from a console window (a shell window or Windows command prompt), it will recognize this fact and prepare `Zend\Mvc` components to handle the request. Console support is enabled by default, but to function properly it requires at least one *console route* and *one action controller* to handle the request.

- *Console routing* allows you to invoke controllers and action depending on command line parameters provided by the user.
- *Module Manager integration* allows ZF2 applications and modules to display help and usage information, in case the command line has not been understood (no route matched).
- *Console-aware action controllers* will receive a console request containing all named parameters and flags. They are able to send output back to the console window.
- *Console adapters* provide a level of abstraction for interacting with console on different operating systems.
- *Console prompts* can be used to interact with the user by asking him questions and retrieving input.

54.1 Writing console routes

A console route defines required and optional command line parameters. When a route matches, it behaves analogical to a standard, *http route* and can point to a *MVC controller* and an action.

Let's assume that we'd like our application to handle the following command line:

```
> zf user resetpassword user@mail.com
```

When a user runs our application (`zf`) with these parameters, we'd like to call action `resetpassword` of `Application\Controller\IndexController`.

Note: We will use `zf` to depict the entry point for your application, it can be shell script in application bin folder or simply an alias for `php public/index.php`

First we need to create a **route definition**:

```
user resetpassword <userEmail>
```

This simple route definition expects exactly 3 arguments: a literal "user", literal "resetpassword" followed by a parameter we're calling "userEmail". Let's assume we also accept one optional parameter, that will turn on verbose operation:

```
user resetpassword [--verbose|-v] <userEmail>
```

Now our console route expects the same 3 parameters but will also recognise an optional `--verbose` flag, or its shorthand version: `-v`.

Note: The order of flags is ignored by `Zend\Console`. Flags can appear before positional parameters, after them or anywhere in between. The order of multiple flags is also irrelevant. This applies both to route definitions and the order that flags are used on the command line.

Let's use the definition above and configure our console route. Console routes are automatically loaded from the following location inside config file:

```
array(
    'router' => array(
        'routes' => array(
            // HTTP routes are defined here
        )
    ),

    'console' => array(
        'router' => array(
            'routes' => array(
                // Console routes go here
            )
        )
    ),
)
```

Let's create our console route and point it to `Application\Controller\IndexController::resetpasswordAction()`

```
// we could define routes for Application\Controller\IndexController in Application module config fi
// which is usually located at modules/application/config/module.config.php
array(
    'console' => array(
        'router' => array(
            'routes' => array(
                'user-reset-password' => array(
                    'options' => array(
                        'route'    => 'user resetpassword [--verbose|-v] <userEmail>',
                        'defaults' => array(
                            'controller' => 'Application\Controller\Index',
                            'action'     => 'password'
                        )
                    )
                )
            )
        )
    )
)
```

See Also:

To learn more about console routes and how to use them, please read this chapter: *Console routes and routing*

54.2 Handling console requests

When a user runs our application from command line and arguments match our console route, a `controller` class will be instantiated and an `action` method will be called, just like it is with http requests.

We will now add `resetpassword` action to `Application\Controller\IndexController`:

```
<?php
namespace Application\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Zend\Console\Request as ConsoleRequest;
use Zend\Math\Rand;

class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        return new ViewModel(); // display standard index page
    }

    public function resetpasswordAction(){
        $request = $this->getRequest();

        // Make sure that we are running in a console and the user has not tricked our
        // application into running this action from a public web server.
        if (!$request instanceof ConsoleRequest){
            throw new \RuntimeException('You can only use this action from a console!');
        }

        // Get user email from console and check if the user used --verbose or -v flag
        $userEmail   = $request->getParam('userEmail');
        $verbose     = $request->getParam('verbose');

        // reset new password
        $newPassword = Rand::getString(16);

        //  Fetch the user and change his password, then email him ...
        // [...]

        if (!$verbose){
            return "Done! $userEmail has received an email with his new password.\n";
        }else{
            return "Done! New password for user $userEmail is '$newPassword'. It has also been emaile
        }
    }
}
```

We have created `resetpasswordAction()` than retrieves current request and checks if it's really coming from the console (as a precaution). In this example we do not want our action to be invocable from a web page. Because we have not defined any http route pointing to it, it should never be possible. However in the future, we might define a wildcard route or a 3rd party module might erroneously route some requests to our action - that is why we want to make sure that the request is always coming from a Console environment.

All console arguments supplied by the user are accessible via `$request->getParam()` method. Flags will be represented by a booleans, where `true` means a flag has been used and `false` otherwise.

When our action has finished working it returns a simple `string` that will be shown to the user in console window.

See Also:

There are different ways you can interact with console from a controller. It has been covered in more detail in the following chapter: *Console-aware action controllers*

54.3 Adding console usage info

It is a common practice for console application to display usage information when run for the first time (without any arguments). This is also handled by `Zend\Console` together with `MVC`.

Usage info in ZF2 console applications is provided by *loaded modules*. In case no console route matches console arguments, `Zend\Console` will query all loaded modules and ask for their console usage info.

Let's modify our `Application\Controller\IndexController` to provide usage info:

```
<?php

namespace Application;

use Zend\ModuleManager\Feature\ConfigProviderInterface;
use Zend\ModuleManager\Feature\ConsoleUsageProviderInterface;
use Zend\Console\Adapter\AdapterInterface as Console;

class Module implements
    AutoloaderProviderInterface,
    ConfigProviderInterface,
    ConsoleUsageProviderInterface   // <- our module implement this feature and provides console usag
{
    public function getConfig()
    {
        // [...]
    }

    public function getAutoloaderConfig()
    {
        // [...]
    }

    public function getConsoleUsage(Console $console){
        return array(
            // Describe available commands
            'user resetpassword [--verbose|-v] EMAIL'    => 'Reset password for a user',

            // Describe expected parameters
            array( 'EMAIL',            'Email of the user for a password reset' ),
            array( '--verbose|-v',     '(optional) turn on verbose mode'        ),
        );
    }
}
```

Each module that implements `ConsoleUsageProviderInterface` will be queried for console usage info. On route mismatch, all info from all modules will be concatenated, formatted to console width and shown to the user.

Note: The order of usage info displayed in the console is the order modules load. If you want your application to display important usage info first, change the order your modules are loaded.

See Also:

Modules can also provide an application banner (title). To learn more about the format expected from `getConsoleUsage()` and about application banners, please read this chapter: *Console-aware modules*

CHAPTER

FIFTYFIVE

CONSOLE ROUTES AND ROUTING

Zend Framework 2 has *native MVC integration with console*, which means that command line arguments are read and used to determine the appropriate *action controller* and action method that will handle the request. Actions can perform any number of task prior to returning a result, that will be displayed to the user in his console window.

There are several routes you can use with Console. All of them are defined in `Zend\Mvc\Router\Console\*` classes.

See Also:

Routes are used to handle real commands, but they are not used to create help messages (usage information). When a zf2 application is run in console for the first time (without arguments) it can *display usage information* that is provided by modules. To learn more about providing usage information, please read this chapter: *Console-aware modules*.

55.1 Router configuration

All Console Routes are automatically read from the following configuration location:

```
// This can sit inside of modules/Application/config/module.config.php or any other module's config.
array(
    'router' => array(
        'routes' => array(
            // HTTP routes are here
        )
    ),

    'console' => array(
        'router' => array(
            'routes' => array(
                // Console routes go here
            )
        )
    ),
)
```

Console Routes will only be processed when the application is run inside console (terminal) window. They have no effect in web (http) request and will be ignored. It is possible to define only HTTP routes (only web application) or only Console routes (which means we want a console-only application which will refuse to run in a browser).

A single route can be described with the following array:

```
// inside config.console.router.routes:
// [...]
'my-first-route' => array(
```

```
    'type'    => 'simple'       // <- simple route is created by default, we can skip that
    'options' => array(
        'route'    => 'foo bar',
        'defaults' => array(
            'controller' => 'Application\Controller\Index',
            'action'     => 'password'
        )
    )
)
```

We have created a `simple` console route with a name `my-first-route`. It expects two parameters: `foo` and `bar`. If user puts these in a console, `Application\Controller\IndexController::passwordAction()` action will be invoked.

See Also:

You can read more about how *ZF2 routing works in this chapter*.

55.2 Basic route

This is the default route type for console. It recognizes the following types of parameters:

- *Literal parameters* (i.e. `create object (external|internal))`
- *Literal flags* (i.e. `--verbose --direct [-d] [-a])`
- *Positional value parameters* (i.e. `create <modelName> [<destination>])`
- *Value flags* (i.e. `--name=NAME [--method=METHOD])`

55.2.1 Literal parameters

These parameters are expected to appear on the command line exactly the way they are spelled in the route. For example:

```
'show-users' => array(
    'options' => array(
        'route'    => 'show users',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'show'
        )
    )
)
```

This route will **only** match for the following command line

```
> zf show users
```

It expects **mandatory literal parameters** `show users`. It will not match if there are any more params, or if one of the words is missing. The order of words is also enforced.

We can also provide **optional literal parameters**, for example:

```
'show-users' => array(
    'options' => array(
        'route'    => 'show [all] users',
        'defaults' => array(
```

```
            'controller' => 'Application\Controller\Users',
            'action'     => 'show'
        )
    )
)
```

Now this route will match for both of these commands:

```
> zf show users
> zf show all users
```

We can also provide **parameter alternative**:

```
'show-users' => array(
    'options' => array(
        'route'    => 'show [all|deleted|locked|admin] users',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'show'
        )
    )
)
```

This route will match both without and with second parameter being one of the words, which enables us to capture commands such:

```
> zf show users
> zf show locked users
> zf show admin users
etc.
```

Note: Whitespaces in route definition are ignored. If you separate your parameters with more spaces, or separate alternatives and pipe characters with spaces, it won't matter for the parser. The above route definition is equivalent to: `show [ all | deleted | locked | admin ] users`

55.2.2 Literal flags

Flags are a common concept for console tools. You can define any number of optional and mandatory flags. The order of flags is ignored. The can be defined in any order and the user can provide them in any other order.

Let's create a route with **optional long flags**

```
'check-users' => array(
    'options' => array(
        'route'    => 'check users [--verbose] [--fast] [--thorough]',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'check'
        )
    )
)
```

This route will match for commands like:

```
> zf check users
> zf check users --fast
```

```
> zf check users --verbose --thorough
> zf check users --thorough --fast
```

We can also define one or more **mandatory long flags** and group them as an alternative:

```
'check-users' => array(
    'options' => array(
        'route'    => 'check users (--suspicious|--expired) [--verbose] [--fast] [--thorough]',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'check'
        )
    )
)
```

This route will **only match** if we provide either `--suspicious` or `--expired` flag, i.e.:

```
> zf check users --expired
> zf check users --expired --fast
> zf check users --verbose --thorough --suspicious
```

We can also use **short flags** in our routes and group them with long flags for convenience, for example:

```
'check-users' => array(
    'options' => array(
        'route'    => 'check users [--verbose|-v] [--fast|-f] [--thorough|-t]',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'check'
        )
    )
)
```

Now we can use short versions of our flags:

```
> zf check users -f
> zf check users -v --thorough
> zf check users -t -f -v
```

55.2.3 Positional value parameters

Value parameters capture any text-based input and come in two forms - positional and flags.

Positional value parameters are expected to appear in an exact position on the command line. Let's take a look at the following route definition:

```
'delete-user' => array(
    'options' => array(
        'route'    => 'delete user <userEmail>',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'delete'
        )
    )
)
```

This route will match for commands like:

```
> zf delete user john@acme.org
> zf delete user betty@acme.org
```

We can access the email value by calling `$this->getRequest()->getParam('userEmail')` inside of our controller action (you can *read more about accessing values here*)

We can also define **optional positional value parameters** by adding square brackets:

```
'delete-user' => array(
    'options' => array(
        'route'    => 'delete user [<userEmail>]',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'delete'
        )
    )
)
```

In this case, `userEmail` parameter will not be required for the route to match. If it is not provided, `userEmail` parameter will not be set.

We can define any number of positional value parameters, for example:

```
'create-user' => array(
    'options' => array(
        'route'    => 'create user <firstName> <lastName> <email> <position>',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'create'
        )
    )
)
```

This allows us to capture commands such as:

```
> zf create user Johnny Bravo john@acme.org Entertainer
```

Note: Command line arguments on all systems must be properly escaped, otherwise they will not be passed to our application correctly. For example, to create a user with two names and a complex position description, we could write something like this:

```
> zf create user "Johnan Tom" Bravo john@acme.org "Head of the Entertainment Department"
```

55.2.4 Value flag parameters

Positional value parameters are only matched if they appear in the exact order as described in the route. If we do not want to enforce the order of parameters, we can define **value flags**.

Value flags can be defined and matched in any order. They can digest text-based values, for example:

```
'find-user' => array(
    'options' => array(
        'route'    => 'find user [--id=] [--firstName=] [--lastName=] [--email=] [--position=] ',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'find'
        )
```

```
    )
)
```

This route will match for any of the following routes:

```
> zf find user
> zf find user --id 29110
> zf find user --id=29110
> zf find user --firstName=Johny --lastName=Bravo
> zf find user --lastName Bravo --firstName Johny
> zf find user --position=Executive --firstName=Bob
> zf find user --position "Head of the Entertainment Department"
```

Note: The order of flags is irrelevant for the parser.

Note: The parser understands values that are provided after equal symbol (=) and separated by a space. Values without whitespaces can be provided after = symbol or after a space. Values with one more whitespaces however, must be properly quoted and written after a space.

In previous example, all value flags are optional. It is also possible to define **mandatory value flags**:

```
'rename-user' => array(
    'options' => array(
        'route'    => 'rename user --id= [--firstName=] [--lastName=]',
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'rename'
        )
    )
)
```

The `--id` parameter **is required** for this route to match. The following commands will work with this route:

```
> zf rename user --id 123
> zf rename user --id 123 --firstName Jonathan
> zf rename user --id=123 --lastName=Bravo
```

55.3 Catchall route

This special route will catch all console requests, regardless of the parameters provided.

```
'default-route' => array(
    'options' => array(
        'type'     => 'catchall',
        'defaults' => array(
            'controller' => 'Application\Controller\Index',
            'action'     => 'consoledefault'
        )
    )
)
```

Note: This route type is rarely used. You could use it as a last console route, to display usage information. Before you do so, read about the *preferred way of displaying console usage information*. It is the recommended way and will guarantee proper inter-operation with other modules in your application.

55.4 Console routes cheat-sheet

Param type	Example route definition	Explanation
Literal params		
Literal	`foo bar`	"foo" followed by "bar"
Literal alternative	`foo (bar|baz)`	"foo" followed by "bar" or "baz"
Literal, optional	`foo [bar]`	"foo", optional "bar"
Literal, optional alternative	`foo [bar|baz]`	"foo", optional "bar" or "baz"
Flags		
Flag long	`foo --bar`	"foo" as first parameter, "–bar" flag before or after
Flag long, optional	`foo [--bar]`	"foo" as first parameter, optional "–bar" flag before or after
Flag long, optional, alternative	`foo [--bar|--baz]`	"foo" as first parameter, optional "–bar" or "–baz", before or after
Flag short	`foo -b`	"foo" as first parameter, "-b" flag before or after
Flag short, optional	`foo [-b]`	"foo" as first parameter, optional "-b" flag before or after
Flag short, optional, alternative	`foo [-b|-z]`	"foo" as first parameter, optional "-b" or "-z", before or after
Flag long/short alternative	`foo [--bar|-b]`	"foo" as first parameter, optional "–bar" or "-b" before or after
Value parameters		
Value positional param	`foo <bar>`	"foo" followed by any text (stored as "bar" param)
Value positional param, optional	`foo [<bar>]`	"foo", optionally followed by any text (stored as "bar" param)
Value Flag	`foo --bar=`	"foo" as first parameter, "–bar" with a value, before or after
Value Flag, optional	`foo [--bar=]`	"foo" as first parameter, optionally "–bar" with a value, before or after
Parameter groups		
Literal params group	`foo (bar|baz):myParam`	"foo" followed by "bar" or "baz" (stored as "myParam" param)
Literal optional params group	`foo [bar|baz]:myParam`	"foo" followed by optional "bar" or "baz" (stored as "myParam" param)
Long flags group	`foo (--bar|--baz):myParam`	"foo", "bar" or "baz" flag before or after (stored as "myParam" param)
Long optional flags group	`foo [--bar|--baz]:myParam`	"foo", optional "bar" or "baz" flag before or after (as "myParam" param)
Short flags group	`foo (-b|-z):myParam`	"foo", "-b" or "-z" flag before or after (stored as "myParam" param)
Short optional flags group	`foo [-b|-z]:myParam`	"foo", optional "-b" or "-z" flag before or after (stored as "myParam" param)

CHAPTER

FIFTYSIX

CONSOLE-AWARE MODULES

Zend Framework 2 has *native MVC integration with console*. The integration also works with *modules loaded with Module Manager*.

ZF2 ships with `RouteNotFoundStrategy` which is responsible of displaying usage information inside Console, in case the user has not provided any arguments, or arguments could not be understood. The strategy currently supports two types of information: *application banners* and *usage information*.

56.1 Application banner

To run the console ZF 2 component, go to your *public* folder, and type *php index.php*. By default, it will simply output the current ZF 2 version, like this:

Our Application module (and any other module) can provide **application banner**. In order to do so, our Module class has to implement `Zend\ModuleManager\Feature\ConsoleBannerProviderInterface`. Let's do this now.

```
// modules/Application/Module.php
<?php
namespace Application;

use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
use Zend\Console\Adapter\AdapterInterface as Console;

class Module implements ConsoleBannerProviderInterface
{
    /**
     * This method is defined in ConsoleBannerProviderInterface
     */
    public function getConsoleBanner(Console $console)
    {
        return 'MyModule 0.0.1';
    }
}
```

As you can see, the application banner should be a single line string that returns the module's name and (if available) its current version.

If several modules define their own banner, they are all shown one after the other (they will be joined together in the order modules are loaded). This way, it makes it very easy to spot which modules provide console commands.

After running our application, we'll see our newly created banner.

Let's create and load second module that provides a banner.

```
<?php
// config/application.config.php
return array(
    'modules' => array(
        'Application',
        'User',     // < load user module in modules/User
    ),
```

User module will add-on a short info about itself:

```
// modules/User/Module.php
<?php
namespace User;

use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
use Zend\Console\Adapter\AdapterInterface as Console;

class Module implements ConsoleBannerProviderInterface
{
    /**
     * This method is defined in ConsoleBannerProviderInterface
     */
    public function getConsoleBanner(Console $console){
        return "User Module 0.0.1";
    }
}
```

Because `User` module is loaded after `Application` module, the result will look like this:

Note: Application banner is displayed as-is - no trimming or other adjustments will be performed on the text. As you can see, banners are also automatically colorized as blue.

56.2 Usage information

In order to display usage information, our Module class has to implement `Zend\ModuleManager\Feature\ConsoleUsageProviderInterface`. Let's modify our example and add new method:

```
// modules/Application/Module.php
<?php
namespace Application;

use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
use Zend\ModuleManager\Feature\ConsoleUsageProviderInterface;
use Zend\Console\Adapter\AdapterInterface as Console;

class Module implements ConsoleBannerProviderInterface, ConsoleUsageProviderInterface
{
    public function getConsoleBanner(Console $console){ // ... }

```

```
    /**
     * This method is defined in ConsoleUsageProviderInterface
     */
    public function getConsoleUsage(Console $console)
    {
        return array(
            'show stats'             => 'Show application statistics',
            'run cron'               => 'Run automated jobs',
            '(enable|disable) debug' => 'Enable or disable debug mode for the application.'
        );
    }
}
```

This will display the following information:

Similar to *application banner* multiple modules can provide usage information, which will be joined together and displayed to the user. The order in which usage information is displayed is the order in which modules are loaded.

As you can see, Console component also prepended each module's usage by the module's name. This helps to visually separate each modules (this can be useful when you have multiple modules that provide commands). By default, the component colorizes those in red.

Note: Usage info provided in modules **does not connect** with *console routing*. You can describe console usage in any form you prefer and it does not affect how MVC handles console commands. In order to handle real console requests you need to define 1 or more *console routes*.

56.2.1 Free-form text

In order to output free-form text as usage information, `getConsoleUsage()` can return a string, or an array of strings, for example:

```
public function getConsoleUsage(Console $console)
{
    return 'User module expects exactly one argument - user name. It will display information for thi
}
```

Note: The text provided is displayed as-is - no trimming or other adjustments will be performed. If you'd like to fit your usage information inside console window, you could check its width with `$console->getWidth()`.

56.2.2 List of commands

If `getConsoleUsage()` returns and associative array, it will be automatically aligned in 2 columns. The first column will be prepended with script name (the entry point for the application). This is useful to display different ways of running the application.

```
public function getConsoleUsage(Console $console)
{
     return array(
        'delete user <userEmail>'        => 'Delete user with email <userEmail>',
        'disable user <userEmail>'       => 'Disable user with email <userEmail>',
```

```
        'list [all|disabled] users'      => 'Show a list of users',
        'find user [--email=] [--name=]' => 'Attempt to find a user by email or name',
    );
}
```

Note: Commands and their descriptions will be aligned in two columns, that fit inside Console window. If the window is resized, some texts might be wrapped but all content will be aligned accordingly. If you don't like this behavior, you can always return *free-form text* that will not be transformed in any way.

56.2.3 List of params and flags

Returning an array of arrays from `getConsoleUsage()` will produce a listing of parameters. This is useful for explaining flags, switches, possible values and other information. The output will be aligned in multiple columns for readability.

Below is an example:

```
public function getConsoleUsage(Console $console){
    return array(
        array( '<userEmail>'   , 'email of the user' ),
        array( '--verbose'     , 'Turn on verbose mode' ),
        array( '--quick'       , 'Perform a "quick" operation' ),
        array( '-v'            , 'Same as --verbose' ),
        array( '-w'            , 'Wide output')
    );
}
```

Using this method, we can display more than 2 columns of information, for example:

```
public function getConsoleUsage(Console $console){
    return array(
        array( '<userEmail>' , 'user email'        , 'Full email address of the user to find.' ),
        array( '--verbose'   , 'verbose mode'      , 'Display additional information during processin
        array( '--quick'     , '"quick" operation' , 'Do not check integrity, just make changes and i
        array( '-v'          , 'Same as --verbose' , 'Display additional information during processin
        array( '-w'          , 'wide output'       , 'When listing users, use the whole available scr
    );
}
```

Note: All info will be aligned in one or more columns that fit inside Console window. If the window is resized, some texts might be wrapped but all content will be aligned accordingly. In case the number of columns changes (i.e. the array() contains different number of elements) a new table will be started, with new alignment and different column widths.

If you don't like this behavior, you can always return *free-form text* that will not be transformed in any way.

56.2.4 Mixing styles

You can use mix together all of the above styles to provide comprehensive usage information, for example:

```
public function getConsoleUsage(Console $console){
    return array(
        'Finding and listing users',
        'list [all|disabled] users [-w]'    => 'Show a list of users',
        'find user [--email=] [--name=]'    => 'Attempt to find a user by email or name',

        array('[all|disabled]',    'Display all users or only disabled accounts'),
        array('--email=EMAIL',     'Email of the user to find'),
        array('--name=NAME',       'Full name of the user to find.'),
        array('-w',                'Wide output - When listing users use the whole available screen w

        'Manipulation of user database:',
        'delete user <userEmail> [--verbose|-v] [--quick]'  => 'Delete user with email <userEmail>',
        'disable user <userEmail> [--verbose|-v]'           => 'Disable user with email <userEmail>',

        array( '<userEmail>' , 'user email'        , 'Full email address of the user to change.' ),
        array( '--verbose'   , 'verbose mode'      , 'Display additional information during processin
        array( '--quick'     , '"quick" operation' , 'Do not check integrity, just make changes and i
        array( '-v'          , 'Same as --verbose' , 'Display additional information during processin

    );
}
```

56.3 Best practices

As a reminder, here are the best practices when providing usage for your commands:

1. Your `getConsoleBanner` should only return a one-line string containing the module's name and its version (if available).
2. Your `getConsoleUsage` should not return module's name; it is prepended automatically for you by Console component.

CHAPTER

FIFTYSEVEN

CONSOLE-AWARE ACTION CONTROLLERS

Zend Framework 2 has built-in *MVC integration with the console*. When the user runs an application in a console window, the request will be routed. By matching command line arguments against *console routes we have defined in our application*, the MVC will invoke a controller and an action.

In this chapter we will learn how ZF2 Controllers can interact with and return output to console window.

See Also:

In order for a controller to be invoked, at least one route must point to it. To learn about creating console routes, please read the chapter *Console routes and routing*

57.1 Handling console requests

Console requests are very similar to HTTP requests. In fact, they implement a common interface and are created at the same time in the MVC workflow. *Console routes* match against command line arguments and provide a `defaults` array, which holds the `controller` and `action` keys. These correspond with controller aliases in the ServiceManager, and method names in the controller class. This is analogous to the way HTTP requests are handled in ZF2.

See Also:

To learn about defining and creating controllers, please read the chapter *Routing and controllers*

In this example we'll use the following simple route:

```
// FILE: modules/Application/config/module.config.php
array(
    'router' => array(
        'routes' => array(
            // HTTP routes are here
        )
    ),

    'console' => array(
        'router' => array(
            'routes' => array(
                'list-users' => array(
                    'options' => array(
                        'route'    => 'show [all|disabled|deleted]:mode users [--verbose|-v]',
                        'defaults' => array(
```

```
                                'controller' => 'Application\Controller\Index',
                                'action'     => 'show-users'
                            )
                        )
                    )
                )
            )
        ),
)
```

This route will match commands such as:

```
> php public/index.php show users
> php public/index.php show all users
> php public/index.php show disabled users
```

This route points to the method `Application\Controller\IndexController::showUsersAction()`.

Let's add it to our controller.

```php
<?php
namespace Application\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        return new ViewModel(); // display standard index page
    }

    public function showUsersAction()
    {
        $request = $this->getRequest();

        // Check verbose flag
        $verbose = $request->getParam('verbose') || $request->getParam('v');

        // Check mode
        $mode = $request->getParam('mode', 'all'); // defaults to 'all'

        $users = array();
        switch ($mode) {
            case 'disabled':
                $users = $this->getServiceLocator()->get('users')->fetchDisabledUsers();
                break;
            case 'deleted':
                $users = $this->getServiceLocator()->get('users')->fetchDeletedUsers();
                break;
            case 'all':
            default:
                $users = $this->getServiceLocator()->get('users')->fetchAllUsers();
                break;
        }
    }
}
```

We fetch the console request, read parameters, and load users from our (theoretical) users service. In order to make

this method functional, we'll have to display the result in the console window.

57.2 Sending output to console

The simplest way for our controller to display data in the console window is to `return` a string. Let's modify our example to output a list of users:

```
public function showUsersAction()
{
    $request = $this->getRequest();

    // Check verbose flag
    $verbose = $request->getParam('verbose') || $request->getParam('v');

    // Check mode
    $mode = $request->getParam('mode', 'all'); // defaults to 'all'

    $users = array();
    switch ($mode) {
        case 'disabled':
            $users = $this->getServiceLocator()->get('users')->fetchDisabledUsers();
            break;
        case 'deleted':
            $users = $this->getServiceLocator()->get('users')->fetchDeletedUsers();
            break;
        case 'all':
        default:
            $users = $this->getServiceLocator()->get('users')->fetchAllUsers();
            break;
    }

    if (count($users) == 0) {
        // Show an error message in the console
        return "There are no users in the database\n";
    }

    $result = '';

    foreach ($users as $user) {
        $result .= $user->name . ' ' . $user->email . "\n";
    }

    return $result; // show it in the console
}
```

On line 27, we are checking if the users service found any users - otherwise we are returning an error message that will be immediately displayed and the application will end.

If there are 1 or more users, we will loop through them with and prepare a listing. It is then returned from the action and displayed in the console window.

57.3 Are we in a console?

Sometimes we might need to check if our method is being called from a console or from a web request. This is useful to block certain methods from running in the console or to change their behavior based on that context.

Here is an example of how to check if we are dealing with a console request:

```
namespace Application\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Zend\Console\Request as ConsoleRequest;
use RuntimeException;

class IndexController extends AbstractActionController
{
    public function showUsersAction()
    {
        $request = $this->getRequest();

        // Make sure that we are running in a console and the user has not tricked our
        // application into running this action from a public web server.
        if (!$request instanceof ConsoleRequest) {
            throw new RuntimeException('You can only use this action from a console!');
        }
        // ...
    }
}
```

Note: You do not need to secure all your controllers and methods from console requests. Controller actions will **only be invoked** when at least one *console route* matches it. HTTP and Console routes are separated and defined in different places in module (and application) configuration.

There is no way to invoke a console action unless there is at least one route pointing to it. Similarly, there is no way for an HTTP action to be invoked unless there is at least one HTTP route that points to it.

The example below shows how a single controller method can handle **both Console and HTTP requests**:

```
namespace Application\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Zend\Console\Request as ConsoleRequest;
use Zend\Http\Request as HttpRequest;
use RuntimeException;

class IndexController extends AbstractActionController
{
    public function showUsersAction()
    {
        $request = $this->getRequest();

        $users = array();
        // ... fetch users from database ...

        if ($request instanceof HttpRequest) {
            // display a web page with users list
            return new ViewModel($result);
        } elseif ($request instanceof ConsoleRequest) {
            // ... prepare console output and return it ...
            return $result;
        } else {
            throw new RuntimeException('Cannot handle request of type ' . get_class($request));
```

```
        }
    }
}
```

57.4 Reading values from console parameters

There are several types of parameters recognized by the Console component - all of them are described in *the console routing chapter*. Here, we'll focus on how to retrieve values from distinct parameters and flags.

57.4.1 Positional parameters

After a route matches, we can access both **literal parameters** and **value parameters** from within the `$request` container.

Assuming we have the following route:

```
// inside of config.console.router.routes:
'show-users' => array(
    'options' => array(
        'route'    => 'show (all|deleted|locked|admin) [<groupName>]'
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'showusers'
        )
    )
)
```

If this route matches, our action can now query parameters in the following way:

```
// an action inside Application\Controller\UsersController:
public function showUsersAction()
{
    $request = $this->getRequest();

    // We can access named value parameters directly by their name:
    $showUsersFromGroup = $request->getParam('groupName');

    // Literal parameters can be checked with isset() against their exact spelling
    if (isset($request->getParam('all'))) {
        // show all users
    } elseif (isset($request->getParam('deleted'))) {
        // show deleted users
    }
    // ...
}
```

In case of parameter alternatives, it is a good idea to **assign a name to the group**, which simplifies the branching in our action controllers. We can do this with the following syntax:

```
// inside of config.console.router.routes:
'show-users' => array(
    'options' => array(
        'route'    => 'show (all|deleted|locked|admin):userTypeFilter [<groupName>]'
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
```

```
            'action'     => 'showusers'
        )
    )
)
```

Now we can use a the group name `userTypeFilter` to check which option has been selected by the user:

```
public function showUsersAction()
{
    $request = $this->getRequest();

    // We can access named value parameters directly by their name:
    $showUsersFromGroup = $request->getParam('groupName');

    // The selected option from second parameter is now stored under 'userTypeFilter'
    $userTypeFilter     = $request->getParam('userTypeFilter');

    switch ($userTypeFilter) {
        case 'all':
            // all users
        case 'deleted':
            // deleted users
        case 'locked'
           // ...
           // ...
    }
}
```

57.4.2 Flags

Flags are directly accessible by name. Value-capturing flags will contain string values, as provided by the user. Non-value flags will be equal to `true`.

Given the following route:

```
'find-user' => array(
    'options' => array(
        'route'    => 'find user [--fast] [--verbose] [--id=] [--firstName=] [--lastName=] [--email=
        'defaults' => array(
            'controller' => 'Application\Controller\Users',
            'action'     => 'find',
        )
    )
)
```

We can easily retrieve values in the following fashion:

```
public function findAction()
{
    $request = $this->getRequest();

    // We can retrieve values from value flags using their name
    $searchId        = $request->getParam('id',        null); // default null
    $searchFirstName = $request->getParam('firstName', null);
    $searchLastName  = $request->getParam('lastName',  null);
    $searchEmail     = $request->getParam('email',     null);

    // Standard flags that have been matched will be equal to TRUE
```

```
    $isFast          = (bool) $request->getParam('fast',   false); // default false
    $isVerbose       = (bool) $request->getParam('verbose',false);

    if ($isFast) {
        // perform a fast query ...
    } else {
        // perform standard query ...
    }
}
```

In case of **flag alternatives**, we have to check each alternative separately:

```
// Assuming our route now reads:
//      'route'    => 'find user [--fast|-f] [--verbose|-v] ... ',
//
public function findAction()
{
    $request = $this->getRequest();

    // Check both alternatives
    $isFast    = $request->getParam('fast',false)    || $request->getParam('f',false);
    $isVerbose = $request->getParam('verbose',false) || $request->getParam('v',false);

    // ...
}
```

CHAPTER

FIFTYEIGHT

CONSOLE ADAPTERS

Zend Framework 2 provides console abstraction layer, which works around various bugs and limitations in operating systems. It handles displaying of colored text, retrieving console window size, charset and provides basic line drawing capabilities.

See Also:

Console Adapters can be used for a low-level access to the console. If you plan on building functional console applications you do not normally need to use adapters. Make sure to *read about console MVC integration first*, because it provides a convenient way for running modular console applications without directly writing to or reading from console window.

58.1 Retrieving console adapter

If you are using *MVC controllers* you can obtain Console adapter instance using Service Manager.

```
namespace Application;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\Console\Adapter\AdapterInterface as Console;
use Zend\Console\Exception\RuntimeException;

class ConsoleController extends AbstractActionController
{
    public function testAction()
    {
        $console = $this->getServiceLocator()->get('console');
        if (!$console instanceof Console) {
            throw new RuntimeException('Cannot obtain console adapter. Are we running in a console?')
        }
    }
}
```

If you are using `Zend\Console` without MVC, we can get adapter using the following code:

```
use Zend\Console\Console;
use Zend\Console\Exception\RuntimeException as ConsoleException;

try {
    $console = Console::getInstance();
} catch (ConsoleException $e){
    // Could not get console adapter - most likely we are not running inside a console window.
}
```

Note: For practical and security reasons, `Console::getInstance()` will always throw an exception if you attempt to get console instance in a non-console environment (i.e. when running on a HTTP server). You can override this behavior by manually instantiating one of `Zend\Console\Adapter\*` classes.

58.2 Using console adapter

58.2.1 Window size and title

$console->getWidth() (*int*) Get real console window width in characters.

$console->getHeight() (*int*) Get real console window height in characters.

$console->getSize() (*array*) Get an `array( $width, $height)` with current console window dimensions.

$console->getTitle() (*string*) Get console window title.

Note: For UTF-8 enabled consoles (terminals) dimensions represent the number of multibyte characters (real characters).

Note: On consoles with virtual buffers (i.e. MS Windows Command Prompt) width and height represent visible (real) size, without scrolling the window. For example - if the window scrolling width is 120 chars, but it's real, visible width is 80 chars, `getWidth()` will return `80`.

58.2.2 Character set

$console->isUtf8() (*boolean*) Is the console UTF-8 compatible (can display unicode strings) ?

$console->getCharset() (*Zend\Console\Charset\CharsetInterface*) This method will return one of `Console\Charset\*` classes that represent the readable charset that can be used for line-drawing. It is automatically detected by the adapter.

58.2.3 Writing to console

$console->write(string $text, $color = null, $bgColor = null) Write a `$text` to the console, optionally using foreground `$color` and background `$bgColor`. Color value is one of the constants in `Zend\Console\ColorInterface`.

$console->writeLine(string $text, $color = null, $bgColor = null) Write a single line of `$text` to the console. This method will output a newline character at the end of text moving console cursor to next line.

$console->writeAt(string $text, int $x, int $y, $color = null, $bgColor = null) Write `$text` at the specified `$x` and `$y` coordinates of console window. Top left corner of the screen has coordinates of `$x = 1; $x = 1`. To retrieve far-right and bottom coordinates, use `getWidth()` and `getHeight()` methods.

58.2.4 Reading from console

$console->readChar(string $mask = null) (*string*) Read a single character from console. Optional `(string) $mask` can be provided to force entering only a selected set of characters. For example, to read a single digit, we can use the following syntax: `$digit = $console->readChar('0123456789');`

$console->readLine(int $maxLength = 2048) (*string*) Read a single line of input from console. Optional `(int) $maxLength` can be used to limit the length of data that will be read. The line will be returned **without ending newline character**.

58.2.5 Miscellaneous

$console->hideCursor() Hide blinking cursor from console.

$console->showCursor() Show blinking cursor in console.

$console->clear() Clear the screen.

$console->clearLine() Clear the line that the cursor currently sits at.

CHAPTER

FIFTYNINE

CONSOLE PROMPTS

In addition to *console abstraction layer* Zend Framework 2 provides numerous convenience classes for interacting with the user in console environment. This chapter describes available `Zend\Console\Prompt` classes and their example usage.

All prompts can be instantiated as objects and provide `show()` method.

```
use Zend\Console\Prompt;

$confirm = new Prompt\Confirm('Are you sure you want to continue?');
$result = $confirm->show();
if ($result) {
    // the user chose to continue
}
```

There is also a shorter method of displaying prompts, using static `prompt()` method:

```
use Zend\Console\Prompt;

$result = Prompt\Confirm::prompt('Are you sure you want to continue?');
if ($result) {
    // the user chose to continue
}
```

Both of above examples will display something like this:

See Also:

Make sure to *read about console MVC integration first*, because it provides a convenient way for running modular console applications without directly writing to or reading from console window.

59.1 Confirm

This prompt is best used for a **yes / no** type of choices.

`Confirm( string $text, string $yesChar = 'y', string $noChar = 'n' )`

$text (*string*) The text to show with the prompt

$yesChar (*string*) The char that corresponds with YES choice. Defaults to `y`.

$noChar (*string*) The char that corresponds with NO choice. Defaults to `n`.

Example usage:

```
use Zend\Console\Prompt\Confirm;

if ( Confirm::prompt('Is this the correct answer? [y/n]', 'y', 'n') ) {
    $console->write("You chose YES");
} else {
    $console->write("You chose NO");
}
```

59.2 Line

This prompt asks for a line of text input.

```
Line(
    string $text = 'Please enter value',
    bool $allowEmpty = false,
    bool $maxLength = 2048
)
```

$text (*string*) The text to show with the prompt

$allowEmpty (*boolean*) Can this prompt be skipped, by pressing [ENTER] ? (default fo false)

$maxLength (*integer*) Maximum length of the input. Anything above this limit will be truncated.

Example usage:

```
use Zend\Console\Prompt\Line;

$name = Line::prompt(
    'What is your name?',
    false,
    100
);

$console->write("Good day to you $name!");
```

59.3 Char

This prompt reads a single keystroke and optionally validates it against a list o allowed characters.

```
Char(
    string $text = 'Please hit a key',
    string $allowedChars = 'abc',
    bool   $ignoreCase = true,
    bool   $allowEmpty = false,
    bool   $echo = true
)
```

$text (*string*) The text to show with the prompt

$allowedChars (*string*) A list of allowed keys that can be pressed.

$ignoreCase (*boolean*) Ignore the case of chars pressed (default to true)

$allowEmpty (*boolean*) Can this prompt be skipped, by pressing [ENTER] ? (default fo false)

$echo (*boolean*) Should the selection be displayed on the screen ?

Example usage:

```
use Zend\Console\Prompt\Char;

$answer = Char::prompt(
    'What is the correct answer? [a,b,c,d,e]',
    'abcde',
    true,
    false,
    true
);

if ($answer == 'b') {
    $console->write('Correct. This it the right answer');
} else {
    $console->write('Wrong ! Try again.');
}
```

59.4 Select

This prompt displays a number of choices and asks the user to pick one.

```
Select(
    string $text = 'Please select one option',
    array  $options = array(),
    bool   $allowEmpty = false,
    bool   $echo = false
)
```

$text (*string*) The text to show with the prompt

$options (*array*) An associative array with keys strokes (chars) and their displayed values.

$allowEmpty (*boolean*) Can this prompt be skipped, by pressing [ENTER] ? (default fo false)

$echo (*boolean*) Should the selection be displayed on the screen ?

Example usage:

```
$options = array(
    'a' => 'Apples',
    'o' => 'Oranges',
    'p' => 'Pears',
    'b' => 'Bananas',
    'n' => 'none of the above...'
);

$answer = Select::prompt(
    'Which fruit do you like the best?',
    $options,
    false,
    false
```

```
);

$console->write("You told me that you like " . $options[$answer]);
```

See Also:

To learn more about accessing console, writing to and reading from it, make sure to read the following chapter: *Console adapters*.

CHAPTER

SIXTY

INTRODUCTION TO ZEND\CRYPT

Zend\Crypt provides support of some cryptographic tools. The available features are:

- encrypt-then-authenticate using symmetric ciphers (the authentication step is provided using HMAC);
- encrypt/decrypt using symmetric and public key algorithm (e.g. RSA algorithm);
- generate digital sign using public key algorithm (e.g. RSA algorithm);
- key exchange using the Diffie-Hellman method;
- Key derivation function (e.g. using PBKDF2 algorithm);
- Secure password hash (e.g. using Bcrypt algorithm);
- generate Hash values;
- generate HMAC values;

The main scope of this component is to offer an easy and secure way to protect and authenticate sensitive data in PHP. Because the use of cryptography is not so easy we recommend to use the `Zend\Crypt` component only if you have a minimum background on this topic. For an introduction to cryptography we suggest the following references:

- Dan Boneh "Cryptography course" Stanford University, Coursera - free online course
- N.Ferguson, B.Schneier, and T.Kohno, "Cryptography Engineering", John Wiley & Sons (2010)
- B.Schneier "Applied Cryptography", John Wiley & Sons (1996)

Note: PHP-CryptLib

Most of the ideas behind the `Zend\Crypt` component have been inspired by the PHP-CryptLib project of Anthony Ferrara. PHP-CryptLib is an all-inclusive pure PHP cryptographic library for all cryptographic needs. It is meant to be easy to install and use, yet extensible and powerful enough for even the most experienced developer.

CHAPTER

SIXTYONE

ENCRYPT/DECRYPT USING BLOCK CIPHERS

`Zend\Crypt\BlockCipher` implements the encrypt-then-authenticate mode using HMAC to provide authentication.

The symmetric cipher can be choose with a specific adapter that implements the `Zend\Crypt\Symmetric\SymmetricInterface`. We support the standard algorithms of the Mcrypt extension. The adapter that implements the Mcrypt is `Zend\Crypt\Symmetric\Mcrypt`.

In the following code we reported an example on how to use the BlockCipher class to encrypt-then-authenticate a string using the AES block cipher (with a key of 256 bit) and the HMAC algorithm (using the SHA-256 hash function).

```
use Zend\Crypt\BlockCipher;

$blockCipher = BlockCipher::factory('mcrypt', array('algo' => 'aes'));
$blockCipher->setKey('encryption key');
$result = $blockCipher->encrypt('this is a secret message');
echo "Encrypted text: $result \n";
```

The BlockCipher is initialized using a factory method with the name of the cipher adapter to use (mcrypt) and the parameters to pass to the adapter (the AES algorithm). In order to encrypt a string we need to specify an encryption key and we used the `setKey()` method for that scope. The encryption is provided by the `encrypt()` method.

The output of the encryption is a string, encoded in Base64 (default), that contains the HMAC value, the IV vector, and the encrypted text. The encryption mode used is the CBC (with a random IV by default) and SHA256 as default hash algorithm of the HMAC. The Mcrypt adapter encrypts using the PKCS#7 padding mechanism by default. You can specify a different padding method using a special adapter for that (Zend\Crypt\Symmetric\Padding). The encryption and authentication keys used by the `BlockCipher` are generated with the PBKDF2 algorithm, used as key derivation function from the user's key specified using the `setKey()` method.

Note: Key size

BlockCipher try to use always the longest size of the key for the specified cipher. For instance, for the AES algorithm it uses 256 bits and for the Blowfish algorithm it uses 448 bits.

You can change all the default settings passing the values to the factory parameters. For instance, if you want to use the Blowfish algorithm, with the CFB mode and the SHA512 hash function for HMAC you have to initialize the class as follow:

```
use Zend\Crypt\BlockCipher;

$blockCipher = BlockCipher::factory('mcrypt', array(
```

```
                                    'algo' => 'blowfish',
                                    'mode' => 'cfb',
                                    'hash' => 'sha512'
                                ));
```

Note: Recommendation

If you are not familiar with symmetric encryption techniques we strongly suggest to use the default values of the `BlockCipher` class. The default values are: AES algorithm, CBC mode, HMAC with SHA256, PKCS#7 padding.

To decrypt a string we can use the `decrypt()` method. In order to successfully decrypt a string we have to configure the BlockCipher with the same parameters of the encryption.

We can also initialize the BlockCipher manually without use the factory method. We can inject the symmetric cipher adapter directly to the constructor of the BlockCipher class. For instance, we can rewrite the previous example as follow:

```
use Zend\Crypt\BlockCipher;
use Zend\Crypt\Symmetric\Mcrypt;

$blockCipher = new BlockCipher(new Mcrypt(array('algo' => 'aes'));
$blockCipher->setKey('encryption key');
$result = $blockCipher->encrypt('this is a secret message');
echo "Encrypted text: $result \n";
```

CHAPTER

SIXTYTWO

KEY DERIVATION FUNCTION

In cryptography, a key derivation function (or KDF) derives one or more secret keys from a secret value such as a master key or other known information such as a password or passphrase using a pseudo-random function. For instance, a KDF function can be used to generate encryption or authentication keys from a user password. The `Zend\Crypt\Key\Derivation` implements a key derivation function using specific adapters.

User passwords are not really suitable to be used as keys in cryptographic algorithms, since users normally choose keys they can write on keyboard. These passwords use only 6 to 7 bits per character (or less). It is highly recommended to use always a KDF function to transform a user's password in a cryptographic key.

The output of the following key derivation functions is a binary string. If you need to store the value in a database or a different persistent storage, we suggest to convert it in Base64 format, using base64_encode() function, or in hex format, using the bin2hex() function.

62.1 Pbkdf2 adapter

Pbkdf2 is a KDF that applies a pseudorandom function, such as a cryptographic hash, to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. The added computational work makes password cracking much more difficult, and is known as key stretching.

In the example below we show a typical usage of the `Pbkdf2` adapter.

```
use Zend\Crypt\Key\Derivation\Pbkdf2;
use Zend\Math\Rand;

$pass = 'password';
$salt = Rand::getBytes(strlen($pass), true);
$key  = Pbkdf2::calc('sha256', $pass, $salt, 10000, strlen($pass)*2);

printf ("Original password: %s\n", $pass);
printf ("Derived key (hex): %s\n", bin2hex($key));
```

The `Pbkdf2` adapter takes the password (`$pass`) and generate a binary key with a size double of the password. The syntax is `calc($hash, $pass, $salt, $iterations, $length)` where `$hash` is the name of the hash function to use, `$pass` is the password, `$salt` is a pseudo random value, `$iterations` is the number of iterations of the algorithm and `$length` is the size of the key to be generated. We used the `Rand::getBytes` function of the `Zend\Math\Rand` class to generate a random bytes using a strong generators (the `true` value means the usage of strong generators).

The number of iterations is a very important parameter for the security of the algorithm. Big values means more security. There is not a fixed value for that because the number of iterations depends on the CPU power. You should always choose a number of iteration that prevent brute force attacks. For instance, a value of 1'000'000 iterations, that

is equal to 1 sec of elaboration for the PBKDF2 algorithm, is enough secure using an Intel Core i5-2500 CPU at 3.3 Ghz.

62.2 SaltedS2k adapter

The SaltedS2k algorithm uses an hash function and a salt to generate a key based on a user's password. This algorithm doesn't use a parameter that specify the number of iterations and for that reason it's considered less secure compared with Pbkdf2. We suggest to use the SaltedS2k algorithm only if you really need it.

Below is reported a usage example of the `SaltedS2k` adapter.

```
use Zend\Crypt\Key\Derivation\SaltedS2k;
use Zend\Math\Rand;

$pass = 'password';
$salt = Rand::getBytes(strlen($pass), true);
$key  = SaltedS2k::calc('sha256', $pass, $salt, strlen($pass)*2);

printf ("Original password: %s\n", $pass);
printf ("Derived key (hex): %s\n", bin2hex($key));
```

62.3 Scrypt adapter

The scrypt algorithm uses the algorithm Salsa20/8 core and *Pbkdf2-SHA256* to generate a key based on a user's password. This algorithm has been designed to be more secure against hardware brute-force attacks than alternative functions such as Pbkdf2 or bcrypt.

The scrypt algorithm is based on the idea of memory-hard algorithms and sequential memory-hard functions. A memory-hard algorithm is thus an algorithm which asymptotically uses almost as many memory locations as it uses operations[#f1]_. A natural way to reduce the advantage provided by an attacker's ability to construct highly parallel circuits is to increase the size of a single key derivation circuit — if a circuit is twice as large, only half as many copies can be placed on a given area of silicon — while still operating within the resources available to software implementations, including a powerful CPU and large amounts of RAM.

> "From a test executed on modern (2009) hardware, if 5 seconds are spent computing a derived key, the cost of a hardware brute-force attack against scrypt is roughly 4000 times greater than the cost of a similar attack against bcrypt (to find the same password), and 20000 times greater than a similar attack against Pbkdf2." *Colin Percival* (the author of scrypt algorithm)

This algorithm uses 4 parameters to generate a key of 64 bytes:

- `salt`, a random string;
- `N`, the CPU cost;
- `r`, the memory cost;
- `p`, the parallelization cost.

Below is reported a usage example of the `Scrypt` adapter.

```
use Zend\Crypt\Key\Derivation\Scrypt;
use Zend\Math\Rand;

$pass = 'password';
$salt = Rand::getBytes(strlen($pass), true);
```

```
6 $key  = Scrypt::calc($pass, $salt, 2048, 2, 1, 64);
7
8 printf ("Original password: %s\n", $pass);
9 printf ("Derived key (hex): %s\n", bin2hex($key));
```

Note: Performance of the scrypt implementation

The aim of the scrypt algorithm is to generate secure derived key preventing brute force attacks. Just like the other derivation functions, the more time (and memory) we spent executing the algorithm, the more secure the derived key will be. Unfortunately a pure PHP implementation of the scrypt algorithm is very slow compared with the C implementation (this is always true, if you compare execution time of C with PHP). If you want use a faster scrypt algorithm we suggest to use the C implementation of scrypt, supported by this Scrypt extension for PHP (please note that this PHP extension is not officially supported by php.net). The Scrypt adapter of Zend Framework is able to recognize if this extension is loaded and use it instead of the pure PHP implementation.

CHAPTER

SIXTYTHREE

PASSWORD

In the `Zend\Crypt\Password` namespace you can find all the password formats supported by Zend Framework. We currently support the following passwords:

- bcrypt;
- Apache (htpasswd).

If you need to choose a password format to store the user's password we suggest to use the *bcrypt* algorithm that is considered secure against brute forcing attacks (see the details below).

63.1 Bcrypt

The bcrypt algorithm is an hashing algorithm that is widely used and suggested by the security community to store user's passwords in a secure way.

Classic hashing mechanisms like MD5 or SHA, with or without a *salt* value, are not considered secure anymore (read this post to know why).

The security of bcrypt is related to the speed of the algorithm. Bcrypt is very slow, it can request even a second to generate an hash value. That means a brute force attack is impossible to execute, due to the amount of time that its need.

Bcrypt uses a *cost* parameter that specify the number of cycles to use in the algorithm. Increasing this number the algorithm will spend more time to generate the hash output. The *cost* parameter is represented by an integer value between 4 to 31. The default *cost* value of the `Zend\Crypt\Password\Bcrypt` component is 14, that means almost a second using a CPU Intel i5 at 3.3Ghz (the *cost* parameter is a relative value according to the speed of the CPU used).

If you want to change the *cost* parameter of the bcrypt algorithm you can use the `setCost()` method. Please note that if you change the cost parameter, the resulting hash will be different. This will not affect the verification process of the algorithm, therefore not breaking the password hashes you already have stored. Bcrypt reads the *cost* parameter from the hash value, during the password authentication. All of the parts needed to verify the hash are all together, separated with $'s, first the algorithm, then the cost, the salt, and then finally the hash.

The example below shows how to use the bcrypt algorithm to store a user's password:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt();
$securePass = $bcrypt->create('user password');
```

The output of the `create()` method is the hash of the password. This value can then be stored in a repository like a database (the output is a string of 60 bytes).

To verify if a given password is valid against a bcrypt value you can use the `verify()` method. An example is reported below:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt();
$securePass = 'the stored bcrypt value';
$password = 'the password to check';

if ($bcrypt->verify($password, $securePass)) {
    echo "The password is correct! \n";
} else {
    echo "The password is NOT correct.\n";
}
```

In the bcrypt uses also a *salt* value to improve the randomness of the algorithm. By default, the `Zend\Crypt\Password\Bcrypt` component generates a random salt for each hash. If you want to specify a preselected salt you can use the `setSalt()` method.

We provide also a `getSalt()` method to retrieve the *salt* specified by the user. The *salt* and the *cost* parameter can be also specified during the constructor of the class, below is reported an example:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt(array(
    'salt' => 'random value',
    'cost' => 13
));
```

Note: Bcrypt with non-ASCII passwords (8-bit characters)

The bcrypt implementation used by PHP < 5.3.7 can contains a security flaw if the password uses 8-bit characters (here's the security report). The impact of this bug was that most (but not all) passwords containing non-ASCII characters with the 8th bit set were hashed incorrectly, resulting in password hashes incompatible with those of OpenBSD's original implementation of bcrypt. This security flaw has been fixed starting from PHP 5.3.7 and the prefix used in the output was changed to '$2y$' in order to put evidence on the correctness of the hash value. If you are using PHP < 5.3.7 with 8-bit passwords, the `Zend\Crypt\Password\Bcrypt` throws an exception suggesting to upgrade to PHP 5.3.7+ or use only 7-bit passwords.

63.2 Apache

The `Zend\Crypt\Password\Apache` supports all the password formats used by Apache (htpasswd). These formats are:

- *CRYPT*, uses the traditional Unix crypt(3) function with a randomly-generated 32-bit salt (only 12 bits used) and the first 8 characters of the password;
- *SHA1*, "{SHA}" + Base64-encoded SHA-1 digest of the password;
- *MD5*, "$apr1$" + the result of an Apache-specific algorithm using an iterated (1,000 times) MD5 digest of various combinations of a random 32-bit salt and the password.
- *Digest*, the MD5 hash of the string *user*:*realm*:*password* as a 32-character string of hexadecimal digits. *realm* is the Authorization Realm argument to the *AuthName* directive in httpd.conf.

In order to specify the format of the Apache's password you can use the `setFormat()` method. An example with all the formats usage is reported below:

```
use Zend\Crypt\Password\Apache;

$apache = new Apache();

$apache->setFormat('crypt');
printf ("CRYPT output: %s\n", $apache->create('password'));

$apache->setFormat('sha1');
printf ("SHA1 output: %s\n", $apache->create('password'));

$apache->setFormat('md5');
printf ("MD5 output: %s\n", $apache->create('password'));

$apache->setFormat('digest');
$apache->setUserName('enrico');
$apache->setAuthName('test');
printf ("Digest output: %s\n", $apache->create('password'));
```

You can also specify the format of the password during the constructor of the class:

```
use Zend\Crypt\Password\Apache;

$apache = new Apache(array(
    'format' => 'md5'
));
```

Other possible parameters to pass in the constructor are *username* and *authname*, for the digest format.

CHAPTER

SIXTYFOUR

PUBLIC KEY CRYPTOGRAPHY

Public-key cryptography refers to a cryptographic system requiring two separate keys, one of which is secret and one of which is public. Although different, the two parts of the key pair are mathematically linked. One key locks or encrypts the plaintext, and the other unlocks or decrypts the cyphertext. Neither key can perform both functions. One of these keys is published or public, while the other is kept private.

In Zend Framework we implemented two public key algorithms: Diffie-Hellman key exchange and RSA.

64.1 Diffie-Hellman

The Diffie-Hellman algorithm is a specific method of exchanging cryptographic keys. It is one of the earliest practical examples of key exchange implemented within the field of cryptography. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

The diagram of operation of the Diffie-Hellman algorithm can be defined by the following picture (taken by the Diffie-Hellman Wikipedia page):

The schema's colors represent the parameters of the algorithm. Here is reported an example of usage using the `Zend\Crypt\PublicKey\DiffieHellman` class:

```
use Zend\Crypt\PublicKey\DiffieHellman;

$aliceOptions = array(
    'prime'     => '155172898181473697471232257763715539915724801966915404479707795314057629378541917
                   '423698188993727816152646631438561595825688188889951272158842675419950341258706556
                   '104870537681476726513255747040765857479291291572334510643245094715007229621094194
                   '98476037559498584825335930558543963844 3',
    'generator'=> '2',
    'private'   => '992093140665725952364085695919679885571412495614942674862518080355353963322786201
                  '813127128916726230726309951803243888416814918577455156967890911274095150092503589
                  '463420498381785213791321533481399080168191962194483101070726325157493390557981225
                  '048287025237969518005750318710516780 91'
);

$bobOptions   = array(
    'prime'     => $aliceOptions['prime'],
    'generator'=> '2',
    'private'   => '334117357926395586257336357178925636125481806504021611510774783148414637079488997
                   '123256347304105519467727528801778689728169635518217403867000760342134081539246925
```

```
                    '6346473315660054548451083307242700347420706465071483108330449773716038209708335687
                    '3161697260870332230258547131926127566 4'
);

$alice = new DiffieHellman($aliceOptions['prime'], $aliceOptions['generator'], $aliceOptions['private
$bob   = new DiffieHellman($bobOptions['prime'], $bobOptions['generator'], $bobOptions['private']);

$alice->generateKeys();
$bob->generateKeys();

$aliceSecretKey = $alice->computeSecretKey($bob->getPublicKey(DiffieHellman::FORMAT_BINARY),
                                           DiffieHellman::FORMAT_BINARY,
                                           DiffieHellman::FORMAT_BINARY);

$bobSecretKey   = $bob->computeSecretKey($alice->getPublicKey(DiffieHellman::FORMAT_BINARY),
                                         DiffieHellman::FORMAT_BINARY,
                                         DiffieHellman::FORMAT_BINARY);

if ($aliceSecretKey !== $bobSecretKey) {
    echo "ERROR!\n";
} else {
    printf("The secret key is: %s\n", base64_encode($aliceSecretKey));
}
```

The parameters of the Diffie-Hellman class are: a prime number (p), a generator (g) that is a primitive root mod p and a private integer number. The security of the Diffie-Hellman exchange algorithm is related to the choice of these parameters. To know how to choose secure numbers you can read the RFC 3526 document.

Note: The `Zend\Crypt\PublicKey\DiffieHellman` class use by default the OpenSSL extension of PHP to generate the parameters. If you don't want to use the OpenSSL library you have to set the `useOpensslExtension` static method to `false`.

64.2 RSA

RSA is an algorithm for public-key cryptography that is based on the presumed difficulty of factoring large integers, the factoring problem. A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message. Whether breaking RSA encryption is as hard as factoring is an open question known as the RSA problem.

The RSA algorithm can be used to encrypt/decrypt message and also to provide authenticity and integrity generating a digital signature of a message. Suppose that Alice wants to send an encrypted message to Bob. Alice must use the public key of Bob to encrypt the message. Bob can decrypt the message using his private key. Because Bob he is the only one that can access to his private key, he is the only one that can decrypt the message. If Alice wants to provide authenticity and integrity of a message to Bob she can use her private key to sign the message. Bob can check the correctness of the digital signature using the public key of Alice. Alice can provide encryption, authenticity and integrity of a message to Bob using the previous schemas in sequence, applying the encryption first and the digital signature after.

Below we reported some examples of usage of the `Zend\Crypt\PublicKey\Rsa` class in order to:

- generate a public key and a private key;
- encrypt/decrypt a string;

- generate a digital signature of a file.

64.2.1 Generate a public key and a private key

In order to generate a public and private key you can use the following code:

```
use Zend\Crypt\PublicKey\RsaOptions;

$rsaOptions = new RsaOptions(array(
    'pass_phrase' => 'test'
));

$rsaOptions->generateKeys(array(
    'private_key_bits' => 2048,
));

file_put_contents('private_key.pem', $rsaOptions->getPrivateKey());
file_put_contents('public_key.pub', $rsaOptions->getPublicKey());
```

This example generates a public and private key of 2048 bit storing the keys in two separate files, the `private_key.pem` for the private key and the `public_key.pub` for the public key. You can also generate the public and private key using OpenSSL from the command line (Unix style syntax):

```
ssh-keygen -t rsa
```

64.2.2 Encrypt and decrypt a string

Below is reported an example on how to encrypt and decrypt a string using the RSA algorithm. You can encrypt only small strings. The maximum size of encryption is given by the length of the public/private key - 88 bits. For instance, if we use a size of 2048 bit you can encrypt string with a maximum size of 1960 bit (245 characters). This limitation is related to the OpenSSL implementation for a security reason related to the nature of the RSA algorithm.

The normal application of a public key encryption algorithm is to store a key or a hash of the data you want to respectively encrypt or sign. A hash is typically 128-256 bits (the PHP sha1() function returns a 160 bit hash). An AES encryption key is 128 to 256 bits. So either of those will comfortably fit inside a single RSA encryption.

```
use Zend\Crypt\PublicKey\Rsa;

$rsa = Rsa::factory(array(
    'public_key'    => 'public_key.pub',
    'private_key'   => 'private_key.pem',
    'pass_phrase'   => 'test',
    'binary_output' => false
));

$text = 'This is the message to encrypt';

$encrypt = $rsa->encrypt($text);
printf("Encrypted message:\n%s\n", $encrypt);

$decrypt = $rsa->decrypt($encrypt);

if ($text !== $decrypt) {
    echo "ERROR\n";
} else {
```

```
    echo "Encryption and decryption performed successfully!\n";
}
```

64.2.3 Generate a digital signature of a file

Below is reported an example of how to generate a digital signature of a file.

```
use Zend\Crypt\PublicKey\Rsa;

$rsa = Rsa::factory(array(
    'private_key'   => 'path/to/private_key',
    'pass_phrase'   => 'passphrase of the private key',
    'binary_output' => false
));

$file = file_get_contents('path/file/to/sign');

$signature = $rsa->sign($file, $rsa->getOptions()->getPrivateKey());
$verify    = $rsa->verify($file, $signature, $rsa->getOptions()->getPublicKey());

if ($verify) {
    echo "The signature is OK\n";
    file_put_contents($filename . '.sig', $signature);
    echo "Signature save in $filename.sig\n";
} else {
     echo "The signature is not valid!\n";
}
```

In this example we used the Base64 format to encode the digital signature of the file (`binary_output` is false).

Note: The implementation of `Zend\Crypt\PublicKey\Rsa` algorithm uses the OpenSSL extension of PHP.

CHAPTER

SIXTYFIVE

ZEND\DB\ADAPTER

The Adapter object is the most important sub-component of `Zend\Db`. It is responsible for adapting any code written in or for Zend\Db to the targeted php extensions and vendor databases. In doing this, it creates an abstraction layer for the PHP extensions, which is called the "Driver" portion of the `Zend\Db` adapter. It also creates a lightweight abstraction layer, called the "Platform" portion of the adapter, for the various idiosyncrasies that each vendor-specific platform might have in its SQL/RDBMS implementation.

65.1 Creating an Adapter - Quickstart

Creating an adapter can simply be done by instantiating the `Zend\Db\Adapter\Adapter` class. The most common use case, while not the most explicit, is to pass an array of configuration to the `Adapter`.

```
$adapter = new Zend\Db\Adapter\Adapter($configArray);
```

This driver array is an abstraction for the extension level required parameters. Here is a table for the key-value pairs that should be in configuration array.

Key	Is Required?	Value
driver	required	Mysqli, Sqlsrv, Pdo_Sqlite, Pdo_Mysql, Pdo=OtherPdoDriver
database	generally required	the name of the database (schema)
username	generally required	the connection username
password	generally required	the connection password
hostname	not generally required	the IP address or hostname to connect to
port	not generally required	the port to connect to (if applicable)
charset	not generally required	the character set to use

Note: Other names will work as well. Effectively, if the PHP manual uses a particular naming, this naming will be supported by our Driver. For example, dbname in most cases will also work for 'database'. Another example is that in the case of Sqlsrv, UID will work in place of username. Which format you chose is up to you, but the above table represents the official abstraction names.

So, for example, a MySQL connection using ext/mysqli:

```
$adapter = new Zend\Db\Adapter\Adapter(array(
    'driver' => 'Mysqli',
    'database' => 'zend_db_example',
    'username' => 'developer',
    'password' => 'developer-password'
));
```

Another example, of a Sqlite connection via PDO:

```
$adapter = new Zend\Db\Adapter\Adapter(array(
    'driver' => 'Pdo_Sqlite',
    'database' => 'path/to/sqlite.db'
));
```

It is important to know that by using this style of adapter creation, the `Adapter` will attempt to create any dependencies that were not explicitly provided. A Driver object will be created from the configuration array provided in the constructor. A Platform object will be created based off the type of Driver class that was instantiated. And lastly, a default ResultSet object is created and utilized. Any of these objects can be injected, to do this, see the next section.

The list of officially supported drivers:

- `Mysqli`: The ext/mysqli driver
- `Pgsql`: The ext/pgsql driver
- `Sqlsrv`: The ext/sqlsrv driver (from Microsoft)
- `Pdo_Mysql`: MySQL through the PDO extension
- `Pdo_Sqlite`: SQLite though the PDO extension
- `Pdo_Pgsql`: PostgreSQL through the PDO extension

65.2 Creating an Adapter Using Dependency Injection

The more expressive and explicit way of creating an adapter is by injecting all your dependencies up front. `Zend\Db\Adapter\Adapter` uses constructor injection, and all required dependencies are injected through the constructor, which has the following signature (in pseudo-code):

```
use Zend\Db\Adapter\Platform\PlatformInterface;
use Zend\Db\ResultSet\ResultSet;

class Zend\Db\Adapter\Adapter {
    public function __construct($driver, PlatformInterface $platform = null, ResultSet $queryResultSe
}
```

What can be injected:

- $driver - an array of connection parameters (see above) or an instance of `Zend\Db\Adapter\Driver\DriverInterface`
- $platform - (optional) an instance of `Zend\Db\Platform\PlatformInterface`, the default will be created based off the driver implementation
- $queryResultSetPrototype - (optional) an instance of `Zend\Db\ResultSet\ResultSet`, to understand this object's role, see the section below on querying through the adapter

65.3 Query Preparation Through Zend\Db\Adapter\Adapter::query()

By default, query() prefers that you use "preparation" as a means for processing SQL statements. This generally means that you will supply a SQL statement with the values substituted by placeholders, and then the parameters for those placeholders are supplied separately. An example of this workflow with `Zend\Db\Adapter\Adapter` is:

```
$adapter->query('SELECT * FROM `artist` WHERE `id` = ?', array(5));
```

The above example will go through the following steps:

- create a new Statement object
- prepare an array into a ParameterContainer if necessary
- inject the ParameterContainer into the Statement object
- execute the Statement object, producing a Result object
- check the Result object to check if the supplied sql was a "query", or a result set producing statement
- if it is a result set producing query, clone the ResultSet prototype, inject Result as datasource, return it
- else, return the Result

65.4 Query Execution Through Zend\Db\Adapter\Adapter::query()

In some cases, you have to execute statements directly. The primary purpose for needing to execute sql instead of prepare and execute a sql statement, might be because you are attempting to execute a DDL statement (which in most extensions and vendor platforms), are un-preparable. An example of executing:

```
$adapter->query('ALTER TABLE ADD INDEX(`foo_index`) ON (`foo_column`)', Adapter::QUERY_MODE_EXECUTE);
```

The primary difference to notice is that you must provide the Adapter::QUERY_MODE_EXECUTE (execute) as the second parameter.

65.5 Creating Statements

While query() is highly useful for one-off and quick querying of a database through Adapter, it generally makes more sense to create a statement and interact with it directly, so that you have greater control over the prepare-then-execute workflow. To do this, Adapter gives you a routine called createStatement() that allows you to create a Driver specific Statement to use so you can manage your own prepare-then-execute workflow.

```
// with optional parameters to bind up-front
$statement = $adapter->createStatement($sql, $optionalParameters);
$result = $statement->execute();
```

65.6 Using the Driver Object

The Driver object is the primary place where `Zend\Db\Adapter\Adapter` implements the connection level abstraction making it possible to use all of ZendDb's interfaces via the various ext/mysqli, ext/sqlsrv, PDO, and other PHP level drivers. To make this possible, each driver is composed of 3 objects:

- A connection: `Zend\Db\Adapter\Driver\ConnectionInterface`
- A statement: `Zend\Db\Adapter\Driver\StatementInterface`
- A result: `Zend\Db\Adapter\Driver\ResultInterface`

Each of the built-in drivers practices "prototyping" as a means of creating objects when new instances are requested. The workflow looks like this:

- An adapter is created with a set of connection parameters
- The adapter chooses the proper driver to instantiate, for example `Zend\Db\Adapter\Driver\Mysqli`
- That driver class is instantiated

- If no connection, statement or result objects are injected, defaults are instantiated

This driver is now ready to be called on when particular workflows are requested. Here is what the Driver API looks like:

```
namespace Zend\Db\Adapter\Driver;

 interface DriverInterface
 {
     const PARAMETERIZATION_POSITIONAL = 'positional';
     const PARAMETERIZATION_NAMED = 'named';
     const NAME_FORMAT_CAMELCASE = 'camelCase';
     const NAME_FORMAT_NATURAL = 'natural';
     public function getDatabasePlatformName($nameFormat = self::NAME_FORMAT_CAMELCASE);
     public function checkEnvironment();
     public function getConnection();
     public function createStatement($sqlOrResource = null);
     public function createResult($resource);
     public function getPrepareType();
     public function formatParameterName($name, $type = null);
     public function getLastGeneratedValue();
 }
```

From this DriverInterface, you can

- Determine the name of the platform this driver supports (useful for choosing the proper platform object)
- Check that the environment can support this driver
- Return the Connection object
- Create a Statement object which is optionally seeded by an SQL statement (this will generally be a clone of a prototypical statement object)
- Create a Result object which is optionally seeded by a statement resource (this will generally be a clone of a prototypical result object)
- Format parameter names, important to distinguish the difference between the various ways parameters are named between extensions
- Retrieve the overall last generated value (such as an auto-increment value)

Statement objects generally look like this:

```
namespace Zend\Db\Adapter\Driver;

interface StatementInterface extends StatementContainerInterface
{
    public function getResource();
    public function prepare($sql = null);
    public function isPrepared();
    public function execute($parameters = null);

    /** Inherited from StatementContainerInterface */
    public function setSql($sql);
    public function getSql();
    public function setParameterContainer(ParameterContainer $parameterContainer);
    public function getParameterContainer();
}
```

Result objects generally look like this:

```
namespace Zend\Db\Adapter\Driver;

interface ResultInterface extends \Countable, \Iterator
{
    public function buffer();
    public function isQueryResult();
    public function getAffectedRows();
    public function getGeneratedValue();
    public function getResource();
    public function getFieldCount();
}
```

65.7 Using The Platform Object

The Platform object provides an API to assist in crafting queries in a way that is specific to the SQL implementation of a particular vendor. Nuances such as how identifiers or values are quoted, or what the identifier separator character is are handled by this object. To get an idea of the capabilities, the interface for a platform object looks like this:

```
namespace Zend\Db\Adapter\Platform;

interface PlatformInterface
{
    public function getName();
    public function getQuoteIdentifierSymbol();
    public function quoteIdentifier($identifier);
    public function quoteIdentifierChain($identiferChain)
    public function getQuoteValueSymbol();
    public function quoteValue($value);
    public function quoteValueList($valueList);
    public function getIdentifierSeparator();
    public function quoteIdentifierInFragment($identifier, array $additionalSafeWords = array());
}
```

While one can instantiate your own Platform object, generally speaking, it is easier to get the proper Platform instance from the configured adapter (by default the Platform type will match the underlying driver implementation):

```
$platform = $adapter->getPlatform();
// or
$platform = $adapter->platform; // magic property access
```

The following is a couple of example of Platform usage:

```
/** @var $adapter Zend\Db\Adapter\Adapter */
/** @var $platform Zend\Db\Adapter\Platform\Sql92 */
$platform = $adapter->getPlatform();

// "first_name"
echo $platform->quoteIdentifier('first_name');

// "
echo $platform->getQuoteIdentifierSymbol();

// "schema"."mytable"
echo $platform->quoteIdentifierChain(array('schema','mytable'));

// '
```

```
echo $platform->getQuoteValueSymbol();

// 'myvalue'
echo $platform->quoteValue('myvalue');

// 'value', 'Foo O\\'Bar'
echo $platform->quoteValueList(array('value',"Foo O'Bar"));

// .
echo $platform->getIdentifierSeparator();

// "foo" as "bar"
echo $platform->quoteIdentifierInFragment('foo as bar');

// additionally, with some safe words:
// ("foo"."bar" = "boo"."baz")
echo $platform->quoteIdentifierInFragment('(foo.bar = boo.baz)', array('(', ')', '='));
```

65.8 Using The Parameter Container

The ParameterContainer object is a container for the various parameters that need to be passed into a Statement object to fulfill all the various parameterized parts of the SQL statement. This object implements the ArrayAccess interface. Below is the ParameterContainer API:

```
namespace Zend\Db\Adapter;

 class ParameterContainer implements \Iterator, \ArrayAccess, \Countable {
     public function __construct(array $data = array())

     /** methods to interact with values */
     public function offsetExists($name)
     public function offsetGet($name)
     public function offsetSetReference($name, $from)
     public function offsetSet($name, $value, $errata = null)
     public function offsetUnset($name)

     /** set values from array (will reset first) */
     public function setFromArray(Array $data)

     /** methods to interact with value errata */
     public function offsetSetErrata($name, $errata)
     public function offsetGetErrata($name)
     public function offsetHasErrata($name)
     public function offsetUnsetErrata($name)

     /** errata only iterator */
     public function getErrataIterator()

     /** get array with named keys */
     public function getNamedArray()

     /** get array with int keys, ordered by position */
     public function getPositionalArray()

     /** iterator: */
     public function count()
```

```
    public function current()
    public function next()
    public function key()
    public function valid()
    public function rewind()

    /** merge existing array of parameters with existing parameters */
    public function merge($parameters)
}
```

In addition to handling parameter names and values, the container will assist in tracking parameter types for PHP type to SQL type handling. For example, it might be important that:

```
$container->offsetSet('limit', 5);
```

be bound as an integer. To achieve this, pass in the ParameterContainer::TYPE_INTEGER constant as the 3rd parameter:

```
$container->offsetSet('limit', 5, $container::TYPE_INTEGER);
```

This will ensure that if the underlying driver supports typing of bound parameters, that this translated information will also be passed along to the actual php database driver.

65.9 Examples

Creating a Driver and Vendor portable Query, Preparing and Iterating Result

```
$adapter = new Zend\Db\Adapter\Adapter($driverConfig);

$qi = function($name) use ($adapter) { return $adapter->platform->quoteIdentifier($name); };
$fp = function($name) use ($adapter) { return $adapter->driver->formatParameterName($name); };

$sql = 'UPDATE ' . $qi('artist')
    . ' SET ' . $qi('name') . ' = ' . $fp('name')
    . ' WHERE ' . $qi('id') . ' = ' . $fp('id');

/** @var $statement Zend\Db\Adapter\Driver\StatementInterface */
$statement = $adapter->query($sql);

$parameters = array(
    'name' => 'Updated Artist',
    'id' => 1
);

$statement->execute($parameters);

// DATA INSERTED, NOW CHECK

/* @var $statement Zend\Db\Adapter\DriverStatementInterface */
$statement = $adapter->query('SELECT * FROM '
    . $qi('artist')
    . ' WHERE id = ' . $fp('id'));

/* @var $results Zend\Db\ResultSet\ResultSet */
$results = $statement->execute(array('id' => 1));

```

```
30 $row = $results->current();
31 $name = $row['name'];
```

CHAPTER

SIXTYSIX

ZEND\DB\RESULTSET

`Zend\Db\ResultSet` is a sub-component of Zend\Db for abstracting the iteration of rowset producing queries. While data sources for this can be anything that is iterable, generally a `Zend\Db\Adapter\Driver\ResultInterface` based object is the primary source for retrieving data.

`Zend\Db\ResultSet`'s must implement the `Zend\Db\ResultSet\ResultSetInterface` and all sub-components of Zend\Db that return a ResultSet as part of their API will assume an instance of a `ResultSetInterface` should be returned. In most casts, the Prototype pattern will be used by consuming object to clone a prototype of a ResultSet and return a specialized ResultSet with a specific data source injected. The interface of ResultSetInterface looks like this:

```
interface ResultSetInterface extends \Traversable, \Countable
{
    public function initialize($dataSource);
    public function getFieldCount();
}
```

66.1 Quickstart

`Zend\Db\ResultSet\ResultSet` is the most basic form of a ResultSet object that will expose each row as either an ArrayObject-like object or an array of row data. By default, `Zend\Db\Adapter\Adapter` will use a prototypical `Zend\Db\ResultSet\ResultSet` object for iterating when using the `Zend\Db\Adapter\Adapter::query()` method.

The following is an example workflow similar to what one might find inside `Zend\Db\Adapter\Adapter::query()`:

```
use Zend\Db\Adapter\Driver\ResultInterface;
use Zend\Db\ResultSet\ResultSet;

$stmt = $driver->createStatement('SELECT * FROM users');
$stmt->prepare();
$result = $stmt->execute($parameters);

if ($result instanceof ResultInterface && $result->isQueryResult()) {
    $resultSet = new ResultSet;
    $resultSet->initialize($result);

    foreach ($resultSet as $row) {
        echo $row->my_column . PHP_EOL;
    }
}
```

66.2 Zend\Db\ResultSet\ResultSet and Zend\Db\ResultSet\AbstractResultSet

For most purposes, either a instance of `Zend\Db\ResultSet\ResultSet` or a derivative of `Zend\Db\ResultSet\AbstractResultSet` will be being used. The implementation of the `AbstractResultSet` offers the following core functionality:

```
abstract class AbstractResultSet implements Iterator, ResultSetInterface
{
    public function initialize($dataSource)
    public function getDataSource()
    public function getFieldCount()

    /** Iterator */
    public function next()
    public function key()
    public function current()
    public function valid()
    public function rewind()

    /** countable */
    public function count()

    /** get rows as array */
    public function toArray()
}
```

66.3 Zend\Db\ResultSet\HydratingResultSet

`Zend\Db\ResultSet\HydratingResultSet` is a more flexible `ResultSet` object that allows the developer to choose an appropriate "hydration strategy" for getting row data into a target object. While iterating over results, `HydratingResultSet` will take a prototype of a target object and clone it once for each row. The `HydratingResultSet` will then hydrate that clone with the row data.

In the example below, rows from the database will be iterated, and during iteration, `HydratingRowSet` will use the Reflection based hydrator to inject the row data directly into the protected members of the cloned UserEntity object:

```
use Zend\Db\Adapter\Driver\ResultInterface;
use Zend\Db\ResultSet\HydratingResultSet;
use Zend\Stdlib\Hydrator\Reflection as ReflectionHydrator;

class UserEntity {
    protected $first_name;
    protected $last_name;
    public function getFirstName() { return $this->first_name; }
    public function getLastName() { return $this->last_name; }
    public function setFirstName($first_name) { $this->first_name = $first_name; }
    public function setLastName($last_name) { $this->last_name = $last_name; }
}

$stmt = $driver->createStatement($sql);
$stmt->prepare($parameters);
$result = $stmt->execute();

if ($result instanceof ResultInterface && $result->isQueryResult()) {
    $resultSet = new HydratingResultSet(new ReflectionHydrator, new UserEntity);
    $resultSet->initialize($result);
```

```

    foreach ($resultSet as $user) {
        echo $user->getFirstName() . ' ' . $user->getLastName() . PHP_EOL;
    }
}
```

For more information, see the `Zend\Stdlib\Hydrator` documentation to get a better sense of the different strategies that can be employed in order to populate a target object.

CHAPTER

SIXTYSEVEN

ZEND\DB\SQL

`Zend\Db\Sql` is a SQL abstraction layer for building platform specific SQL queries via a object-oriented API. The end result of an `Zend\Db\Sql` object will be to either produce a Statement and Parameter container that represents the target query, or a full string that can be directly executed against the database platform. To achieve this, `Zend\Db\Sql` objects require a `Zend\Db\Adapter\Adapter` object in order to produce the desired results.

67.1 Zend\Db\Sql\Sql (Quickstart)

As there are four primary tasks associated with interacting with a database (from the DML, or Data Manipulation Language): selecting, inserting, updating and deleting. As such, there are four primary objects that developers can interact or building queries, `Zend\Db\Sql\Select`, `Insert`, `Update` and `Delete`.

Since these four tasks are so closely related, and generally used together within the same application, `Zend\Db\Sql\Sql` objects help you create them and produce the result you are attempting to achieve.

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select(); // @return Zend\Db\Sql\Select
$insert = $sql->insert(); // @return Zend\Db\Sql\Insert
$update = $sql->update(); // @return Zend\Db\Sql\Update
$delete = $sql->delete(); // @return Zend\Db\Sql\Delete
```

As a developer, you can now interact with these objects, as described in the sections below, to specialize each query. Once they have been populated with values, they are ready to either be prepared or executed.

To prepare (using a Select object):

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select();
$select->from('foo');
$select->where(array('id' => 2));

$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
```

To execute (using a Select object)

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select();
$select->from('foo');
$select->where(array('id' => 2));
```

```

$selectString = $sql->getSqlStringForSqlObject($select);
$results = $adapter->query($selectString, $adapter::QUERY_MODE_EXECUTE);
```

Zend\Db\Sql\Sql objects can also be bound to a particular table so that in getting a select, insert, update, or delete object, they are all primarily seeded with the same table when produced.

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter, 'foo');
$select = $sql->select();
$select->where(array('id' => 2)); // $select already has the from('foo') applied
```

67.2 Zend\Db\Sql's Select, Insert, Update and Delete

Each of these objects implements the following (2) interfaces:

```
interface PreparableSqlInterface {
     public function prepareStatement(Adapter $adapter, StatementInterface $statement);
}
interface SqlInterface {
     public function getSqlString(PlatformInterface $adapterPlatform = null);
}
```

These are the functions you can call to either produce (a) a prepared statement, or (b) a string to be executed.

67.3 Zend\Db\Sql\Select

`Zend\Db\Sql\Select` is an object who's primary function is to present a unified API for building platform specific SQL SELECT queries. The class can be instantiated and consumed without `Zend\Db\Sql\Sql`:

```
use Zend\Db\Sql\Select;
$select = new Select();
// or, to produce a $select bound to a specific table
$select = new Select('foo');
```

If a table is provided to the Select object, then from() cannot be called later to change the name of the table.

Once you have a valid Select object, the following API can be used to further specify various select statement parts:

```
class Select extends AbstractSql implements SqlInterface, PreparableSqlInterface
{
    const JOIN_INNER = 'inner';
    const JOIN_OUTER = 'outer';
    const JOIN_LEFT = 'left';
    const JOIN_RIGHT = 'right';
    const SQL_STAR = '*';
    const ORDER_ASCENDING = 'ASC';
    const ORDER_DESCENDING = 'DESC';

    public $where; // @param Where $where

    public function __construct($table = null);
    public function from($table);
    public function columns(array $columns, $prefixColumnsWithTable = true);
    public function join($name, $on, $columns = self::SQL_STAR, $type = self::JOIN_INNER);
```

```
    public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
    public function group($group);
    public function having($predicate, $combination = Predicate\PredicateSet::OP_AND);
    public function order($order);
    public function limit($limit);
    public function offset($offset);
}
```

67.3.1 from():

```
// as a string:
$select->from('foo');

// as an array to specify an alias:
// produces SELECT "t".* FROM "table" AS "t"

$select->from(array('t' => 'table'));

// using a Sql\TableIdentifier:
// same output as above

$select->from(new TableIdentifier(array('t' => 'table')));
```

67.3.2 columns():

```
// as array of names
$select->columns(array('foo', 'bar'));

// as an associative array with aliases as the keys:
// produces 'bar' AS 'foo', 'bax' AS 'baz'

$select->columns(array('foo' => 'bar', 'baz' => 'bax'));
```

67.3.3 join():

```
$select->join(
     'foo' // table name,
     'id = bar.id', // expression to join on (will be quoted by platform object before insertion),
     array('bar', 'baz'), // (optional) list of columns, same requirements as columns() above
     $select::JOIN_OUTER // (optional), one of inner, outer, left, right also represented by constant
);

$select->from(array('f' => 'foo'))  // base table
    ->join(array('b' => 'bar'),     // join table with alias
    'f.foo_id = b.foo_id');         // join expression
```

67.3.4 where(), having():

The `Zend\Db\Sql\Select` object provides bit of flexibility as it regards to what kind of parameters are acceptable when calling where() or having(). The method signature is listed as:

```
/**
 * Create where clause
 *
 * @param  Where|\Closure|string|array $predicate
 * @param  string $combination One of the OP_* constants from Predicate\PredicateSet
 * @return Select
 */
public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
```

As you can see, there are a number of different ways to pass criteria to both having() and where().

If you provide a `Zend\Db\Sql\Where` object to where() or a `Zend\Db\Sql\Having` object to having(), the internal objects for Select will be replaced completely. When the where/having() is processed, this object will be iterated to produce the WHERE or HAVING section of the SELECT statement.

If you provide a `Closure` to where() or having(), this function will be called with the Select's `Where` object as the only parameter. So the following is possible:

```
$spec = function (Where $where) {
    $where->like('username', 'ralph%');
};

$select->where($spec);
```

If you provide a string, this string will be used to instantiate a `Zend\Db\Sql\Predicate\Expression` object so that it's contents will be applied as is. This means that there will be no quoting in the fragment provided.

Consider the following code:

```
// SELECT "foo".* FROM "foo" WHERE x = 5
$select->from('foo')->where('x = 5');
```

If you provide an array who's values are keyed by an integer, the value can either be a string that will be then used to build a `Predicate\Expression` or any object that implements `Predicate\PredicateInterface`. These objects are pushed onto the Where stack with the $combination provided.

Consider the following code:

```
// SELECT "foo".* FROM "foo" WHERE x = 5 AND y = z
$select->from('foo')->where(array('x = 5', 'y = z'));
```

If you provide an array who's values are keyed with a string, these values will be handled in the following:

- PHP value nulls will be made into a `Predicate\IsNull` object
- PHP value array()s will be made into a `Predicate\In` object
- PHP value strings will be made into a `Predicate\Operator` object such that the string key will be identifier, and the value will target value.

Consider the following code:

```
// SELECT "foo".* FROM "foo" WHERE "c1" IS NULL AND "c2" IN (?, ?, ?) AND "c3" IS NOT NULL
$select->from('foo')->where(array(
    'c1' => null,
    'c2' => array(1, 2, 3),
    new \Zend\Db\Sql\Predicate\IsNotNull('c3')
));
```

67.3.5 order():

```
$select = new Select;
$select->order('id DESC'); // produces 'id' DESC

$select = new Select;
$select->order('id DESC')
     ->order('name ASC, age DESC'); // produces 'id' DESC, 'name' ASC, 'age' DESC

$select = new Select;
$select->order(array('name ASC', 'age DESC')); // produces 'name' ASC, 'age' DESC
```

67.3.6 limit() and offset():

```
$select = new Select;
$select->limit(5); // always takes an integer/numeric
$select->offset(10); // similarly takes an integer/numeric
```

67.4 Zend\Db\Sql\Insert

The Insert API:

```
class Insert implements SqlInterface, PreparableSqlInterface
{
     const VALUES_MERGE = 'merge';
     const VALUES_SET   = 'set';

     public function __construct($table = null);
     public function into($table);
     public function columns(array $columns);
     public function values(array $values, $flag = self::VALUES_SET);
}
```

Similarly to Select objects, the table can be set at construction time or via into().

67.4.1 columns():

```
$insert->columns(array('foo', 'bar')); // set the valid columns
```

67.4.2 values():

```
// default behavior of values is to set the values
// successive calls will not preserve values from previous calls
$insert->values(array(
     'col_1' => 'value1',
     'col_2' => 'value2'
));
```

```
// merging values with previous calls
$insert->values(array('col_2' => 'value2'), $insert::VALUES_MERGE);
```

67.5 Zend\Db\Sql\Update

```
class Update
{
    const VALUES_MERGE = 'merge';
    const VALUES_SET   = 'set';

    public $where; // @param Where $where
    public function __construct($table = null);
    public function table($table);
    public function set(array $values, $flag = self::VALUES_SET);
    public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
}
```

67.5.1 set():

```
$update->set(array('foo' => 'bar', 'baz' => 'bax'));
```

67.5.2 where():

See where section below.

67.6 Zend\Db\Sql\Delete

```
class Delete
{
    public $where; // @param Where $where
    public function __construct($table = null);
    public function from($table);
    public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
}
```

67.6.1 where():

See where section below.

67.7 Zend\Db\Sql\Where & Zend\Db\Sql\Having

In the following, we will talk about Where, Having is implies as being the same API.

Effectively, Where and Having extend from the same base object, a Predicate (and PredicateSet). All of the parts that make up a where or having that are and'ed or or'd together are called predicates. The full set of predicates is called a PredicateSet. This object set generally contains the values (and identifiers) separate from the fragment they belong to until the last possible moment when the statement is either used to be prepared (parameteritized), or executed. In parameterization, the parameters will be replaced with their proper placeholder (a named or positional parameter), and the values stored inside a Adapter\ParameterContainer. When executed, the values will be interpolated into the fragments they belong to and properly quoted.

It is important to know that in this API, a distinction is made between what elements are considered identifiers (TYPE_IDENTIFIER) and which of those is a value (TYPE_VALUE). There is also a special use case type for literal values (TYPE_LITERAL). These are all exposed via the `Zend\Db\Sql\ExpressionInterface` interface.

The Zend\Db\Sql\Where (Predicate/PredicateSet) API:

```
// Where & Having:
class Predicate extends PredicateSet
{
     public $and;
     public $or;
     public $AND;
     public $OR;
     public $NEST;
     public $UNNEST;

     public function nest();
     public function setUnnest(Predicate $predicate);
     public function unnest();
     public function equalTo($left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYP
     public function lessThan($left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TY
     public function greaterThan($left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self
     public function lessThanOrEqualTo($left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType =
     public function greaterThanOrEqualTo($left, $right, $leftType = self::TYPE_IDENTIFIER, $rightTyp
     public function like($identifier, $like);
     public function literal($literal, $parameter);
     public function isNull($identifier);
     public function isNotNull($identifier);
     public function in($identifier, array $valueSet = array());
     public function between($identifier, $minValue, $maxValue);


     // Inherited From PredicateSet

     public function addPredicate(PredicateInterface $predicate, $combination = null);
     public function getPredicates();
     public function orPredicate(PredicateInterface $predicate);
     public function andPredicate(PredicateInterface $predicate);
     public function getExpressionData();
     public function count();
}
```

Each method in the Where API will produce a corresponding Predicate object of a similarly named type, described below, with the full API of the object:

67.7.1 equalTo(), lessThan(), greaterThan(), lessThanOrEqualTo(), greaterThanOrEqualTo():

```
$where->equalTo('id', 5);

// same as the following workflow
$where->addPredicate(
     new Predicate\Operator($left, Operator::OPERATOR_EQUAL_TO, $right, $leftType, $rightType)
);

class Operator implements PredicateInterface
{
```

```
    const OPERATOR_EQUAL_TO                    = '=';
    const OP_EQ                                = '=';
    const OPERATOR_NOT_EQUAL_TO                = '!=';
    const OP_NE                                = '!=';
    const OPERATOR_LESS_THAN                   = '<';
    const OP_LT                                = '<';
    const OPERATOR_LESS_THAN_OR_EQUAL_TO       = '<=';
    const OP_LTE                               = '<=';
    const OPERATOR_GREATER_THAN                = '>';
    const OP_GT                                = '>';
    const OPERATOR_GREATER_THAN_OR_EQUAL_TO    = '>=';
    const OP_GTE                               = '>=';

    public function __construct($left = null, $operator = self::OPERATOR_EQUAL_TO, $right = null, $le
    public function setLeft($left);
    public function getLeft();
    public function setLeftType($type);
    public function getLeftType();
    public function setOperator($operator);
    public function getOperator();
    public function setRight($value);
    public function getRight();
    public function setRightType($type);
    public function getRightType();
    public function getExpressionData();
}
```

67.7.2 like($identifier, $like):

```
$where->like($identifier, $like):

// same as
$where->addPredicate(
     new Predicate\Like($identifier, $like)
);

// full API

class Like implements PredicateInterface
{
    public function __construct($identifier = null, $like = null);
    public function setIdentifier($identifier);
    public function getIdentifier();
    public function setLike($like);
    public function getLike();
}
```

67.7.3 literal($literal, $parameter);

```
$where->literal($literal, $parameter);

// same as
$where->addPredicate(
    new Predicate\Expression($literal, $parameter)
);
```

```

// full API
class Expression implements ExpressionInterface, PredicateInterface
{
    const PLACEHOLDER = '?';
     public function __construct($expression = null, $valueParameter = null /*[, $valueParameter, ..
    public function setExpression($expression);
    public function getExpression();
    public function setParameters($parameters);
    public function getParameters();
    public function setTypes(array $types);
    public function getTypes();
}
```

67.7.4 isNull($identifier);

```
$where->isNull($identifier);

// same as
$where->addPredicate(
    new Predicate\IsNull($identifier)
);

// full API
class IsNull implements PredicateInterface
{
    public function __construct($identifier = null);
    public function setIdentifier($identifier);
    public function getIdentifier();
}
```

67.7.5 isNotNull($identifier);

```
$where->isNotNull($identifier);

// same as
$where->addPredicate(
    new Predicate\IsNotNull($identifier)
);

// full API
class IsNotNull implements PredicateInterface
{
    public function __construct($identifier = null);
    public function setIdentifier($identifier);
    public function getIdentifier();
}
```

67.7.6 in($identifier, array $valueSet = array());

```
$where->in($identifier, array $valueSet = array());

// same as
```

```
$where->addPredicate(
    new Predicate\In($identifier, $valueSet)
);

// full API
class In implements PredicateInterface
{
    public function __construct($identifier = null, array $valueSet = array());
    public function setIdentifier($identifier);
    public function getIdentifier();
    public function setValueSet(array $valueSet);
    public function getValueSet();
}
```

67.7.7 between($identifier, $minValue, $maxValue);

```
$where->between($identifier, $minValue, $maxValue);

// same as
$where->addPredicate(
    new Predicate\Between($identifier, $minValue, $maxValue)
);

// full API
class Between implements PredicateInterface
{
    public function __construct($identifier = null, $minValue = null, $maxValue = null);
    public function setIdentifier($identifier);
    public function getIdentifier();
    public function setMinValue($minValue);
    public function getMinValue();
    public function setMaxValue($maxValue);
    public function getMaxValue();
    public function setSpecification($specification);
}
```

CHAPTER

SIXTYEIGHT

ZEND\DB\TABLEGATEWAY

The Table Gateway object is intended to provide an object that represents a table in a database, and the methods of this object mirror the most common operations on a database table. In code, the interface for such an object looks like this:

```
interface Zend\Db\TableGateway\TableGatewayInterface
{
    public function getTable();
    public function select($where = null);
    public function insert($set);
    public function update($set, $where = null);
    public function delete($where);
}
```

There are two primary implementations of the `TableGatewayInterface` that are of the most useful: `AbstractTableGateway` and `TableGateway`. The `AbstractTableGateway` is an abstract basic implementation that provides functionality for `select()`, `insert()`, `update()`, `delete()`, as well as an additional API for doing these same kinds of tasks with explicit SQL objects. These methods are `selectWith()`, `insertWith()`, `updateWith()` and `deleteWith()`. In addition, AbstractTableGateway also implements a "Feature" API, that allows for expanding the behaviors of the base `TableGateway` implementation without having to extend the class with this new functionality. The `TableGateway` concrete implementation simply adds a sensible constructor to the `AbstractTableGateway` class so that out-of-the-box, `TableGateway` does not need to be extended in order to be consumed and utilized to its fullest.

68.1 Basic Usage

The quickest way to get up and running with Zend\Db\TableGateway is to configure and utilize the concrete implementation of the `TableGateway`. The API of the concrete `TableGateway` is:

```
class TableGateway extends AbstractTableGateway
{
     public $lastInsertValue;
     public $table;
     public $adapter;

     public function __construct($table, Adapter $adapter, $features = null, ResultSet $resultSetProt

     /** Inherited from AbstractTableGateway */

    public function isInitialized();
    public function initialize();
    public function getTable();
```

```
    public function getAdapter();
    public function getColumns();
    public function getFeatureSet();
    public function getResultSetPrototype();
    public function getSql();
    public function select($where = null);
    public function selectWith(Select $select);
    public function insert($set);
    public function insertWith(Insert $insert);
    public function update($set, $where = null);
    public function updateWith(Update $update);
    public function delete($where);
    public function deleteWith(Delete $delete);
    public function getLastInsertValue();
}
```

The concrete `TableGateway` object practices constructor injection for getting dependencies and options into the instance. The table name and an instance of an Adapter are all that is needed to setup a working `TableGateway` object.

Out of the box, this implementation makes no assumptions about table structure or metadata, and when `select()` is executed, a simple ResultSet object with the populated Adapter's Result (the datasource) will be returned and ready for iteration.

```
use Zend\Db\TableGateway\TableGateway;
$projectTable = new TableGateway('project', $adapter);
$rowset = $projectTable->select(array('type' => 'PHP'));

echo 'Projects of type PHP: ';
foreach ($rowset as $projectRow) {
     echo $projectRow['name'] . PHP_EOL;
}

// or, when expecting a single row:
$artistTable = new TableGateway('artist', $adapter);
$rowset = $artistTable->select(array('id' => 2));
$artistRow = $rowset->current();

var_dump($artistRow);
```

The `select()` method takes the same arguments as `Zend\Db\Sql\Select::where()` with the addition of also being able to accept a closure, which in turn, will be passed the current Select object that is being used to build the SELECT query. The following usage is possible:

```
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\Sql\Select;
$artistTable = new TableGateway('artist', $adapter);

// search for at most 2 artists who's name starts with Brit, ascending
$rowset = $artistTable->select(function (Select $select) {
     $select->where->like('name', 'Brit%');
     $select->order('name ASC')->limit(2);
});
```

68.2 TableGateway Features

The Features API allows for extending the functionality of the base `TableGateway` object without having to polymorphically extend the base class. This allows for a wider array of possible mixing and matching of features to achieve a particular behavior that needs to be attained to make the base implementation of `TableGateway` useful for a particular problem.

With the `TableGateway` object, features should be injected though the constructor. The constructor can take Features in 3 different forms: as a single feature object, as a FeatureSet object, or as an array of Feature objects.

There are a number of features built-in and shipped with Zend\Db:

- GlobalAdapterFeature: the ability to use a global/static adapter without needing to inject it into a `TableGateway` instance. This is more useful when you are extending the `AbstractTableGateway` implementation:

```
use Zend\Db\TableGateway\AbstractTableGateway;
use Zend\Db\TableGateway\Feature;

class MyTableGateway extends AbstractTableGateway
{
   public function __construct()
   {
      $this->table = 'my_table';
             $this->featureSet = new Feature\FeatureSet();
             $this->featureSet->addFeature(new Feature\GlobalAdapterFeature());
             $this->initialize();
   }
}

// elsewhere in code, in a bootstrap
Zend\Db\TableGateway\Feature\GlobalAdapterFeature::setStaticAdapter($adapter);

// in a controller, or model somewhere
$table = new MyTableGateway(); // adapter is statically loaded
```

- MasterSlaveFeature: the ability to use a master adapter for insert(), update(), and delete() while using a slave adapter for all select() operations.

```
$table = new TableGateway('artist', $adapter, new Feature\MasterSlaveFeature($slaveAdapter));
```

- MetadataFeature: the ability populate `TableGateway` with column information from a Metadata object. It will also store the primary key information in case RowGatewayFeature needs to consume this information.

```
$table = new TableGateway('artist', $adapter, new Feature\MetadataFeature());
```

- EventFeature: the ability utilize a `TableGateway` object with Zend\EventManager and to be able to subscribe to various events in a `TableGateway` lifecycle.

```
$table = new TableGateway('artist', $adapter, new Feature\EventFeature($eventManagerInstance));
```

- RowGatewayFeature: the ability for `select()` to return a ResultSet object that upon iteration will return a `RowGateway` object for each row.

```
$table = new TableGateway('artist', $adapter, new Feature\RowGatewayFeature('id'));
$results = $table->select(array('id' => 2));

$artistRow = $results->current();
```

```
5 $artistRow->name = 'New Name';
6 $artistRow->save();
```

CHAPTER

SIXTYNINE

ZEND\DB\ROWGATEWAY

`Zend\Db\RowGateway` is a sub-component of Zend\Db that implements the Row Gateway pattern from PoEAA. This effectively means that Row Gateway objects primarily model a row in a database, and have methods such as save() and delete() that will help persist this row-as-an-object in the database itself. Likewise, after a row from the database is retrieved, it can then be manipulated and save()'d back to the database in the same position (row), or it can be delete()'d from the table.

The interface for a Row Gateway object simply adds save() and delete() and this is the interface that should be assumed when a component has a dependency that is expected to be an instance of a RowGateway object:

```
interface RowGatewayInterface
{
    public function save();
    public function delete();
}
```

69.1 Quickstart

While most of the time, RowGateway will be used in conjunction with other Zend\Db\ResultSet producing objects, it is possible to use it standalone. To use it standalone, you simply need an Adapter and a set of data to work with. The following use case demonstrates Zend\Db\RowGateway\RowGateway usage in its simplest form:

```
use Zend\Db\RowGateway\RowGateway;

// query the database
$resultSet = $adapter->query('SELECT * FROM `user` WHERE `id` = ?', array(2));

// get array of data
$rowData = $resultSet->current()->getArrayCopy();

// row gateway
$rowGateway = new RowGateway('id', 'my_table', $adapter);
$rowGateway->populate($rowData);

$rowGateway->first_name = 'New Name';
$rowGateway->save();

// or delete this row:
$rowGateway->delete();
```

The workflow described above is greatly simplified when RowGateway is used in conjunction with the TableGateway feature. What this achieves is a Table Gateway object that when select()'ing from a table, will produce a ResultSet

that is then capable of producing valid Row Gateway objects. Its usage looks like this:

```
use Zend\Db\TableGateway\Feature\RowGatewayFeature;
use Zend\Db\TableGateway\TableGateway;

$table = new TableGateway('artist', $adapter, new RowGatewayFeature('id'));
$results = $table->select(array('id' => 2));

$artistRow = $results->current();
$artistRow->name = 'New Name';
$artistRow->save();
```

69.2 ActiveRecord Style Objects

If you wish to have custom behaviour for your RowGateway objects (essentially making them behave similarly to the ActiveRecord pattern), pass a prototype object implementing the `RowGatewayInterface` to the `RowGatewayFeature` constructor instead of a primary key:

```
use Zend\Db\TableGateway\Feature\RowGatewayFeature;
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\RowGateway\RowGatewayInterface;

class Artist implements RowGatewayInterface
{
    protected $adapter;

    public function __construct($adapter)
    {
       $this->adapter = $adapter;
    }

    // ... save() and delete() implementations
}

$table = new TableGateway('artist', $adapter, new RowGatewayFeature(new Artist($adapter)));
```

CHAPTER

SEVENTY

ZEND\DB\METADATA

`Zend\Db\Metadata` is as sub-component of Zend\Db that makes it possible to get metadata information about tables, columns, constraints, triggers, and other information from a database in a standardized way. The primary interface for the Metadata objects is:

```
interface MetadataInterface
{
    public function getSchemas();

    public function getTableNames($schema = null, $includeViews = false);
    public function getTables($schema = null, $includeViews = false);
    public function getTable($tableName, $schema = null);

    public function getViewNames($schema = null);
    public function getViews($schema = null);
    public function getView($viewName, $schema = null);

    public function getColumnNames($table, $schema = null);
    public function getColumns($table, $schema = null);
    public function getColumn($columnName, $table, $schema = null);

    public function getConstraints($table, $schema = null);
    public function getConstraint($constraintName, $table, $schema = null);
    public function getConstraintKeys($constraint, $table, $schema = null);

    public function getTriggerNames($schema = null);
    public function getTriggers($schema = null);
    public function getTrigger($triggerName, $schema = null);
}
```

70.1 Basic Usage

Usage of `Zend\Db\Metadata` is very straight forward. The top level class Zend\Db\Metadata\Metadata will, given an adapter, choose the best strategy (based on the database platform being used) for retrieving metadata. In most cases, information will come from querying the INFORMATION_SCHEMA tables generally accessible to all database connections about the currently accessible schema.

Metadata::get*Names() methods will return an array of strings, while the other methods will return specific value objects with the containing information. This is best demonstrated by the script below.

```
$metadata = new Zend\Db\Metadata\Metadata($adapter);

```

```
// get the table names
$tableNames = $metadata->getTableNames();

foreach ($tableNames as $tableName) {
    echo 'In Table ' . $tableName . PHP_EOL;

    $table = $metadata->getTable($tableName);


    echo '    With columns: ' . PHP_EOL;
    foreach ($table->getColumns() as $column) {
        echo '        ' . $column->getName()
            . ' -> ' . $column->getDataType()
            . PHP_EOL;
    }

    echo PHP_EOL;
    echo '    With constraints: ' . PHP_EOL;

    foreach ($metadata->getConstraints($tableName) as $constraint) {
        /** @var $constraint Zend\Db\Metadata\Object\ConstraintObject */
        echo '        ' . $constraint->getName()
            . ' -> ' . $constraint->getType()
            . PHP_EOL;
        if (!$constraint->hasColumns()) {
            continue;
        }
        echo '            column: ' . implode(', ', $constraint->getColumns());
        if ($constraint->isForeignKey()) {
            $fkCols = array();
            foreach ($constraint->getReferencedColumns() as $refColumn) {
                $fkCols[] = $constraint->getReferencedTableName() . '.' . $refColumn;
            }
            echo ' => ' . implode(', ', $fkCols);
        }
        echo PHP_EOL;

    }

    echo '----' . PHP_EOL;
}
```

Metadata returns value objects that provide an interface to help developers better explore the metadata. Below is the API for the various value objects:

The TableObject:

```
class Zend\Db\Metadata\Object\TableObject
{
    public function __construct($name);
    public function setColumns(array $columns);
    public function getColumns();
    public function setConstraints($constraints);
    public function getConstraints();
    public function setName($name);
    public function getName();
}
```

The ColumnObject:

```
class Zend\Db\Metadata\Object\ColumnObject {
    public function __construct($name, $tableName, $schemaName = null);
    public function setName($name);
    public function getName();
    public function getTableName();
    public function setTableName($tableName);
    public function setSchemaName($schemaName);
    public function getSchemaName();
    public function getOrdinalPosition();
    public function setOrdinalPosition($ordinalPosition);
    public function getColumnDefault();
    public function setColumnDefault($columnDefault);
    public function getIsNullable();
    public function setIsNullable($isNullable);
    public function isNullable();
    public function getDataType();
    public function setDataType($dataType);
    public function getCharacterMaximumLength();
    public function setCharacterMaximumLength($characterMaximumLength);
    public function getCharacterOctetLength();
    public function setCharacterOctetLength($characterOctetLength);
    public function getNumericPrecision();
    public function setNumericPrecision($numericPrecision);
    public function getNumericScale();
    public function setNumericScale($numericScale);
    public function getNumericUnsigned();
    public function setNumericUnsigned($numericUnsigned);
    public function isNumericUnsigned();
    public function getErratas();
    public function setErratas(array $erratas);
    public function getErrata($errataName);
    public function setErrata($errataName, $errataValue);
}
```

The ConstraintObject:

```
class Zend\Db\Metadata\Object\ConstraintObject
{
    public function __construct($name, $tableName, $schemaName = null);
    public function setName($name);
    public function getName();
    public function setSchemaName($schemaName);
    public function getSchemaName();
    public function getTableName();
    public function setTableName($tableName);
    public function setType($type);
    public function getType();
    public function hasColumns();
    public function getColumns();
    public function setColumns(array $columns);
    public function getReferencedTableSchema();
    public function setReferencedTableSchema($referencedTableSchema);
    public function getReferencedTableName();
    public function setReferencedTableName($referencedTableName);
    public function getReferencedColumns();
    public function setReferencedColumns(array $referencedColumns);
    public function getMatchOption();
    public function setMatchOption($matchOption);
```

```
    public function getUpdateRule();
    public function setUpdateRule($updateRule);
    public function getDeleteRule();
    public function setDeleteRule($deleteRule);
    public function getCheckClause();
    public function setCheckClause($checkClause);
    public function isPrimaryKey();
    public function isUnique();
    public function isForeignKey();
    public function isCheck();

}
```

The TriggerObject:

```
class Zend\Db\Metadata\Object\TriggerObject
{
    public function getName();
    public function setName($name);
    public function getEventManipulation();
    public function setEventManipulation($eventManipulation);
    public function getEventObjectCatalog();
    public function setEventObjectCatalog($eventObjectCatalog);
    public function getEventObjectSchema();
    public function setEventObjectSchema($eventObjectSchema);
    public function getEventObjectTable();
    public function setEventObjectTable($eventObjectTable);
    public function getActionOrder();
    public function setActionOrder($actionOrder);
    public function getActionCondition();
    public function setActionCondition($actionCondition);
    public function getActionStatement();
    public function setActionStatement($actionStatement);
    public function getActionOrientation();
    public function setActionOrientation($actionOrientation);
    public function getActionTiming();
    public function setActionTiming($actionTiming);
    public function getActionReferenceOldTable();
    public function setActionReferenceOldTable($actionReferenceOldTable);
    public function getActionReferenceNewTable();
    public function setActionReferenceNewTable($actionReferenceNewTable);
    public function getActionReferenceOldRow();
    public function setActionReferenceOldRow($actionReferenceOldRow);
    public function getActionReferenceNewRow();
    public function setActionReferenceNewRow($actionReferenceNewRow);
    public function getCreated();
    public function setCreated($created);
}
```

CHAPTER

SEVENTYONE

INTRODUCTION TO ZEND\DI

71.1 Dependency Injection

Dependency Injection (here-in called DI) is a concept that has been talked about in numerous places over the web. Simply put, we'll explain the act of injecting dependencies simply with this below code:

```
$b = new MovieLister(new MovieFinder());
```

Above, MovieFinder is a dependency of MovieLister, and MovieFinder was injected into MovieLister. If you are not familiar with the concept of DI, here are a couple of great reads: Matthew Weier O'Phinney's Analogy, Ralph Schindler's Learning DI, or Fabien Potencier's Series on DI.

71.2 Dependency Injection Containers

When your code is written in such a way that all your dependencies are injected into consuming objects, you might find that the simple act of wiring an object has gotten more complex. When this becomes the case, and you find that this wiring is creating more boilerplate code, this makes for an excellent opportunity to utilize a Dependency Injection Container.

In it's simplest form, a Dependency Injection Container (here-in called a DiC for brevity), is an object that is capable of creating objects on request and managing the "wiring", or the injection of required dependencies, for those requested objects. Since the patterns that developers employ in writing DI capable code vary, DiC's are generally either in the form of smallish objects that suit a very specific pattern, or larger DiC frameworks.

Zend\Di is a DiC framework. While for the simplest code there is no configuration needed, and the use cases are quite simple; for more complex code, Zend\Di is capable of being configured to wire these complex use cases

CHAPTER

SEVENTYTWO

ZEND\DI QUICKSTART

This QuickStart is intended to get developers familiar with the concepts of the Zend\Di DiC. Generally speaking, code is never as simple as it is inside this example, so working knowledge of the other sections of the manual is suggested.

Assume for a moment, you have the following code as part of your application that you feel is a good candidate for being managed by a DiC, after all, you are already injecting all your dependencies:

```
namespace MyLibrary
{
    class DbAdapter
    {
        protected $username = null;
        protected $password = null;
        public function __construct($username, $password)
        {
            $this->username = $username;
            $this->password = $password;
        }
    }
}

namespace MyMovieApp
{
    class MovieFinder
    {
        protected $dbAdapter = null;
        public function __construct(\MyLibrary\DbAdapter $dbAdapter)
        {
            $this->dbAdapter = $dbAdapter;
        }
    }

    class MovieLister
    {
        protected $movieFinder = null;
        public function __construct(MovieFinder $movieFinder)
        {
            $this->movieFinder = $movieFinder;
        }
    }
}
```

With the above code, you find yourself writing the following to wire and utilize this code:

```
// $config object is assumed

$dbAdapter = new MyLibrary\DbAdapter($config->username, $config->password);
$movieFinder = new MyMovieApp\MovieFinder($dbAdapter);
$movieLister = new MyMovieApp\MovieLister($movieFinder);
foreach ($movieLister as $movie) {
    // iterate and display $movie
}
```

If you are doing this above wiring in each controller or view that wants to list movies, not only can this become repetitive and boring to write, but also unmaintainable if for example you want to swap out one of these dependencies on a wholesale scale.

Since this example of code already practices good dependency injection, with constructor injection, it is a great candidate for using Zend\Di. The usage is as simple as:

```
    // inside a bootstrap somewhere
    $di = new Zend\Di\Di();
    $di->instanceManager()->setParameters('MyLibrary\DbAdapter', array(
        'username' => $config->username,
        'password' => $config->password
    ));

    // inside each controller
    $movieLister = $di->get('MyMovieApp\MovieLister');
    foreach ($movieLister as $movie) {
        // iterate and display $movie
    }
```

In the above example, we are obtaining a default instance of Zend\Di\Di. By 'default', we mean that Zend\Di\Di is constructed with a DefinitionList seeded with a RuntimeDefinition (uses Reflection) and an empty instance manager and no configuration. Here is the Zend\Di\Di constructor:

```
    public function __construct(DefinitionList $definitions = null, InstanceManager $instanceManager
    {
        $this->definitions = ($definitions) ?: new DefinitionList(new Definition\RuntimeDefinition()
        $this->instanceManager = ($instanceManager) ?: new InstanceManager();

        if ($config) {
            $this->configure($config);
        }
    }
```

This means that when $di->get() is called, it will be consulting the RuntimeDefinition, which uses reflection to understand the structure of the code. Once it knows the structure of the code, it can then know how the dependencies fit together and how to go about wiring your objects for you. Zend\Di\Definition\RuntimeDefinition will utilize the names of the parameters in the methods as the class parameter names. This is how both username and password key are mapped to the first and second parameter, respectively, of the constructor consuming these named parameters.

If you were to want to pass in the username and password at call time, this is achieved by passing them as the second argument of get():

```
    // inside each controller
    $di = new Zend\Di\Di();
    $movieLister = $di->get('MyMovieApp\MovieLister', array(
        'username' => $config->username,
        'password' => $config->password
    ));
    foreach ($movieLister as $movie) {
```

```
8           // iterate and display $movie
9       }
```

It is important to note that when using call time parameters, these parameter names will be applied to any class that accepts a parameter of such name.

By calling $di->get(), this instance of MovieLister will be automatically shared. This means subsequent calls to get() will return the same instance as previous calls. If you wish to have completely new instances of MovieLister, you can utilize $di->newInstance().

CHAPTER

SEVENTYTHREE

ZEND\DI DEFINITION

Definitions are the place where Zend\Di attempts to understand the structure of the code it is attempting to wire. This means that if you've written non-ambiguous, clear and concise code; Zend\Di has a very good chance of understanding how to wire things up without much added complexity.

73.1 DefinitionList

Definitions are introduced to the Zend\Di\Di object through a definition list implemented as Zend\Di\DefinitionList (SplDoublyLinkedList). Order is important. Definitions in the front of the list will be consulted on a class before definitions at the end of the list.

Note: Regardless of what kind of Definition strategy you decide to use, it is important that your autoloaders are already setup and ready to use.

73.2 RuntimeDefinition

The default DefinitionList instantiated by Zend\Di\Di, when no other DefinitionList is provided, has as Definition\RuntimeDefinition baked-in. The RuntimeDefinition will respond to query's about classes by using Reflection. This Runtime definitions uses any available information inside methods: their signature, the names of parameters, the type-hints of the parameters, and the default values to determine if something is optional or required when making a call to that method. The more explicit you can be in your method naming and method signatures, the easier of a time Zend\Di\Definition\RuntimeDefinition will have determining the structure of your code.

This is what the constructor of a RuntimeDefinition looks like:

```
public function __construct(IntrospectionStrategy $introspectionStrategy = null, array $explicitClass
{
    $this->introspectionStrategy = ($introspectionStrategy) ?: new IntrospectionStrategy();
    if ($explicitClasses) {
        $this->setExplicitClasses($explicitClasses);
    }
}
```

The IntrospectionStrategy object is an object that determines the rules, or guidelines, for how the RuntimeDefinition will introspect information about your classes. Here are the things it knows how to do:

- Whether or not to use Annotations (Annotations are expensive and off by default, read more about these in the Annotations section)
- Which method names to include in the introspection, by default, the pattern /^set[A-Z]{1}\w*/ is registered by default, this is a list of patterns.

- Which interface names represent the interface injection pattern. By default, the pattern /\w*Aware\w*/ is registered, this is a list of patterns.

The constructor for the IntrospectionStrategy looks like this:

```
public function __construct(AnnotationManager $annotationManager = null)
{
    $this->annotationManager = ($annotationManager) ?: $this->createDefaultAnnotationManager();
}
```

This goes to say that an AnnotationManager is not required, but if you wish to create a special AnnotationManager with your own annotations, and also wish to extend the RuntimeDefinition to look for these special Annotations, this is the place to do it.

The RuntimeDefinition also can be used to look up either all classes (implicitly, which is default), or explicitly look up for particular pre-defined classes. This is useful when your strategy for inspecting one set of classes might differ from those of another strategy for another set of classes. This can be achieved by using the setExplicitClasses() method or by passing a list of classes as a second argument to the constructor of the RuntimeDefinition.

73.3 CompilerDefinition

The CompilerDefinition is very much similar in nature to the RuntimeDefinition with the exception that it can be seeded with more information for the purposes of "compiling" a definition. This is useful when you do not want to be making all those (sometimes expensive) calls to reflection and the annotation scanning system during the request of your application. By using the compiler, a definition can be created and written to disk to be used during a request, as opposed to the task of scanning the actual code.

For example, let's assume we want to create a script that will create definitions for some of our library code:

```
// in "package name" format
$components = array(
    'My_MovieApp',
    'My_OtherClasses',
);

foreach ($components as $component) {
    $diCompiler = new Zend\Di\Definition\CompilerDefinition;
    $diCompiler->addDirectory('/path/to/classes/' . str_replace('_', '/', $component));

    $diCompiler->compile();
    file_put_contents(
        __DIR__ . '/../data/di/' . $component . '-definition.php',
        '<?php return ' . var_export($diCompiler->toArrayDefinition()->toArray(), true) . ';'
    );
}
```

This will create a couple of files that will return an array of the definition for that class. To utilize this in an application, the following code will suffice:

```
protected function setupDi(Application $app)
{
    $definitionList = new DefinitionList(array(
        new Definition\ArrayDefinition(include __DIR__ . '/path/to/data/di/My_MovieApp-definition.php
        new Definition\ArrayDefinition(include __DIR__ . '/path/to/data/di/My_OtherClasses-definition
        $runtime = new Definition\RuntimeDefinition(),
    ));
    $di = new Di($definitionList, null, new Configuration($this->config->di));
```

```
    $di->instanceManager()->addTypePreference('Zend\Di\LocatorInterface', $di);
    $app->setLocator($di);
}
```

The above code would more than likely go inside your application's or module's bootstrap file. This represents the simplest and most performant way of configuring your DiC for usage.

73.4 ClassDefinition

The idea behind using a ClassDefinition is two-fold. First, you may want to override some information inside of a RuntimeDefinition. Secondly, you might want to simply define your complete class's definition with an xml, ini, or php file describing the structure. This class definition can be fed in via Configuration or by directly instantiating and registering the Definition with the DefinitionList.

Todo - example

CHAPTER

SEVENTYFOUR

ZEND\DI INSTANCEMANAGER

The InstanceManager is responsible for any runtime information associated with the Zend\Di\Di DiC. This means that the information that goes into the instance manager is specific to both how the particular consuming Application's needs and even more specifically to the environment in which the application is running.

74.1 Parameters

Parameters are simply entry points for either dependencies or instance configuration values. A class consist of a set of parameters, each uniquely named. When writing your classes, you should attempt to not use the same parameter name twice in the same class when you expect that that parameters is used for either instance configuration or an object dependency. This leads to an ambiguous parameter, and is a situation best avoided.

Our movie finder example can be further used to explain these concepts:

```
namespace MyLibrary
{
    class DbAdapter
    {
        protected $username = null;
        protected $password = null;
        public function __construct($username, $password)
        {
            $this->username = $username;
            $this->password = $password;
        }
    }
}

namespace MyMovieApp
{
    class MovieFinder
    {
        protected $dbAdapter = null;
        public function __construct(\MyLibrary\DbAdapter $dbAdapter)
        {
            $this->dbAdapter = $dbAdapter;
        }
    }

    class MovieLister
    {
        protected $movieFinder = null;
```

```
        public function __construct(MovieFinder $movieFinder)
        {
            $this->movieFinder = $movieFinder;
        }
    }
}
```

In the above example, the class DbAdapter has 2 parameters: username and password; MovieFinder has one parameter: dbAdapter, and MovieLister has one parameter: movieFinder. Any of these can be utilized for injection of either dependencies or scalar values during instance configuration or during call time.

When looking at the above code, since the dbAdapter parameter and the movieFinder parameter are both type-hinted with concrete types, the DiC can assume that it can fulfill these object tendencies by itself. On the other hand, username and password do not have type-hints and are, more than likely, scalar in nature. Since the DiC cannot reasonably know this information, it must be provided to the instance manager in the form of parameters. Not doing so will force $di->get('MyMovieApp\MovieLister') to throw an exception.

The following ways of using parameters are available:

```
// setting instance configuration into the instance manager
$di->instanceManager()->setParameters('MyLibrary\DbAdapter', array(
    'username' => 'myusername',
    'password' => 'mypassword'
));

// forcing a particular dependency to be used by the instance manager
$di->instanceManager()->setParameters('MyMovieApp\MovieFinder', array(
    'dbAdapter' => new MyLibrary\DbAdapter('myusername', 'mypassword')
));

// passing instance parameters at call time
$movieLister = $di->get('MyMovieApp\MovieLister', array(
    'username' => $config->username,
    'password' => $config->password
));

// passing a specific instance at call time
$movieLister = $di->get('MyMovieApp\MovieLister', array(
    'dbAdapter' => new MyLibrary\DbAdaper('myusername', 'mypassword')
));
```

74.2 Preferences

In some cases, you might be using interfaces as type hints as opposed to concrete types. Lets assume the movie example was modified in the following way:

```
namespace MyMovieApp
{
    interface MovieFinderInterface
    {
        // methods required for this type
    }

    class GenericMovieFinder implements MovieFinderInterface
    {
        protected $dbAdapter = null;
```

```
        public function __construct(\MyLibrary\DbAdapter $dbAdapter)
        {
            $this->dbAdapter = $dbAdapter;
        }
    }

    class MovieLister
    {
        protected $movieFinder = null;
        public function __construct(MovieFinderInterface $movieFinder)
        {
            $this->movieFinder = $movieFinder;
        }
    }
}
```

What you'll notice above is that now the MovieLister type minimally expects that the dependency injected implements the MovieFinderInterface. This allows multiple implementations of this base interface to be used as a dependency, if that is what the consumer decides they want to do. As you can imagine, Zend\Di, by itself would not be able to determine what kind of concrete object to use fulfill this dependency, so this type of 'preference' needs to be made known to the instance manager.

To give this information to the instance manager, see the following code example:

```
$di->instanceManager()->addTypePreference('MyMovieApp\MovieFinderInterface', 'MyMovieApp\GenericMovie
// assuming all instance config for username, password is setup
$di->get('MyMovieApp\MovieLister');
```

74.3 Aliases

In some situations, you'll find that you need to alias an instance. There are two main reasons to do this. First, it creates a simpler, alternative name to use when using the DiC, as opposed to using the full class name. Second, you might find that you need to have the same object type in two separate contexts. This means that when you alias a particular class, you can then attach a specific instance configuration to that alias; as opposed to attaching that configuration to the class name.

To demonstrate both of these points, we'll look at a use case where we'll have two separate DbAdapters, one will be for read-only operations, the other will be for read-write operations:

Note: Aliases can also have parameters registered at alias time

```
// assume the MovieLister example of code from the QuickStart

$im = $di->instanceManager();

// add alias for short naming
$im->addAlias('movielister', 'MyMovieApp\MovieLister');

// add aliases for specific instances
$im->addAlias('dbadapter-readonly', 'MyLibrary\DbAdapter', array(
    'username' => $config->db->readAdapter->username,
    'password' => $config->db->readAdapter->password,
));
$im->addAlias('dbadapter-readwrite', 'MyLibrary\DbAdapter', array(
    'username' => $config->db->readWriteAdapter->username,
    'password' => $config->db->readWriteAdapter->password,
```

```
16 ));
17
18 // set a default type to use, pointing to an alias
19 $im->addTypePreference('MyLibrary\DbAdapter', 'dbadapter-readonly');
20
21 $movieListerRead = $di->get('MyMovieApp\MovieLister');
22 $movieListerReadWrite = $di->get('MyMovieApp\MovieLister', array('dbAdapter' => 'dbadapter-readwrite'
```

CHAPTER

SEVENTYFIVE

ZEND\DI CONFIGURATION

Most of the configuration for both the setup of Definitions as well as the setup of the Instance Manager can be attained by a configuration file. This file will produce an array (typically) and have a particular iterable structure.

The top two keys are 'definition' and 'instance', each specifying values for respectively, definition setup and instance manager setup.

The definition section expects the following information expressed as a PHP array:

```php
$config = array(
    'definition' => array(
        'compiler' => array(/* @todo compiler information */),
        'runtime'  => array(/* @todo runtime information */),
        'class' => array(
            'instantiator' => '', // the name of the instantiator, by default this is __construct
            'supertypes'   => array(), // an array of supertypes the class implements
            'methods'      => array(
                'setSomeParameter' => array( // a method name
                    'parameterName' => array(
                        'name',       // string parameter name
                        'type',       // type or null
                        'is-required' // bool
                    )
                )

            )
        )
    )
);
```

CHAPTER

SEVENTYSIX

ZEND\DI DEBUGGING & COMPLEX USE CASES

76.1 Debugging a DiC

It is possible to dump the information contained within both the Definition and InstanceManager for a Di instance.

The easiest way is to do the following:

```
    Zend\Di\Display\Console::export($di);
```

If you are using a RuntimeDefinition where upon you expect a particular definition to be resolve at the first-call, you can see that information to the console display to force it to read that class:

```
        Zend\Di\Display\Console::export($di, array('A\ClassIWantTo\GetTheDefinitionFor'));
```

76.2 Complex Use Cases

76.2.1 Interface Injection

```
namespace Foo\Bar {
    class Baz implements BamAwareInterface {
        public $bam;
        public function setBam(Bam $bam){
            $this->bam = $bam;
        }
    }
    class Bam {
    }
    interface BamAwareInterface
    {
        public function setBam(Bam $bam);
    }
}

namespace {
    include 'zf2bootstrap.php';
    $di = new Zend\Di\Di;
    $baz = $di->get('Foo\Bar\Baz');
}
```

76.2.2 Setter Injection with Class Definition

```
namespace Foo\Bar {
    class Baz {
        public $bam;
        public function setBam(Bam $bam){
            $this->bam = $bam;
        }
    }
    class Bam {
    }
}

namespace {
    $di = new Zend\Di\Di;
    $di->configure(new Zend\Di\Config(array(
        'definition' => array(
            'class' => array(
                'Foo\Bar\Baz' => array(
                    'setBam' => array('required' => true)
                )
            )
        )
    )));
    $baz = $di->get('Foo\Bar\Baz');
}
```

76.2.3 Multiple Injections To A Single Injection Point

```
namespace Application {
    class Page {
        public $blocks;
        public function addBlock(Block $block){
            $this->blocks[] = $block;
        }
    }
    interface Block {
    }
}

namespace MyModule {
    class BlockOne implements \Application\Block {}
    class BlockTwo implements \Application\Block {}
}

namespace {
    include 'zf2bootstrap.php';
    $di = new Zend\Di\Di;
    $di->configure(new Zend\Di\Config(array(
        'definition' => array(
            'class' => array(
                'Application\Page' => array(
                    'addBlock' => array(
                        'block' => array('type' => 'Application\Block', 'required' => true)
                    )
                )
            )
```

```
29          ),
30          'instance' => array(
31              'Application\Page' => array(
32                  'injections' => array(
33                      'MyModule\BlockOne',
34                      'MyModule\BlockTwo'
35                  )
36              )
37          )
38      )));
39      $page = $di->get('Application\Page');
40  }
```

CHAPTER

SEVENTYSEVEN

INTRODUCTION TO ZEND\DOM

The `Zend\Dom` component provides tools for working with *DOM* documents and structures. Currently, we offer `Zend\Dom\Query`, which provides a unified interface for querying *DOM* documents utilizing both XPath and *CSS* selectors.

CHAPTER

SEVENTYEIGHT

ZEND\DOM\QUERY

`Zend\Dom\Query` provides mechanisms for querying *XML* and (X) *HTML* documents utilizing either XPath or *CSS* selectors. It was developed to aid with functional testing of *MVC* applications, but could also be used for rapid development of screen scrapers.

CSS selector notation is provided as a simpler and more familiar notation for web developers to utilize when querying documents with *XML* structures. The notation should be familiar to anybody who has developed Cascading Style Sheets or who utilizes Javascript toolkits that provide functionality for selecting nodes utilizing *CSS* selectors (Prototype's $$() and Dojo's dojo.query were both inspirations for the component).

78.1 Theory of Operation

To use `Zend\Dom\Query`, you instantiate a `Zend\Dom\Query` object, optionally passing a document to query (a string). Once you have a document, you can use either the `query()` or `queryXpath()` methods; each method will return a `Zend\Dom\NodeList` object with any matching nodes.

The primary difference between `Zend\Dom\Query` and using DOMDocument + DOMXPath is the ability to select against *CSS* selectors. You can utilize any of the following, in any combination:

- **element types**: provide an element type to match: 'div', 'a', 'span', 'h2', etc.
- **style attributes**: *CSS* style attributes to match: '`.error`', '`div.error`', '`label.required`', etc. If an element defines more than one style, this will match as long as the named style is present anywhere in the style declaration.
- **id attributes**: element ID attributes to match: '#content', 'div#nav', etc.
- **arbitrary attributes**: arbitrary element attributes to match. Three different types of matching are provided:
  - **exact match**: the attribute exactly matches the string: 'div[bar="baz"]' would match a div element with a "bar" attribute that exactly matches the value "baz".
  - **word match**: the attribute contains a word matching the string: 'div[bar~="baz"]' would match a div element with a "bar" attribute that contains the word "baz". ' ' would match, but ' ' would not.
  - **substring match**: the attribute contains the string: 'div[bar*="baz"]' would match a div element with a "bar" attribute that contains the string "baz" anywhere within it.
- **direct descendents**: utilize '>' between selectors to denote direct descendents. 'div > span' would select only 'span' elements that are direct descendents of a 'div'. Can also be used with any of the selectors above.
- **descendents**: string together multiple selectors to indicate a hierarchy along which to search. '`div .foo span #one`' would select an element of id 'one' that is a descendent of arbitrary depth beneath a 'span' element, which is in turn a descendent of arbitrary depth beneath an element with a class of 'foo', that is an

descendent of arbitrary depth beneath a 'div' element. For example, it would match the link to the word 'One' in the listing below:

```
<div>
<table>
    <tr>
        <td class="foo">
            <div>
                Lorem ipsum <span class="bar">
                    <a href="/foo/bar" id="one">One</a>
                    <a href="/foo/baz" id="two">Two</a>
                    <a href="/foo/bat" id="three">Three</a>
                    <a href="/foo/bla" id="four">Four</a>
                </span>
            </div>
        </td>
    </tr>
</table>
</div>
```

Once you've performed your query, you can then work with the result object to determine information about the nodes, as well as to pull them and/or their content directly for examination and manipulation. `Zend\Dom\NodeList` implements `Countable` and `Iterator`, and stores the results internally as a DOMDocument and DOMNodeList. As an example, consider the following call, that selects against the *HTML* above:

```
use Zend\Dom\Query;

$dom = new Query($html);
$results = $dom->execute('.foo .bar a');

$count = count($results); // get number of matches: 4
foreach ($results as $result) {
    // $result is a DOMElement
}
```

`Zend\Dom\Query` also allows straight XPath queries utilizing the `queryXpath()` method; you can pass any valid XPath query to this method, and it will return a `Zend\Dom\NodeList` object.

78.2 Methods Available

The `Zend\Dom\Query` family of classes have the following methods available.

78.2.1 Zend\Dom\Query

The following methods are available to `Zend\Dom\Query`:

- `setDocumentXml($document, $encoding = null)`: specify an *XML* string to query against.
- `setDocumentXhtml($document, $encoding = null)`: specify an *XHTML* string to query against.
- `setDocumentHtml($document, $encoding = null)`: specify an *HTML* string to query against.
- `setDocument($document, $encoding = null)`: specify a string to query against; `Zend\Dom\Query` will then attempt to autodetect the document type.
- `setEncoding($encoding)`: specify an encoding string to use. This encoding will be passed to DOMDocument's constructor if specified.

- `getDocument()`: retrieve the original document string provided to the object.
- `getDocumentType()`: retrieve the document type of the document provided to the object; will be one of the `DOC_XML`, `DOC_XHTML`, or `DOC_HTML` class constants.
- `getEncoding()`: retrieves the specified encoding.
- `execute($query)`: query the document using *CSS* selector notation.
- `queryXpath($xPathQuery)`: query the document using XPath notation.

78.2.2 Zend\Dom\NodeList

As mentioned previously, `Zend\Dom\NodeList` implements both `Iterator` and `Countable`, and as such can be used in a `foreach()` loop as well as with the `count()` function. Additionally, it exposes the following methods:

- `getCssQuery()`: return the *CSS* selector query used to produce the result (if any).
- `getXpathQuery()`: return the XPath query used to produce the result. Internally, `Zend\Dom\Query` converts *CSS* selector queries to XPath, so this value will always be populated.
- `getDocument()`: retrieve the DOMDocument the selection was made against.

CHAPTER

SEVENTYNINE

INTRODUCTION TO ZEND\ESCAPER

The OWASP Top 10 web security risks study lists Cross-Site Scripting (XSS) in second place. PHP's sole functionality against XSS is limited to two functions of which one is commonly misapplied. Thus, the `Zend\Escaper` component was written. It offers developers a way to escape output and defend from XSS and related vulnerabilities by introducing **contextual escaping based on peer-reviewed rules**.

`Zend\Escaper` was written with ease of use in mind, so it can be used completely stand-alone from the rest of the framework, and as such can be installed with Composer.

For easier use of the Escaper component within the framework itself, especially with the `Zend\View` component, a *set of view helpers* is provided.

> **Warning:** The `Zend\Escaper` is a security related component. As such, if you believe you found an issue with this component, we ask that you follow our Security Policy and report security issues accordingly. The Zend Framework team and the contributors thanks you in advance.

79.1 Overview

The `Zend\Escaper` component provides one class, `Zend\Escaper\Escaper` which in turn, provides five methods for escaping output. Which method to use when, depends on the context in which the outputted data is used. It is up to the developer to use the right methods in the right context.

`Zend\Escaper\Escaper` has the following escaping methods available for each context:

- **escapeHtml**: escape a string for the HTML Body context.
- **escapeHtmlAttr**: escape a string for the HTML Attribute context.
- **escapeJs**: escape a string for the Javascript context.
- **escapeCss**: escape a string for the CSS context.
- **escapeUrl**: escape a string for the URI or Parameter contexts.

Usage of each method will be discussed in detail in later chapters.

79.2 What Zend\Escaper is not

`Zend\Escaper` is meant to be used only for escaping data that is to be output, and as such should not be misused for filtering input data. For such tasks, the *Zend\Filter component*, HTMLPurifier or PHP's Filter component should be used.

CHAPTER

EIGHTY

THEORY OF OPERATION

`Zend\Escaper` provides methods for escaping output data, dependent on the context in which the data will be used. Each method is based on peer-reviewed rules and is in compliance with the current OWASP recommendations.

The escaping follows a well known and fixed set of encoding rules for each key HTML context, which are defined by OWASP. These rules cannot be impacted or negated by browser quirks or edge-case HTML parsing unless the browser suffers a catastrophic bug in it's HTML parser or Javascript interpreter - both of these are unlikely.

The contexts in which `Zend\Escaper` should be used are **HTML Body**, **HTML Attribute**, **Javascript**, **CSS** and **URL/URI** contexts.

Every escaper method will take the data to be escaped, make sure it is utf-8 encoded data, or try to convert it to utf-8, do the context-based escaping, encode the escaped data back to it's original encoding and return the data to the caller.

The actual escaping of the data differs between each method, they all have their own set of rules according to which the escaping is done. An example will allow us to clearly demonstrate the difference, and how the same characters are being escaped differently between contexts:

```
$escaper = new Zend\Escaper\Escaper('utf-8');

// &lt;script&gt;alert(&quot;zf2&quot;)&lt;/script&gt;
echo $escaper->escapeHtml('<script>alert("zf2")</script>');
// &lt;script&gt;alert&#x28;&quot;zf2&quot;&#x29;&lt;&#x2F;script&gt;
echo $escaper->escapeHtmlAttr('<script>alert("zf2")</script>');
// \x3Cscript\x3Ealert\x28\x22zf2\x22\x29\x3C\x2Fscript\x3E
echo $escaper->escapeJs('<script>alert("zf2")</script>');
// \3C script\3E alert\28 \22 zf2\22 \29 \3C \2F script\3E
echo $escaper->escapeCss('<script>alert("zf2")</script>');
// %3Cscript%3Ealert%28%22zf2%22%29%3C%2Fscript%3E
echo $escaper->escapeUrl('<script>alert("zf2")</script>');
```

More detailed examples will be given in later chapters.

80.1 The Problem with Inconsistent Functionality

At present, programmers orient towards the following PHP functions for each common HTML context:

- **HTML Body**: htmlspecialchars() or htmlentities()
- **HTML Attribute**: htmlspecialchars() or htmlentities()
- **Javascript**: addslashes() or json_encode()
- **CSS**: n/a

- **URL/URI**: rawurlencode() or urlencode()

In practice, these decisions appear to depend more on what PHP offers, and if it can be interpreted as offering sufficient escaping safety, than it does on what is recommended in reality to defend against XSS. While these functions can prevent some forms of XSS, they do not cover all use cases or risks and are therefore insufficient defenses.

Using htmlspecialchars() in a perfectly valid HTML5 unquoted attribute value, for example, is completely useless since the value can be terminated by a space (among other things) which is never escaped. Thus, in this instance, we have a conflict between a widely used HTML escaper and a modern HTML specification, with no specific function available to cover this use case. While it's tempting to blame users, or the HTML specification authors, escaping just needs to deal with whatever HTML and browsers allow.

Using addslashes(), custom backslash escaping or json_encode() will typically ignore HTML special characters such as ampersands which may be used to inject entities into Javascript. Under the right circumstances, browser will convert these entities into their literal equivalents before interpreting Javascript thus allowing attackers to inject arbitrary code.

Inconsistencies with valid HTML, insecure default parameters, lack of character encoding awareness, and misrepresentations of what functions are capable of by some programmers - these all make escaping in PHP an unnecessarily convoluted quest.

To circumvent the lack of escaping methods in PHP, `Zend\Escaper` addresses the need to apply context-specific escaping in web applications. It implements methods that specifically target XSS and offers programmers a tool to secure their applications without misusing other inadequate methods, or using, most likely incomplete, home-grown solutions.

80.2 Why Contextual Escaping?

To understand why multiple standardised escaping methods are needed, here's a couple of quick points (by no means a complete set!):

80.2.1 HTML escaping of unquoted HTML attribute values still allows XSS

This is probably the best known way to defeat htmlspecialchars() when used on attribute values since any space (or character interpreted as a space - there are a lot) lets you inject new attributes whose content can't be neutralised by HTML escaping. The solution (where this is possible) is additional escaping as defined by the OWASP ESAPI codecs. The point here can be extended further - escaping only works if a programmer or designer know what they're doing. In many contexts, there are additional practices and gotchas that need to be carefully monitored since escaping sometimes needs a little extra help to protect against XSS - even if that means ensuring all attribute values are properly double quoted despite this not being required for valid HTML.

80.2.2 HTML escaping of CSS, Javascript or URIs is often reversed when passed to non-HTML interpreters by the browser

HTML escaping is just that - it's designed to escape a string for HTML (i.e. prevent tag or attribute insertion) but not alter the underlying meaning of the content whether it be Text, Javascript, CSS or URIs. For that purpose a fully HTML escaped version of any other context may still have its unescaped form extracted before it's interpreted or executed. For this reason we need separate escapers for Javascript, CSS and URIs and those writing templates **must** know which escaper to apply to which context. Of course this means you need to be able to identify the correct context before selecting the right escaper!

80.2.3 DOM based XSS requires a defence using at least two levels of different escaping in many cases

DOM based XSS has become increasingly common as Javascript has taken off in popularity for large scale client side coding. A simple example is Javascript defined in a template which inserts a new piece of HTML text into the DOM. If the string is only HTML escaped, it may still contain Javascript that will execute in that context. If the string is only Javascript escaped, it may contain HTML markup (new tags and attributes) which will be injected into the DOM and parsed once the inserting Javascript executes. Damned either way? The solution is to escape twice - first escape the string for HTML (make it safe for DOM insertion), and then for Javascript (make it safe for the current Javascript context). Nested contexts are a common means of bypassing naive escaping habits (e.g. you can inject Javascript into a CSS expression within a HTML Attribute).

80.2.4 PHP has no known anti-XSS escape functions (only those kidnapped from their original purposes)

A simple example, widely used, is when you see `json_encode()` used to escape Javascript, or worse, some kind of mutant `addslashes()` implementation. These were never designed to eliminate XSS yet PHP programmers use them as such. For example, `json_encode()` does not escape the ampersand or semi-colon characters by default. That means you can easily inject HTML entities which could then be decoded before the Javascript is evaluated in a HTML document. This lets you break out of strings, add new JS statements, close tags, etc. In other words, using `json_encode()` is insufficient and naive. The same, arguably, could be said for `htmlspecialchars()` which has its own well known limitations that make a singular reliance on it a questionable practice.

CHAPTER

EIGHTYONE

CONFIGURING ZEND\ESCAPER

`Zend\Escaper\Escaper` has only one configuration option available, and that is the encoding to be used by the Escaper object.

The default encoding is **utf-8**. Other supported encodings are:

- iso-8859-1
- iso-8859-5
- iso-8859-15
- cp866, ibm866, 866
- cp1251, windows-1251
- cp1252, windows-1252
- koi8-r, koi8-ru
- big5, big5-hkscs, 950, gb2312, 936
- shift_jis, sjis, sjis-win, cp932
- eucjp, eucjp-win
- macroman

If an unsupported encoding is passed to `Zend\Escaper\Escaper`, a `Zend\Escaper\Exception\InvalidArgumentException` will be thrown.

CHAPTER

EIGHTYTWO

ESCAPING HTML

Probably the most common escaping happens in the **HTML Body context**. There are very few characters with special meaning in this context, yet it is quite common to escape data incorrectly, namely by setting the wrong flags and character encoding.

For escaping data in the HTML Body context, use `Zend\Escaper\Escaper`'s `escapeHtml` method. Internally it uses PHP's `htmlspecialchars`, and additionally correctly sets the flags and encoding.

```
// outputting this without escaping would be a bad idea!
$input = '<script>alert("zf2")</script>';

$escaper = new Zend\Escaper\Escaper('utf-8');

// somewhere in an HTML template
<div class="user-provided-input">
    <?php
    echo $escaper->escapeHtml($input); // all safe!
    ?>
</div>
```

One thing a developer needs to pay special attention too, is that the encoding in which the document is served to the client, as it **must be the same** as the encoding used for escaping!

82.1 Examples of Bad HTML Escaping

An example of incorrect usage:

```
<?php
$input = '<script>alert("zf2")</script>';
$escaper = new Zend\Escaper\Escaper('utf-8');
?>
<?php header('Content-Type: text/html; charset=ISO-8859-1'); ?>
<!DOCTYPE html>
<html>
<head>
    <title>Encodings set incorrectly!</title>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
</head>
<body>
    <?php
    // Bad! The escaper's and the document's encodings are different!
    echo $escaper->escapeHtml($input);
```

```
    ?>
</body>
```

82.2 Examples of Good HTML Escaping

An example of correct usage:

```
<?php
$input = '<script>alert("zf2")</script>';
$escaper = new Zend\Escaper\Escaper('utf-8');
?>
<?php header('Content-Type: text/html; charset=UTF-8'); ?>
<!DOCTYPE html>
<html>
<head>
    <title>Encodings set correctly!</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
    <?php
    // Good! The escaper's and the document's encodings are same!
    echo $escaper->escapeHtml($input);
    ?>
</body>
```

CHAPTER

EIGHTYTHREE

ESCAPING HTML ATTRIBUTES

Escaping data in the **HTML Attribute context** is most often done incorrectly, if not overlooked completely by developers. Regular *HTML escaping* can be used for escaping HTML attributes, *but* only if the attribute value can be **guaranteed as being properly quoted**! To avoid confusion, we recommend always using the HTML Attribute escaper method in the HTML Attribute context.

To escape data in the HTML Attribute, use `Zend\Escaper\Escaper`'s `escapeHtmlAttr` method. Internally it will convert the data to UTF-8, check for it's validity, and use an extended set of characters to escape that are not covered by `htmlspecialchars` to cover the cases where an attribute might be unquoted or quoted illegally.

83.1 Examples of Bad HTML Attribute Escaping

An example of incorrect HTML attribute escaping:

```
<?php header('Content-Type: text/html; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
' onmouseover='alert(/ZF2!/);
INPUT;
/**
 * NOTE: This is equivalent to using htmlspecialchars($input, ENT_COMPAT)
 */
$output = htmlspecialchars($input);
?>
<html>
<head>
    <title>Single Quoted Attribute</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
    <div>
        <?php
        // the span tag will look like:
        // <span title='' onmouseover='alert(/ZF2!/);'>
        ?>
        <span title='<?php echo $output ?>'>
            What framework are you using?
        </span>
    </div>
</body>
</html>
```

In the above example, the default `ENT_COMPAT` flag is being used, which does not escape single quotes, thus resulting in an alert box popping up when the `onmouseover` event happens on the `span` element.

Another example of incorrect HTML attribute escaping can happen when unquoted attributes are used, which is, by the way, perfectly valid HTML5:

```
<?php header('Content-Type: text/html; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
faketitle onmouseover=alert(/ZF2!/);
INPUT;
// Tough luck using proper flags when the title attribute is unquoted!
$output = htmlspecialchars($input,ENT_QUOTES);
?>
<html>
<head>
    <title>Quoteless Attribute</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
    <div>
        <?php
        // the span tag will look like:
        // <span title=faketitle onmouseover=alert(/ZF2!/);>
        ?>
        <span title=<?php echo $output ?>>
            What framework are you using?
        </span>
    </div>
</body>
</html>
```

The above example shows how it is easy to break out from unquoted attributes in HTML5.

83.2 Examples of Good HTML Attribute Escaping

Both of the previous examples can be avoided by simply using the `escapeHtmlAttr` method:

```
<?php header('Content-Type: text/html; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
faketitle onmouseover=alert(/ZF2!/);
INPUT;
$escaper = new Zend\Escaper\Escaper('utf-8');
$output = $escaper->escapeHtmlAttr($input);
?>
<html>
<head>
    <title>Quoteless Attribute</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
    <div>
        <?php
        // the span tag will look like:
```

```
        // <span title=faketitle&#x20;onmouseover&#x3D;alert&#x28;&#x2F;ZF2&#x21;&#x2F;&#x29;&#x3B;>
        ?>
        <span title=<?php echo $output ?>>
            What framework are you using?
        </span>
    </div>
</body>
</html>
```

In the above example, the malicious input from the attacker becomes completely harmless as we used proper HTML attribute escaping!

CHAPTER

EIGHTYFOUR

ESCAPING JAVASCRIPT

Javascript string literals in HTML are subject to significant restrictions particularly due to the potential for unquoted attributes and any uncertainty as to whether Javascript will be viewed as being CDATA or PCDATA by the browser. To eliminate any possible XSS vulnerabilities, Javascript escaping for HTML extends the escaping rules of both ECMAScript and JSON to include any potentially dangerous character. Very similar to HTML attribute value escaping, this means escaping everything except basic alphanumeric characters and the comma, period and underscore characters as hexadecimal or unicode escapes.

Javascript escaping applies to all literal strings and digits. It is not possible to safely escape other Javascript markup.

To escape data in the **Javascript context**, use `Zend\Escaper\Escaper`'s `escapeJs` method. An extended set of characters are escaped beyond ECMAScript's rules for Javascript literal string escaping in order to prevent misinterpretation of Javascript as HTML leading to the injection of special characters and entities.

84.1 Examples of Bad Javascript Escaping

An example of incorrect Javascript escaping:

```
<?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
bar&quot;; alert(&quot;Meow!&quot;); var xss=&quot;true
INPUT;
$output = json_encode($input);
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Unescaped Entities</title>
    <meta charset="UTF-8"/>
    <script type="text/javascript">
        <?php
        // this will result in
        // var foo = "bar&quot;; alert(&quot;Meow!&quot;); var xss=&quot;true";
        ?>
        var foo = <?php echo $output ?>;
    </script>
</head>
<body>
    <p>json_encode() is not good for escaping javascript!</p>
</body>
</html>
```

The above example will show an alert popup box as soon as the page is loaded, because the data is not properly escaped for the Javascript context.

84.2 Examples of Good Javascript Escaping

By using the `escapeJs` method in the Javascript context, such attacks can be prevented:

```
<?php header('Content-Type: text/html; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
bar&quot;; alert(&quot;Meow!&quot;); var xss=&quot;true
INPUT;
$escaper = new Zend\Escaper\Escaper('utf-8');
$output = $escaper->escapeJs($input);
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Escaped Entities</title>
    <meta charset="UTF-8"/>
    <script type="text/javascript">
        <?php
        // this will look like
        // var foo = bar\x26quot\x3B\x3B\x20alert\x28\x26quot\x3BMeow\x21\x26quot\x3B\x29\x3B\x20var
        ?>
        var foo = <?php echo $output ?>;
    </script>
</head>
<body>
    <p>Zend\Escaper\Escaper::escapeJs() is good for escaping javascript!</p>
</body>
</html>
```

In the above example, the Javascript parser will most likely report a `SyntaxError`, but at least the targeted application remains safe from such attacks.

CHAPTER

EIGHTYFIVE

ESCAPING CASCADING STYLE SHEETS

CSS is similar to *Javascript* for the same reasons. CSS escaping excludes only basic alphanumeric characters and escapes all other characters into valid CSS hexadecimal escapes.

85.1 Examples of Bad CSS Escaping

In most cases developers forget to escape CSS completely:

```
<?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
body {
    background-image: url('http://example.com/foo.jpg?</style><script>alert(1)</script>');
}
INPUT;
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Unescaped CSS</title>
    <meta charset="UTF-8"/>
    <style>
    <?php echo $input; ?>
    </style>
</head>
<body>
    <p>User controlled CSS needs to be properly escaped!</p>
</body>
</html>
```

In the above example, by failing to escape the user provided CSS, an attacker can execute an XSS attack fairly easily.

85.2 Examples of Good CSS Escaping

By using `escapeCss` method in the CSS context, such attacks can be prevented:

```
<?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
body {
    background-image: url('http://example.com/foo.jpg?</style><script>alert(1)</script>');
}
INPUT;
$escaper = new Zend\Escaper\Escaper('utf-8');
$output = $escaper->escapeCss($input);
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Escaped CSS</title>
    <meta charset="UTF-8"/>
    <style>
    <?php
    // output will look something like
    // body\20 \7B \A \20 \20 \20 \20 background\2D image\3A \20 url\28 ...
    echo $output;
    ?>
    </style>
</head>
<body>
    <p>User controlled CSS needs to be properly escaped!</p>
</body>
</html>
```

By properly escaping user controlled CSS, we can prevent XSS attacks in our web applications.

CHAPTER

EIGHTYSIX

ESCAPING URLS

This method is basically an alias for PHP's `rawurlencode()` which has applied RFC 3986 since PHP 5.3. It is included primarily for consistency.

URL escaping applies to data being inserted into a URL and not to the whole URL itself.

86.1 Examples of Bad URL Escaping

XSS attacks are easy if data inserted into URLs is not escaped properly:

```
<?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
" onmouseover="alert('zf2')
INPUT;
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Unescaped URL data</title>
    <meta charset="UTF-8"/>
</head>
<body>
    <a href="http://example.com/?name=<?php echo $input; ?>">Click here!</a>
</body>
</html>
```

86.2 Examples of Good URL Escaping

By properly escaping data in URLs by using `escapeUrl`, we can prevent XSS attacks:

```
<?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
<!DOCTYPE html>
<?php
$input = <<<INPUT
" onmouseover="alert('zf2')
INPUT;
$escaper = new Zend\Escaper\Escaper('utf-8');
$output = $escaper->escapeUrl($input);
?>
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
    <title>Unescaped URL data</title>
    <meta charset="UTF-8"/>
</head>
<body>
    <a href="http://example.com/?name=<?php echo $output; ?>">Click here!</a>
</body>
</html>
```

CHAPTER

EIGHTYSEVEN

THE EVENTMANAGER

87.1 Overview

The `EventManager` is a component designed for the following use cases:

- Implementing simple subject/observer patterns.
- Implementing Aspect-Oriented designs.
- Implementing event-driven architectures.

The basic architecture allows you to attach and detach listeners to named events, both on a per-instance basis as well as via shared collections; trigger events; and interrupt execution of listeners.

87.2 Quick Start

Typically, you will compose an `EventManager` instance in a class.

```
use Zend\EventManager\EventManagerInterface;
use Zend\EventManager\EventManager;
use Zend\EventManager\EventManagerAwareInterface;

class Foo implements EventManagerAwareInterface
{
    protected $events;

    public function setEventManager(EventManagerInterface $events)
    {
        $events->setIdentifiers(array(
            __CLASS__,
            get_called_class(),
        ));
        $this->events = $events;
        return $this;
    }

    public function getEventManager()
    {
        if (null === $this->events) {
            $this->setEventManager(new EventManager());
        }
        return $this->events;
```

```
    }
}
```

The above allows users to access the `EventManager` instance, or reset it with a new instance; if one does not exist, it will be lazily instantiated on-demand.

An `EventManager` is really only interesting if it triggers some events. Basic triggering takes three arguments: the event name, which is usually the current function/method name; the "context", which is usually the current object instance; and the arguments, which are usually the arguments provided to the current function/method.

```
class Foo
{
    // ... assume events definition from above

    public function bar($baz, $bat = null)
    {
        $params = compact('baz', 'bat');
        $this->getEventManager()->trigger(__FUNCTION__, $this, $params);
    }
}
```

In turn, triggering events is only interesting if something is listening for the event. Listeners attach to the `EventManager`, specifying a named event and the callback to notify. The callback receives an `Event` object, which has accessors for retrieving the event name, context, and parameters. Let's add a listener, and trigger the event.

```
use Zend\Log\Factory as LogFactory;

$log = LogFactory($someConfig);
$foo = new Foo();
$foo->getEventManager()->attach('bar', function ($e) use ($log) {
    $event  = $e->getName();
    $target = get_class($e->getTarget());
    $params = json_encode($e->getParams());

    $log->info(sprintf(
        '%s called on %s, using params %s',
        $event,
        $target,
        $params
    ));
});

// Results in log message:
$foo->bar('baz', 'bat');
// reading: bar called on Foo, using params {"baz" : "baz", "bat" : "bat"}"
```

Note that the second argument to `attach()` is any valid callback; an anonymous function is shown in the example in order to keep the example self-contained. However, you could also utilize a valid function name, a functor, a string referencing a static method, or an array callback with a named static method or instance method. Again, any PHP callback is valid.

Sometimes you may want to specify listeners without yet having an object instance of the class composing an `EventManager`. Zend Framework enables this through the concept of a `SharedEventCollection`. Simply put, you can inject individual `EventManager` instances with a well-known `SharedEventCollection`, and the `EventManager` instance will query it for additional listeners. Listeners attach to a `SharedEventCollection` in roughly the same way the do normal event managers; the call to `attach` is identical to the `EventManager`, but expects an additional parameter at the beginning: a named instance. Remember the example of composing an `EventManager`, how we passed it `__CLASS__`? That value, or any strings you provide in an array to the constructor, may be used to identify an instance when using a `SharedEventCollection`. As an example, assuming

we have a `SharedEventManager` instance that we know has been injected in our `EventManager` instances (for instance, via dependency injection), we could change the above example to attach via the shared collection:

```
use Zend\Log\Factory as LogFactory;

// Assume $events is a Zend\EventManager\SharedEventManager instance

$log = LogFactory($someConfig);
$events->attach('Foo', 'bar', function ($e) use ($log) {
    $event  = $e->getName();
    $target = get_class($e->getTarget());
    $params = json_encode($e->getParams());

    $log->info(sprintf(
        '%s called on %s, using params %s',
        $event,
        $target,
        $params
    ));
});

// Later, instantiate Foo:
$foo = new Foo();
$foo->getEventManager()->setSharedManager($events);

// And we can still trigger the above event:
$foo->bar('baz', 'bat');
// results in log message:
// bar called on Foo, using params {"baz" : "baz", "bat" : "bat"}"
```

Note: StaticEventManager

As of 2.0.0beta3, you can use the `StaticEventManager` singleton as a `SharedEventCollection`. As such, you do not need to worry about where and how to get access to the `SharedEventCollection`; it's globally available by simply calling *StaticEventManager::getInstance()*.

Be aware, however, that its usage is deprecated within the framework, and starting with 2.0.0beta4, you will instead configure a `SharedEventManager` instance that will be injected by the framework into individual `EventManager` instances.

The `EventManager` also provides the ability to detach listeners, short-circuit execution of an event either from within a listener or by testing return values of listeners, test and loop through the results returned by listeners, prioritize listeners, and more. Many of these features are detailed in the examples.

87.2.1 Wildcard Listeners

Sometimes you'll want to attach the same listener to many events or to all events of a given instance – or potentially, with a shared event collection, many contexts, and many events. The `EventManager` component allows for this.

Attaching to many events at once

```
$events = new EventManager();
$events->attach(array('these', 'are', 'event', 'names'), $callback);
```

Note that if you specify a priority, that priority will be used for all events specified.

Attaching using the wildcard

```
$events = new EventManager();
$events->attach('*', $callback);
```

Note that if you specify a priority, that priority will be used for this listener for any event triggered.

What the above specifies is that **any** event triggered will result in notification of this particular listener.

Attaching to many events at once via a SharedEventManager

```
$events = new SharedEventManager();
// Attach to many events on the context "foo"
$events->attach('foo', array('these', 'are', 'event', 'names'), $callback);

// Attach to many events on the contexts "foo" and "bar"
$events->attach(array('foo', 'bar'), array('these', 'are', 'event', 'names'), $callback);
```

Note that if you specify a priority, that priority will be used for all events specified.

Attaching to many events at once via a SharedEventManager

```
$events = new SharedEventManager();
// Attach to all events on the context "foo"
$events->attach('foo', '*', $callback);

// Attach to all events on the contexts "foo" and "bar"
$events->attach(array('foo', 'bar'), '*', $callback);
```

Note that if you specify a priority, that priority will be used for all events specified.

The above is specifying that for the contexts "foo" and "bar", the specified listener should be notified for any event they trigger.

87.3 Configuration Options

EventManager Options

identifier A string or array of strings to which the given `EventManager` instance can answer when accessed via a `SharedEventManager`.

event_class The name of an alternate `Event` class to use for representing events passed to listeners.

shared_collections An instance of a `SharedEventCollection` instance to use when triggering events.

87.4 Available Methods

__construct `__construct(null|string|int $identifier)`

Constructs a new `EventManager` instance, using the given identifier, if provided, for purposes of shared collections.

setEventClass `setEventClass(string $class)`

Provide the name of an alternate `Event` class to use when creating events to pass to triggered listeners.

setSharedCollections `setSharedCollections(SharedEventCollection $collections = null)`

An instance of a `SharedEventCollection` instance to use when triggering events.

getSharedCollections `getSharedCollections()`

Returns the currently attached `SharedEventCollection` instance. Returns either a `null` if no collection is attached, or a `SharedEventCollection` instance otherwise.

trigger `trigger(string $event, mixed $target = null, mixed $argv, callback $callback = null)`

Triggers all listeners to a named event. The recommendation is to use the current function/method name for `$event`, appending it with values such as ".pre", ".post", etc. as needed. `$target` should be the current object instance, or the name of the function if not triggering within an object. `$argv` should typically be an associative array or `ArrayAccess` instance; we recommend using the parameters passed to the function/method (`compact()` is often useful here). This method can also take a callback and behave in the same way as `triggerUntil()`.

The method returns an instance of `ResponseCollection`, which may be used to introspect return values of the various listeners, test for short-circuiting, and more.

triggerUntil `triggerUntil(string $event, mixed $target, mixed $argv = null, callback $callback = null)`

Triggers all listeners to a named event, just like *trigger()*, with the addition that it passes the return value from each listener to `$callback`; if `$callback` returns a boolean `true` value, execution of the listeners is interrupted. You can test for this using *$result->stopped()*.

attach `attach(string $event, callback $callback, int $priority)`

Attaches `$callback` to the `EventManager` instance, listening for the event `$event`. If a `$priority` is provided, the listener will be inserted into the internal listener stack using that priority; higher values execute earliest. (Default priority is "1", and negative priorities are allowed.)

The method returns an instance of `Zend\Stdlib\CallbackHandler`; this value can later be passed to `detach()` if desired.

attachAggregate `attachAggregate(string|ListenerAggregate $aggregate)`

If a string is passed for `$aggregate`, instantiates that class. The `$aggregate` is then passed the `EventManager` instance to its `attach()` method so that it may register listeners.

The `ListenerAggregate` instance is returned.

detach `detach(CallbackHandler|ListenerAggregateInterface $listener)`

Scans all listeners, and detaches any that match `$listener` so that they will no longer be triggered.

Returns a boolean `true` if any listeners have been identified and unsubscribed, and a boolean `false` otherwise.

detachAggregate `detachAggregate(ListenerAggregateInterface $aggregate)`

Loops through all listeners of all events to identify listeners that are represented by the aggregate; for all matches, the listeners will be removed.

Returns a boolean `true` if any listeners have been identified and unsubscribed, and a boolean `false` otherwise.

getEvents `getEvents()`

Returns an array of all event names that have listeners attached.

getListeners `getListeners(string $event)`

Returns a `Zend\Stdlib\PriorityQueue` instance of all listeners attached to `$event`.

clearListeners `clearListeners(string $event)`

Removes all listeners attached to `$event`.

prepareArgs `prepareArgs(array $args)`

Creates an `ArrayObject` from the provided `$args`. This can be useful if you want yours listeners to be able to modify arguments such that later listeners or the triggering method can see the changes.

87.5 Examples

Modifying Arguments

Occasionally it can be useful to allow listeners to modify the arguments they receive so that later listeners or the calling method will receive those changed values.

As an example, you might want to pre-filter a date that you know will arrive as a string and convert it to a `DateTime` argument.

To do this, you can pass your arguments to `prepareArgs()`, and pass this new object when triggering an event. You will then pull that value back into your method.

```
class ValueObject
{
    // assume a composed event manager

    function inject(array $values)
    {
        $argv = compact('values');
        $argv = $this->getEventManager()->prepareArgs($argv);
        $this->getEventManager()->trigger(__FUNCTION__, $this, $argv);
        $date = isset($argv['values']['date']) ? $argv['values']['date'] : new DateTime('now');

        // ...
    }
}

$v = new ValueObject();

$v->getEventManager()->attach('inject', function($e) {
    $values = $e->getParam('values');
    if (!$values) {
        return;
    }
    if (!isset($values['date'])) {
        $values['date'] = new \DateTime('now');
        return;
    }
    $values['date'] = new \Datetime($values['date']);
});

$v->inject(array(
    'date' => '2011-08-10 15:30:29',
));
```

Short Circuiting

One common use case for events is to trigger listeners until either one indicates no further processing should be done, or until a return value meets specific criteria. As examples, if an event creates a Response object, it may want execution to stop.

```
$listener = function($e) {
    // do some work

    // Stop propagation and return a response
    $e->stopPropagation(true);
    return $response;
};
```

Alternately, we could do the check from the method triggering the event.

```
class Foo implements DispatchableInterface
{
    // assume composed event manager

    public function dispatch(Request $request, Response $response = null)
    {
        $argv = compact('request', 'response');
        $results = $this->getEventManager()->triggerUntil(__FUNCTION__, $this, $argv, function($v) {
            return ($v instanceof Response);
        });
    }
}
```

Typically, you may want to return a value that stopped execution, or use it some way. Both `trigger()` and `triggerUntil()` return a `ResponseCollection` instance; call its `stopped()` method to test if execution was stopped, and `last()` method to retrieve the return value from the last executed listener:

```
class Foo implements DispatchableInterface
{
    // assume composed event manager

    public function dispatch(Request $request, Response $response = null)
    {
        $argv = compact('request', 'response');
        $results = $this->getEventManager()->triggerUntil(__FUNCTION__, $this, $argv, function($v) {
            return ($v instanceof Response);
        });

        // Test if execution was halted, and return last result:
        if ($results->stopped()) {
            return $results->last();
        }

        // continue...
    }
}
```

Assigning Priority to Listeners

One use case for the `EventManager` is for implementing caching systems. As such, you often want to check the cache early, and save to it late.

The third argument to `attach()` is a priority value. The higher this number, the earlier that listener will execute; the lower it is, the later it executes. The value defaults to 1, and values will trigger in the order registered within a given priority.

So, to implement a caching system, our method will need to trigger an event at method start as well as at method end. At method start, we want an event that will trigger early; at method end, an event should trigger late.

Here is the class in which we want caching:

```
class SomeValueObject
{
    // assume it composes an event manager

    public function get($id)
    {
        $params = compact('id');
        $results = $this->getEventManager()->trigger('get.pre', $this, $params);

        // If an event stopped propagation, return the value
        if ($results->stopped()) {
            return $results->last();
        }

        // do some work...

        $params['__RESULT__'] = $someComputedContent;
        $this->getEventManager()->trigger('get.post', $this, $params);
    }
}
```

Now, let's create a `ListenerAggregateInterface` that can handle caching for us:

```
use Zend\Cache\Cache;
use Zend\EventManager\EventManagerInterface;
use Zend\EventManager\ListenerAggregateInterface;
use Zend\EventManager\EventInterface;

class CacheListener implements ListenerAggregateInterface
{
    protected $cache;

    protected $listeners = array();

    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }

    public function attach(EventManagerInterface $events)
    {
        $this->listeners[] = $events->attach('get.pre', array($this, 'load'), 100);
        $this->listeners[] = $events->attach('get.post', array($this, 'save'), -100);
    }

    public function detach(EventManagerInterface $events)
    {
        foreach ($this->listeners as $index => $listener) {
            if ($events->detach($listener)) {
                unset($this->listeners[$index]);
            }
```

```
        }
    }

    public function load(EventInterface $e)
    {
        $id = get_class($e->getTarget()) . '-' . json_encode($e->getParams());
        if (false !== ($content = $this->cache->load($id))) {
            $e->stopPropagation(true);
            return $content;
        }
    }

    public function save(EventInterface $e)
    {
        $params  = $e->getParams();
        $content = $params['__RESULT__'];
        unset($params['__RESULT__']);

        $id = get_class($e->getTarget()) . '-' . json_encode($params);
        $this->cache->save($content, $id);
    }
}
```

We can then attach the aggregate to an instance.

```
$value         = new SomeValueObject();
$cacheListener = new CacheListener($cache);
$value->getEventManager()->attachAggregate($cacheListener);
```

Now, as we call `get()`, if we have a cached entry, it will be returned immediately; if not, a computed entry will be cached when we complete the method.

CHAPTER

EIGHTYEIGHT

INTRODUCTION TO ZEND\FEED

`Zend\Feed` provides functionality for consuming *RSS* and *Atom* feeds. It provides a natural syntax for accessing elements of feeds, feed attributes, and entry attributes. `Zend\Feed` also has extensive support for modifying feed and entry structure with the same natural syntax, and turning the result back into *XML*. In the future, this modification support could provide support for the Atom Publishing Protocol.

`Zend\Feed` consists of `Zend\Feed\Reader` for reading *RSS* and *Atom* feeds, `Zend\Feed\Writer` for writing *RSS* and *Atom* feeds, and `Zend\Feed\PubSubHubbub` for working with Hub servers. Furthermore, both `Zend\Feed\Reader` and `Zend\Feed\Writer` support extensions which allows for working with additional data in feeds, not covered in the core *API* but used in conjunction with RSS and Atom feeds.

In the example below, we demonstrate a simple use case of retrieving an *RSS* feed and saving relevant portions of the feed data to a simple *PHP* array, which could then be used for printing the data, storing to a database, etc.

Note: Be aware

Many *RSS* feeds have different channel and item properties available. The *RSS* specification provides for many optional properties, so be aware of this when writing code to work with *RSS* data. `Zend\Feed` supports all optional properties of the core *RSS* and *Atom* specifications.

Reading RSS Feed Data with Zend\Feed\Reader

```
// Fetch the latest Slashdot headlines
try {
    $slashdotRss =
        Zend\Feed\Reader\Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
} catch (Zend\Feed\Exception\Reader\RuntimeException $e) {
    // feed import failed
    echo "Exception caught importing feed: {$e->getMessage()}\n";
    exit;
}

// Initialize the channel/feed data array
$channel = array(
    'title'       => $slashdotRss->getTitle(),
    'link'        => $slashdotRss->getLink(),
    'description' => $slashdotRss->getDescription(),
    'items'       => array()
    );

// Loop over each channel item/entry and store relevant data for each
foreach ($slashdotRss as $item) {
```

```
    $channel['items'][] = array(
        'title'       => $item->getTitle(),
        'link'        => $item->getLink(),
        'description' => $item->getDescription()
        );
}
```

Your `$channel` array now contains the basic meta-information for the RSS channel and all items that it contained. The process is identical for *Atom* feeds since `Zend\Feed` features a common denominator API, i.e. all getters and setters are the same regardless of feed format.

CHAPTER

EIGHTYNINE

IMPORTING FEEDS

`Zend\Feed` enables developers to retrieve feeds very easily, by using `Zend\Feader\Reader`. If you know the *URI* of a feed, simply use the `Zend\Feed\Reader\Reader::import()` method:

```
$feed = Zend\Feed\Reader\Reader::import('http://feeds.example.com/feedName');
```

You can also use `Zend\Feed\Reader\Reader` to fetch the contents of a feed from a file or the contents of a *PHP* string variable:

```
// importing a feed from a text file
$feedFromFile = Zend\Feed\Reader\Reader::importFile('feed.xml');

// importing a feed from a PHP string variable
$feedFromPHP = Zend\Feed\Reader\Reader::importString($feedString);
```

In each of the examples above, an object of a class that extends `Zend\Feed\Reader\Feed\AbstractFeed` is returned upon success, depending on the type of the feed. If an *RSS* feed were retrieved via one of the import methods above, then a `Zend\Feed\Reader\Feed\Rss` object would be returned. On the other hand, if an Atom feed were imported, then a `Zend\Feed\Reader\Feed\Atom` object is returned. The import methods will also throw a `Zend\Feed\Exception\Reader\RuntimeException` object upon failure, such as an unreadable or malformed feed.

89.1 Dumping the contents of a feed

To dump the contents of a `Zend\Feed\Reader\Feed\AbstractFeed` instance, you may use the `saveXml()` method.

```
assert($feed instanceof Zend\Feed\Reader\Feed\AbstractFeed);

// dump the feed to standard output
print $feed->saveXml();
```

CHAPTER

NINETY

RETRIEVING FEEDS FROM WEB PAGES

Web pages often contain **<link>** tags that refer to feeds with content relevant to the particular page. `Zend\Feed\Reader\Reader` enables you to retrieve all feeds referenced by a web page with one simple method call:

```
$feedLinks = Zend\Feed\Reader\Reader::findFeedLinks('http://www.example.com/news.html');
```

Here the `findFeedLinks()` method returns a `Zend\Feed\Reader\FeedSet` object, that is in turn, a collection of other `Zend\Feed\Reader\FeedSet` objects, that are referenced by **<link>** tags on the `news.html` web page. `Zend\Feed\Reader\Reader` will throw a `Zend\Feed\Reader\Exception\RuntimeException` upon failure, such as an *HTTP* 404 response code or a malformed feed.

You can examine all feed links located by iterating across the collection:

```
$rssFeed = null;
$feedLinks = Zend\Feed\Reader\Reader::findFeedLinks('http://www.example.com/news.html');
foreach ($feedLinks as $link) {
    if (stripos($link['type'], 'application/rss+xml') !== false) {
        $rssFeed = $link['href'];
        break;
}
```

Each `Zend\Feed\Reader\FeedSet` object will expose the rel, href, type and title properties of detected links for all *RSS*, *Atom* or *RDF* feeds. You can always select the first encountered link of each type by using a shortcut:

```
$rssFeed = null;
$feedLinks = Zend\Feed\Reader\Reader::findFeedLinks('http://www.example.com/news.html');
$firstAtomFeed = $feedLinks->atom;
```

CHAPTER

NINETYONE

CONSUMING AN RSS FEED

Reading an *RSS* feed is as simple as passing the *URL* of the feed to `Zend\Feed\Reader\Reader`'s `import` method.

```
$channel = new Zend\Feed\Reader\Reader::import('http://rss.example.com/channelName');
```

If any errors occur fetching the feed, a `Zend\Feed\Reader\Exception\RuntimeException` will be thrown.

Once you have a feed object, you can access any of the standard *RSS* "channel" properties directly on the object:

```
echo $channel->getTitle();
```

Properties of the channel can be accessed via getter methods, such as `getTitle`, `getAuthor` ...

If channel properties have attributes, the getter method will return a key/value pair, where the key is the attribute name, and the value is the attribute value.

```
$author = $channel->getAuthor();
echo $author['name'];
```

Most commonly you'll want to loop through the feed and do something with its entries. `Zend\Feed\Reader\Feed\Rss` internally converts all entries to a `Zend\Feed\Reader\Entry\Rss`. Entry properties, similarly to channel properties, can be accessed via getter methods, such as `getTitle`, `getDescription` ...

An example of printing all titles of articles in a channel is:

```
foreach ($channel as $item) {
    echo $item->getTitle() . "\n";
}
```

If you are not familiar with *RSS*, here are the standard elements you can expect to be available in an *RSS* channel and in individual *RSS* items (entries).

Required channel elements:

- `title`- The name of the channel
- `link`- The *URL* of the web site corresponding to the channel
- `description`- A sentence or several describing the channel

Common optional channel elements:

- `pubDate`- The publication date of this set of content, in *RFC* 822 date format
- `language`- The language the channel is written in

- `category`- One or more (specified by multiple tags) categories the channel belongs to

RSS **<item>** elements do not have any strictly required elements. However, either `title` or `description` must be present.

Common item elements:

- `title`- The title of the item
- `link`- The *URL* of the item
- `description`- A synopsis of the item
- `author`- The author's email address
- `category`- One more categories that the item belongs to
- `comments`-*URL* of comments relating to this item
- `pubDate`- The date the item was published, in *RFC* 822 date format

In your code you can always test to see if an element is non-empty with:

```
if ($item->getPropname()) {
    // ... proceed.
}
```

Where relevant, `Zend\Feed` supports a number of common RSS extensions including Dublin Core, Atom (inside RSS) and the Content, Slash, Syndication, Syndication/Thread and several other extensions or modules.

For further information, the official *RSS* 2.0 specification is available at: http://blogs.law.harvard.edu/tech/rss

CHAPTER

NINETYTWO

CONSUMING AN ATOM FEED

`Zend\Feed\Reader\Feed\Atom` is used in much the same way as `Zend\Feed\Reader\Feed\Rss`. It provides the same access to feed-level properties and iteration over entries in the feed. The main difference is in the structure of the Atom protocol itself. Atom is a successor to *RSS*; it is a more generalized protocol and it is designed to deal more easily with feeds that provide their full content inside the feed, splitting *RSS*' `description` tag into two elements, `summary` and `content`, for that purpose.

Basic Use of an Atom Feed

Read an Atom feed and print the `title` and `summary` of each entry:

```
$feed = Zend\Feed\Reader\Reader::import('http://atom.example.com/feed/');
echo 'The feed contains ' . $feed->count() . ' entries.' . "\n\n";
foreach ($feed as $entry) {
    echo 'Title: ' . $entry->getTitle() . "\n";
    echo 'Description: ' . $entry->getDescription() . "\n";
    echo 'URL: ' . $entry->getLink() . "\n\n";
}
```

In an Atom feed you can expect to find the following feed properties:

- `title`- The feed's title, same as *RSS*'s channel title
- `id`- Every feed and entry in Atom has a unique identifier
- `link`- Feeds can have multiple links, which are distinguished by a `type` attribute

  The equivalent to *RSS*'s channel link would be `type="text/html"`. if the link is to an alternate version of the same content that's in the feed, it would have a `rel="alternate"` attribute.
- `subtitle`- The feed's description, equivalent to *RSS*' channel description
- `author`- The feed's author, with `name` and `email` sub-tags

Atom entries commonly have the following properties:

- `id`- The entry's unique identifier
- `title`- The entry's title, same as *RSS* item titles
- `link`- A link to another format or an alternate view of this entry

  The link property of an atom entry typically has an `href` attribute.
- `summary`- A summary of this entry's content
- `content`- The full content of the entry; can be skipped if the feed just contains summaries

- `author`- with `name` and `email` sub-tags like feeds have
- `published`- the date the entry was published, in *RFC* 3339 format
- `updated`- the date the entry was last updated, in *RFC* 3339 format

Where relevant, `Zend\Feed` supports a number of common RSS extensions including Dublin Core and the Content, Slash, Syndication, Syndication/Thread and several other extensions in common use on blogs.

For more information on Atom and plenty of resources, see http://www.atomenabled.org/.

CHAPTER

NINETYTHREE

CONSUMING A SINGLE ATOM ENTRY

Single Atom `<entry>` elements are also valid by themselves. Usually the *URL* for an entry is the feed's *URL* followed by `/<entryId>`, such as `http://atom.example.com/feed/1`, using the example *URL* we used above. This pattern may exist for some web services which use *Atom* as a container syntax.

If you read a single entry, you will have a `Zend\Feed\Reader\Entry\Atom` object.

Reading a Single-Entry Atom Feed

```
$entry = Zend\Feed\Reader\Reader::import('http://atom.example.com/feed/1');
echo 'Entry title: ' . $entry->getTitle();
```

CHAPTER

NINETYFOUR

ZEND\FEED AND SECURITY

94.1 Introduction

As with any data coming from a source that is beyond the developer's control, special attention needs to be given to securing, validating and filtering that data. Similar to data input to our application by users, data coming from *RSS* and Atom feeds should also be considered unsafe and potentially dangerous, as it allows the delivery of *HTML* and *xHTML* [1]. Because data validation and filtration is out of `Zend\Feed`'s scope, this task is left for implementation by the developer, by using libraries such as `Zend\Escaper` for escaping and HTMLPurifier for validating and filtering feed data.

Escaping and filtering of potentially insecure data is highly recommended before outputting it anywhere in our application or before storing that data in some storage engine (be it a simple file, a database...).

94.2 Filtering data using HTMLPurifier

Currently the best available library for filtering and validating *(x)HTML* data in PHP is HTMLPurifier and, as such, is the recommended tool for this task. HTMLPurifier works by filtering out all *(x)HTML* from the data, except for the tags and attributes specifically allowed in a whitelist, and by checking and fixing nesting of tags, ensuring a standards-compliant output.

The following examples will show a basic usage of HTMLPurifier, but developers are urged to go through and read HTMLPurifier's documentation.

```
// Setting HTMLPurifier's options
$options = array(
    // Allow only paragraph tags
    // and anchor tags wit the href attribute
    array(
        'HTML.Allowed',
        'p,a[href]'
    ),
    // Format end output with Tidy
    array(
        'Output.TidyFormat',
        true
    ),
    // Assume XHTML 1.0 Strict Doctype
    array(
        'HTML.Doctype',
```

[1] http://tools.ietf.org/html/rfc4287#section-8.1

```
        'XHTML 1.0 Strict'
    ),
    // Disable cache, but see note after the example
    array(
        'Cache.DefinitionImpl',
        null
    )
);

// Configuring HTMLPurifier
$config = HTMLPurifier_Config::createDefault();
foreach ($options as $option) {
    $config->set($option[0], $option[1]);
}

// Creating a HTMLPurifier with it's config
$purifier = new HTMLPurifier($config);

// Fetch the RSS
try {
   $rss = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rss/');
} catch (Zend\Feed\Exception\Reader\RuntimeException $e) {
   // feed import failed
   echo "Exception caught importing feed: {$e->getMessage()}\n";
   exit;
}

// Initialize the channel data array
// See that we're cleaning the description with HTMLPurifier
$channel = array(
   'title'       => $rss->getTitle(),
   'link'        => $rss->getLink(),
   'description' => $purifier->purify($rss->getDescription()),
   'items'       => array()
);

// Loop over each channel item and store relevant data
// See that we're cleaning the descriptions with HTMLPurifier
foreach ($rss as $item) {
   $channel['items'][] = array(
       'title'       => $item->getTitle(),
       'link'        => $item->getLink(),
       'description' => $purifier->purify($item->getDescription())
   );
}
```

Note: HTMLPurifier is using the PHP Tidy extension to clean and repair the final output. If this extension is not available, it will silently fail but its availability has no impact on the library's security.

Note: For the sake of this example, the HTMLPurifier's cache is disabled, but it is recommended to configure caching and use its standalone include file as it can improve the performance of HTMLPurifier substantially.

94.3 Escaping data using Zend\Escaper

To help prevent XSS attacks, Zend Framework has a new component `Zend\Escaper`, which complies to the current OWASP recommendations, and as such, is the recommended tool for escaping HTML tags and attributes, Javascript, CSS and URLs before outputing any potentially insecure data to the users.

```
try {
    $rss = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rss/');
} catch (Zend\Feed\Exception\Reader\RuntimeException $e) {
    // feed import failed
    echo "Exception caught importing feed: {$e->getMessage()}\n";
    exit;
}

// Validate all URIs
$linkValidator = new Zend\Validator\Uri;
$link = null;
if ($linkValidator->isValid($rss->getLink())) {
    $link = $rss->getLink();
}

// Escaper used for escaping data
$escaper = new Zend\Escaper\Escaper('utf-8');

// Initialize the channel data array
$channel = array(
    'title'       => $escaper->escapeHtml($rss->getTitle()),
    'link'        => $escaper->escapeHtml($link),
    'description' => $escaper->escapeHtml($rss->getDescription()),
    'items'       => array()
);

// Loop over each channel item and store relevant data
foreach ($rss as $item) {
    $link = null;
    if ($linkValidator->isValid($rss->getLink())) {
        $link = $item->getLink();
    }
    $channel['items'][] = array(
        'title'       => $escaper->escapeHtml($item->getTitle()),
        'link'        => $escaper->escapeHtml($link),
        'description' => $escaper->escapeHtml($item->getDescription())
    );
}
```

The feed data is now safe for output to HTML templates. You can, of course, skip escaping when simply storing the data persistently but remember to escape it on output later!

Of course, these are just basic examples, and cannot cover all possible scenarios that you, as a developer, can, and most likely will, encounter. Your responsibility is to learn what libraries and tools are at your disposal, and when and how to use them to secure your web applications.

CHAPTER

NINETYFIVE

ZEND\FEED\READER\READER

95.1 Introduction

`Zend\Feed\Reader\Reader` is a component used to consume *RSS* and Atom feeds of any version, including *RDF*/*RSS* 1.0, *RSS* 2.0, Atom 0.3 and Atom 1.0. The *API* for retrieving feed data is deliberately simple since `Zend\Feed\Reader` is capable of searching any feed of any type for the information requested through the *API*. If the typical elements containing this information are not present, it will adapt and fall back on a variety of alternative elements instead. This ability to choose from alternatives removes the need for users to create their own abstraction layer on top of the component to make it useful or have any in-depth knowledge of the underlying standards, current alternatives, and namespaced extensions.

Internally, `Zend\Feed\Reader\Reader` works almost entirely on the basis of making XPath queries against the feed *XML*'s Document Object Model. This singular approach to parsing is consistent and the component offers a plugin system to add to the Feed and Entry level *API* by writing Extensions on a similar basis.

Performance is assisted in three ways. First of all, `Zend\Feed\Reader\Reader` supports caching using `Zend\Cache` to maintain a copy of the original feed *XML*. This allows you to skip network requests for a feed *URI* if the cache is valid. Second, the Feed and Entry level *API* is backed by an internal cache (non-persistent) so repeat *API* calls for the same feed will avoid additional *DOM* or XPath use. Thirdly, importing feeds from a *URI* can take advantage of *HTTP* Conditional `GET` requests which allow servers to issue an empty 304 response when the requested feed has not changed since the last time you requested it. In the final case, an instance of `Zend\Cache` will hold the last received feed along with the ETag and Last-Modified header values sent in the *HTTP* response.

`Zend\Feed\Reader\Reader` is not capable of constructing feeds and delegates this responsibility to `Zend\Feed\Writer\Writer`.

95.2 Importing Feeds

Feeds can be imported from a string, file or an *URI*. Importing from a *URI* can additionally utilise a *HTTP* Conditional `GET` request. If importing fails, an exception will be raised. The end result will be an object of type `Zend\Feed\Reader\Feed\AbstractFeed`, the core implementations of which are `Zend\Feed\Reader\Feed\Rss` and `Zend\Feed\Reader\Feed\Atom`. Both objects support multiple (all existing) versions of these broad feed types.

In the following example, we import an *RDF*/*RSS* 1.0 feed and extract some basic information that can be saved to a database or elsewhere.

```
1 $feed = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rdf/');
2 $data = array(
3     'title'        => $feed->getTitle(),
4     'link'         => $feed->getLink(),
```

```
    'dateModified' => $feed->getDateModified(),
    'description'  => $feed->getDescription(),
    'language'     => $feed->getLanguage(),
    'entries'      => array(),
);

foreach ($feed as $entry) {
    $edata = array(
        'title'        => $entry->getTitle(),
        'description'  => $entry->getDescription(),
        'dateModified' => $entry->getDateModified(),
        'authors'      => $entry->getAuthors(),
        'link'         => $entry->getLink(),
        'content'      => $entry->getContent()
    );
    $data['entries'][] = $edata;
}
```

The example above demonstrates `Zend\Feed\Reader\Reader`'s *API*, and it also demonstrates some of its internal operation. In reality, the *RDF* feed selected does not have any native date or author elements, however it does utilise the Dublin Core 1.1 module which offers namespaced creator and date elements. `Zend\Feed\Reader\Reader` falls back on these and similar options if no relevant native elements exist. If it absolutely cannot find an alternative it will return `NULL`, indicating the information could not be found in the feed. You should note that classes implementing `Zend\Feed\Reader\Feed\AbstractFeed` also implement the *SPL* `Iterator` and `Countable` interfaces.

Feeds can also be imported from strings or files.

```
// from a URI
$feed = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rdf/');

// from a String
$feed = Zend\Feed\Reader\Reader::importString($feedXmlString);

// from a file
$feed = Zend\Feed\Reader\Reader::importFile('./feed.xml');
```

95.3 Retrieving Underlying Feed and Entry Sources

`Zend\Feed\Reader\Reader` does its best not to stick you in a narrow confine. If you need to work on a feed outside of `Zend\Feed\Reader\Reader`, you can extract the base DOMDocument or DOMElement objects from any class, or even an *XML* string containing these. Also provided are methods to extract the current DOMXPath object (with all core and Extension namespaces registered) and the correct prefix used in all XPath queries for the current Feed or Entry. The basic methods to use (on any object) are `saveXml()`, `getDomDocument()`, `getElement()`, `getXpath()` and `getXpathPrefix()`. These will let you break free of `Zend\Feed\Reader` and do whatever else you want.

- `saveXml()` returns an *XML* string containing only the element representing the current object.
- `getDomDocument()` returns the DOMDocument object representing the entire feed (even if called from an Entry object).
- `getElement()` returns the DOMElement of the current object (i.e. the Feed or current Entry).
- `getXpath()` returns the DOMXPath object for the current feed (even if called from an Entry object) with the namespaces of the current feed type and all loaded Extensions pre-registered.

- `getXpathPrefix()` returns the query prefix for the current object (i.e. the Feed or current Entry) which includes the correct XPath query path for that specific Feed or Entry.

Here's an example where a feed might include an *RSS* Extension not supported by `Zend\Feed\Reader\Reader` out of the box. Notably, you could write and register an Extension (covered later) to do this, but that's not always warranted for a quick check. You must register any new namespaces on the DOMXPath object before use unless they are registered by `Zend\Feed\Reader` or an Extension beforehand.

```
$feed        = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rdf/');
$xpathPrefix = $feed->getXpathPrefix();
$xpath       = $feed->getXpath();
$xpath->registerNamespace('admin', 'http://webns.net/mvcb/');
$reportErrorsTo = $xpath->evaluate('string('
                                   . $xpathPrefix
                                   . '/admin:errorReportsTo)');
```

> **Warning:** If you register an already registered namespace with a different prefix name to that used internally by `Zend\Feed\Reader\Reader`, it will break the internal operation of this component.

95.4 Cache Support and Intelligent Requests

95.4.1 Adding Cache Support to ZendFeedReaderReader

`Zend\Feed\Reader\Reader` supports using an instance of `Zend\Cache` to cache feeds (as *XML*) to avoid unnecessary network requests. Adding a cache is as simple here as it is for other Zend Framework components, create and configure your cache and then tell `Zend\Feed\Reader\Reader` to use it! The cache key used is "`Zend\Feed\Reader\`" followed by the *MD5* hash of the feed's *URI*.

```
$cache = Zend\Cache\StorageFactory::adapterFactory('Memory');

Zend\Feed\Reader\Reader::setCache($cache);
```

95.4.2 HTTP Conditional GET Support

The big question often asked when importing a feed frequently, is if it has even changed. With a cache enabled, you can add *HTTP* Conditional `GET` support to your arsenal to answer that question.

Using this method, you can request feeds from *URI*s and include their last known ETag and Last-Modified response header values with the request (using the If-None-Match and If-Modified-Since headers). If the feed on the server remains unchanged, you should receive a 304 response which tells `Zend\Feed\Reader\Reader` to use the cached version. If a full feed is sent in a response with a status code of 200, this means the feed has changed and `Zend\Feed\Reader\Reader` will parse the new version and save it to the cache. It will also cache the new ETag and Last-Modified header values for future use.

These "conditional" requests are not guaranteed to be supported by the server you request a *URI* of, but can be attempted regardless. Most common feed sources like blogs should however have this supported. To enable conditional requests, you will need to provide a cache to `Zend\Feed\Reader\Reader`.

```
$cache = Zend\Cache\StorageFactory::adapterFactory('Memory');

Zend\Feed\Reader\Reader::setCache($cache);
Zend\Feed\Reader\Reader::useHttpConditionalGet();

$feed = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rdf/');
```

In the example above, with *HTTP* Conditional `GET` requests enabled, the response header values for ETag and Last-Modified will be cached along with the feed. For the the cache's lifetime, feeds will only be updated on the cache if a non-304 response is received containing a valid *RSS* or Atom *XML* document.

If you intend on managing request headers from outside `Zend\Feed\Reader\Reader`, you can set the relevant If-None-Matches and If-Modified-Since request headers via the *URI* import method.

```
$lastEtagReceived = '5e6cefe7df5a7e95c8b1ba1a2ccaff3d';
$lastModifiedDateReceived = 'Wed, 08 Jul 2009 13:37:22 GMT';
$feed = Zend\Feed\Reader\Reader::import(
    $uri, $lastEtagReceived, $lastModifiedDateReceived
);
```

95.5 Locating Feed URIs from Websites

These days, many websites are aware that the location of their *XML* feeds is not always obvious. A small *RDF*, *RSS* or Atom graphic helps when the user is reading the page, but what about when a machine visits trying to identify where your feeds are located? To assist in this, websites may point to their feeds using <link> tags in the <head> section of their *HTML*. To take advantage of this, you can use `Zend\Feed\Reader\Reader` to locate these feeds using the static `findFeedLinks()` method.

This method calls any *URI* and searches for the location of *RSS*, *RDF* and Atom feeds assuming the website's *HTML* contains the relevant links. It then returns a value object where you can check for the existence of a *RSS*, *RDF* or Atom feed *URI*.

The returned object is an `ArrayObject` subclass called `Zend\Feed\Reader\FeedSet` so you can cast it to an array, or iterate over it, to access all the detected links. However, as a simple shortcut, you can just grab the first *RSS*, *RDF* or Atom link using its public properties as in the example below. Otherwise, each element of the `ArrayObject` is a simple array with the keys "type" and "uri" where the type is one of "rdf", "rss" or "atom".

```
$links = Zend\Feed\Reader\Reader::findFeedLinks('http://www.planet-php.net');

if (isset($links->rdf)) {
    echo $links->rdf, "\n"; // http://www.planet-php.org/rdf/
}
if (isset($links->rss)) {
    echo $links->rss, "\n"; // http://www.planet-php.org/rss/
}
if (isset($links->atom)) {
    echo $links->atom, "\n"; // http://www.planet-php.org/atom/
}
```

Based on these links, you can then import from whichever source you wish in the usual manner.

This quick method only gives you one link for each feed type, but websites may indicate many links of any type. Perhaps it's a news site with a *RSS* feed for each news category. You can iterate over all links using the ArrayObject's iterator.

```
$links = Zend\Feed\Reader::findFeedLinks('http://www.planet-php.net');

foreach ($links as $link) {
    echo $link['href'], "\n";
}
```

95.6 Attribute Collections

In an attempt to simplify return types, return types from the various feed and entry level methods may include an object of type `Zend\Feed\Reader\Collection\AbstractCollection`. Despite the special class name which I'll explain below, this is just a simple subclass of *SPL*'s `ArrayObject`.

The main purpose here is to allow the presentation of as much data as possible from the requested elements, while still allowing access to the most relevant data as a simple array. This also enforces a standard approach to returning such data which previously may have wandered between arrays and objects.

The new class type acts identically to `ArrayObject` with the sole addition being a new method `getValues()` which returns a simple flat array containing the most relevant information.

A simple example of this is `Zend\Feed\Reader\Reader\FeedInterface::getCategories()`. When used with any *RSS* or Atom feed, this method will return category data as a container object called `Zend\Feed\Reader\Collection\Category`. The container object will contain, per category, three fields of data: term, scheme and label. The "term" is the basic category name, often machine readable (i.e. plays nice with *URI*s). The scheme represents a categorisation scheme (usually a *URI* identifier) also known as a "domain" in *RSS* 2.0. The "label" is a human readable category name which supports *HTML* entities. In *RSS* 2.0, there is no label attribute so it is always set to the same value as the term for convenience.

To access category labels by themselves in a simple value array, you might commit to something like:

```
$feed = Zend\Feed\Reader\Reader::import('http://www.example.com/atom.xml');
$categories = $feed->getCategories();
$labels = array();
foreach ($categories as $cat) {
    $labels[] = $cat['label']
}
```

It's a contrived example, but the point is that the labels are tied up with other information.

However, the container class allows you to access the "most relevant" data as a simple array using the `getValues()` method. The concept of "most relevant" is obviously a judgement call. For categories it means the category labels (not the terms or schemes) while for authors it would be the authors' names (not their email addresses or *URI*s). The simple array is flat (just values) and passed through `array_unique()` to remove duplication.

```
$feed = Zend\Feed\Reader\Reader::import('http://www.example.com/atom.xml');
$categories = $feed->getCategories();
$labels = $categories->getValues();
```

The above example shows how to extract only labels and nothing else thus giving simple access to the category labels without any additional work to extract that data by itself.

95.7 Retrieving Feed Information

Retrieving information from a feed (we'll cover entries and items in the next section though they follow identical principals) uses a clearly defined *API* which is exactly the same regardless of whether the feed in question is *RSS*, *RDF* or Atom. The same goes for sub-versions of these standards and we've tested every single *RSS* and Atom version. While the underlying feed *XML* can differ substantially in terms of the tags and elements they present, they nonetheless are all trying to convey similar information and to reflect this all the differences and wrangling over alternative tags are handled internally by `Zend\Feed\Reader\Reader` presenting you with an identical interface for each. Ideally, you should not have to care whether a feed is *RSS* or Atom so long as you can extract the information you want.

Note: While determining common ground between feed types is itself complex, it should be noted that *RSS* in particular is a constantly disputed "specification". This has its roots in the original *RSS* 2.0 document which contains

ambiguities and does not detail the correct treatment of all elements. As a result, this component rigorously applies the *RSS* 2.0.11 Specification published by the *RSS* Advisory Board and its accompanying *RSS* Best Practices Profile. No other interpretation of *RSS* 2.0 will be supported though exceptions may be allowed where it does not directly prevent the application of the two documents mentioned above.

Of course, we don't live in an ideal world so there may be times the *API* just does not cover what you're looking for. To assist you, `Zend\Feed\Reader\Reader` offers a plugin system which allows you to write Extensions to expand the core *API* and cover any additional data you are trying to extract from feeds. If writing another Extension is too much trouble, you can simply grab the underlying *DOM* or XPath objects and do it by hand in your application. Of course, we really do encourage writing an Extension simply to make it more portable and reusable, and useful Extensions may be proposed to the Framework for formal addition.

Here's a summary of the Core *API* for Feeds. You should note it comprises not only the basic *RSS* and Atom standards, but also accounts for a number of included Extensions bundled with `Zend\Feed\Reader\Reader`. The naming of these Extension sourced methods remain fairly generic - all Extension methods operate at the same level as the Core *API* though we do allow you to retrieve any specific Extension object separately if required.

Table 95.1: Feed Level API Methods

getId()	Returns a unique ID associated with this feed
getTitle()	Returns the title of the feed
getDescription()	Returns the text description of the feed.
getLink()	Returns a URI to the HTML website containing the same or similar information as this feed (i.e. if the feed is from a blog, it should provide the blog's URI where the HTML version of the entries can be read).
getFeedLink()	Returns the URI of this feed, which may be the same as the URI used to import the feed. There are important cases where the feed link may differ because the source URI is being updated and is intended to be removed in the future.
getAuthors()	Returns an object of type ZendFeedReaderCollectionAuthor which is an ArrayObject whose elements are each simple arrays containing any combination of the keys "name", "email" and "uri". Where irrelevant to the source data, some of these keys may be omitted.
getAuthor(integer $index = 0)	Returns either the first author known, or with the optional $index parameter any specific index on the array of Authors as described above (returning NULL if an invalid index).
getDateCreated()	Returns the date on which this feed was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created. The returned date will be a DateTime object.
getDateModified()	Returns the date on which this feed was last modified. The returned date will be a DateTime object.
getLastBuildDate()	Returns the date on which this feed was last built. The returned date will be a DateTime object. This is only supported by RSS - Atom feeds will always return NULL.
getLanguage()	Returns the language of the feed (if defined) or simply the language noted in the XML document.
getGenerator()	Returns the generator of the feed, e.g. the software which generated it. This may differ between RSS and Atom since Atom defines a different notation.
getCopyright()	Returns any copyright notice associated with the feed.
getHubs()	Returns an array of all Hub Server URI endpoints which are advertised by the feed for use with the Pubsubhubbub Protocol, allowing subscriptions to the feed for real-time updates.
getCategories()	Returns a ZendFeedReaderCollectionCategory object containing the details of any categories associated with the overall feed. The supported fields include "term" (the machine readable category name), "scheme" (the categorisation scheme and domain for this category), and "label" (a HTML decoded human readable category name). Where any of the three fields are absent from the field, they are either set to the closest available alternative or, in the case of "scheme", set to NULL.
getImage()	Returns an array containing data relating to any feed image or logo, or NULL if no image found. The resulting array may contain the following keys: uri, link, title, description, height, and width. Atom logos only contain a URI so the remaining metadata is drawn from RSS feeds only.

Given the variety of feeds in the wild, some of these methods will undoubtedly return `NULL` indicating the relevant information couldn't be located. Where possible, `Zend\Feed\Reader\Reader` will fall back on alternative elements during its search. For example, searching an *RSS* feed for a modification date is more complicated than it looks. *RSS* 2.0 feeds should include a `<lastBuildDate>` tag and (or) a `<pubDate>` element. But what if it doesn't, maybe this is an *RSS* 1.0 feed? Perhaps it instead has an `<atom:updated>` element with identical information (Atom may be used to supplement *RSS*'s syntax)? Failing that, we could simply look at the entries, pick the most recent, and use its `<pubDate>` element. Assuming it exists... Many feeds also use Dublin Core 1.0 or 1.1 `<dc:date>` elements for feeds and entries. Or we could find Atom lurking again.

The point is, `Zend\Feed\Reader\Reader` was designed to know this. When you ask for the modification date (or anything else), it will run off and search for all these alternatives until it either gives up and returns `NULL`, or finds an alternative that should have the right answer.

In addition to the above methods, all Feed objects implement methods for retrieving the *DOM* and XPath objects for the current feeds as described earlier. Feed objects also implement the *SPL* Iterator and Countable interfaces. The extended *API* is summarised below.

95.8 Retrieving Entry/Item Information

Retrieving information for specific entries or items (depending on whether you speak Atom or *RSS*) is identical to feed level data. Accessing entries is simply a matter of iterating over a Feed object or using the *SPL* `Iterator` interface Feed objects implement and calling the appropriate method on each.

Table 95.2: Entry Level API Methods

getId()	Returns a unique ID for the current entry.
getTitle()	Returns the title of the current entry.
getDescription()	Returns a description of the current entry.
getLink()	Returns a URI to the HTML version of the current entry.
getPermaLink()	Returns the permanent link to the current entry. In most cases, this is the same as using getLink().
getAuthors()	Returns an object of type ZendFeedReaderCollectionAuthor which is an ArrayObject whose elements are each simple arrays containing any combination of the keys "name", "email" and "uri". Where irrelevant to the source data, some of these keys may be omitted.
getAuthor(integer $index = 0)	Returns either the first author known, or with the optional $index parameter any specific index on the array of Authors as described above (returning NULL if an invalid index).
getDateCreated()	Returns the date on which the current entry was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created.
getDateModified()	Returns the date on which the current entry was last modified
getContent()	Returns the content of the current entry (this has any entities reversed if possible assuming the content type is HTML). The description is returned if a separate content element does not exist.
getEnclosure()	Returns an array containing the value of all attributes from a multi-media <enclosure> element including as array keys: url, length, type. In accordance with the RSS Best Practices Profile of the RSS Advisory Board, no support is offers for multiple enclosures since such support forms no part of the RSS specification.
getCommentCount()	Returns the number of comments made on this entry at the time the feed was last generated
getCommentLink()	Returns a URI pointing to the HTML page where comments can be made on this entry
getComment-FeedLink([string $type = 'atom'|'rss'])	Returns a URI pointing to a feed of the provided type containing all comments for this entry (type defaults to Atom/RSS depending on current feed type).
getCategories()	Returns a ZendFeedReaderCollectionCategory object containing the details of any categories associated with the entry. The supported fields include "term" (the machine readable category name), "scheme" (the categorisation scheme and domain for this category), and "label" (a HTML decoded human readable category name). Where any of the three fields are absent from the field, they are either set to the closest available alternative or, in the case of "scheme", set to NULL.

The extended *API* for entries is identical to that for feeds with the exception of the Iterator methods which are not needed here.

Caution: There is often confusion over the concepts of modified and created dates. In Atom, these are two clearly defined concepts (so knock yourself out) but in *RSS* they are vague. *RSS* 2.0 defines a single **<pubDate>** element which typically refers to the date this entry was published, i.e. a creation date of sorts. This is not always the case, and it may change with updates or not. As a result, if you really want to check whether an entry has changed, don't rely on the results of `getDateModified()`. Instead, consider tracking the *MD5* hash of three other elements concatenated, e.g. using `getTitle()`, `getDescription()` and `getContent()`. If the entry was truly updated, this hash computation will give a different result than previously saved hashes for the same entry. This is obviously content oriented, and will not assist in detecting changes to other relevant elements. Atom feeds should not require such steps.
Further muddying the waters, dates in feeds may follow different standards. Atom and Dublin Core dates should follow *ISO* 8601, and *RSS* dates should follow *RFC* 822 or *RFC* 2822 which is also common. Date methods will throw an exception if `DateTime` cannot load the date string using one of the above standards, or the *PHP* recognised possibilities for *RSS* dates.

Warning: The values returned from these methods are not validated. This means users must perform validation on all retrieved data including the filtering of any *HTML* such as from `getContent()` before it is output from your application. Remember that most feeds come from external sources, and therefore the default assumption should be that they cannot be trusted.

95.9 Extending Feed and Entry APIs

Extending `Zend\Feed\Reader\Reader` allows you to add methods at both the feed and entry level which cover the retrieval of information not already supported by `Zend\Feed\Reader\Reader`. Given the number of *RSS* and Atom extensions that exist, this is a good thing since `Zend\Feed\Reader\Reader` couldn't possibly add everything.

There are two types of Extensions possible, those which retrieve information from elements which are immediate children of the root element (e.g. `<channel>` for *RSS* or `<feed>` for Atom) and those who retrieve information from child elements of an entry (e.g. `<item>` for *RSS* or `<entry>` for Atom). On the filesystem these are grouped as classes within a namespace based on the extension standard's name. For example, internally we have `Zend\Feed\Reader\Extension\DublinCore\Feed` and `Zend\Feed\Reader\Extension\DublinCore\Entry` classes which are two Extensions implementing Dublin Core 1.0 and 1.1 support.

Extensions are loaded into `Zend\Feed\Reader\Reader` using a `Zend\ServiceManager\AbstractPluginManager` implementation, `Zend\Feed\Reader\ExtensionManager`, so its operation will be familiar from other Zend Framework components. `Zend\Feed\Reader\Reader` already bundles a number of these Extensions, however those which are not used internally and registered by default (so called Core Extensions) must be registered to `Zend\Feed\Reader\Reader` before they are used. The bundled Extensions include:

Table 95.3: Core Extensions (pre-registered)

DublinCore (Feed and Entry)	Implements support for Dublin Core Metadata Element Set 1.0 and 1.1
Content (Entry only)	Implements support for Content 1.0
Atom (Feed and Entry)	Implements support for Atom 0.3 and Atom 1.0
Slash	Implements support for the Slash RSS 1.0 module
WellFormedWeb	Implements support for the Well Formed Web CommentAPI 1.0
Thread	Implements support for Atom Threading Extensions as described in RFC 4685
Podcast	Implements support for the Podcast 1.0 DTD from Apple

The Core Extensions are somewhat special since they are extremely common and multi-faceted. For example, we have a Core Extension for Atom. Atom is implemented as an Extension (not just a base class) because it doubles as a valid

RSS module - you can insert Atom elements into *RSS* feeds. I've even seen *RDF* feeds which use a lot of Atom in place of more common Extensions like Dublin Core.

Table 95.4: Non-Core Extensions (must register manually)

Syndication	Implements Syndication 1.0 support for RSS feeds
CreativeCommons	A RSS module that adds an element at the <channel> or <item> level that specifies which Creative Commons license applies.

The additional non-Core Extensions are offered but not registered to `Zend\Feed\Reader\Reader` by default. If you want to use them, you'll need to tell `Zend\Feed\Reader\Reader` to load them in advance of importing a feed. Additional non-Core Extensions will be included in future iterations of the component.

Registering an Extension with `Zend\Feed\Reader\Reader`, so it is loaded and its *API* is available to Feed and Entry objects, is a simple affair using the `Zend\Feed\Reader\ExtensionManager`. Here we register the optional Syndication Extension, and discover that it can be directly called from the Entry level *API* without any effort. Note that Extension names are case sensitive and use camel casing for multiple terms.

```
Zend\Feed\Reader\Reader::registerExtension('Syndication');
$feed = Zend\Feed\Reader\Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
$updatePeriod = $feed->getUpdatePeriod();
```

In the simple example above, we checked how frequently a feed is being updated using the `getUpdatePeriod()` method. Since it's not part of `Zend\Feed\Reader\Reader`'s core *API*, it could only be a method supported by the newly registered Syndication Extension.

As you can also notice, the new methods from Extensions are accessible from the main *API* using *PHP*'s magic methods. As an alternative, you can also directly access any Extension object for a similar result as seen below.

```
Zend\Feed\Reader\Reader::registerExtension('Syndication');
$feed = Zend\Feed\Reader\Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
$syndication = $feed->getExtension('Syndication');
$updatePeriod = $syndication->getUpdatePeriod();
```

95.9.1 Writing ZendFeedReaderReader Extensions

Inevitably, there will be times when the `Zend\Feed\Reader\Reader` *API* is just not capable of getting something you need from a feed or entry. You can use the underlying source objects, like DOMDocument, to get these by hand however there is a more reusable method available by writing Extensions supporting these new queries.

As an example, let's take the case of a purely fictitious corporation named Jungle Books. Jungle Books have been publishing a lot of reviews on books they sell (from external sources and customers), which are distributed as an *RSS* 2.0 feed. Their marketing department realises that web applications using this feed cannot currently figure out exactly what book is being reviewed. To make life easier for everyone, they determine that the geek department needs to extend *RSS* 2.0 to include a new element per entry supplying the *ISBN*-10 or *ISBN*-13 number of the publication the entry concerns. They define the new `<isbn>` element quite simply with a standard name and namespace *URI*:

```
JungleBooks 1.0:
http://example.com/junglebooks/rss/module/1.0/
```

A snippet of *RSS* containing this extension in practice could be something similar to:

```
<?xml version="1.0" encoding="utf-8" ?>
<rss version="2.0"
   xmlns:content="http://purl.org/rss/1.0/modules/content/"
   xmlns:jungle="http://example.com/junglebooks/rss/module/1.0/">
<channel>
    <title>Jungle Books Customer Reviews</title>
```

```
    <link>http://example.com/junglebooks</link>
    <description>Many book reviews!</description>
    <pubDate>Fri, 26 Jun 2009 19:15:10 GMT</pubDate>
    <jungle:dayPopular>
        http://example.com/junglebooks/book/938
    </jungle:dayPopular>
    <item>
        <title>Review Of Flatland: A Romance of Many Dimensions</title>
        <link>http://example.com/junglebooks/review/987</link>
        <author>Confused Physics Student</author>
        <content:encoded>
        A romantic square?!
        </content:encoded>
        <pubDate>Thu, 25 Jun 2009 20:03:28 -0700</pubDate>
        <jungle:isbn>048627263X</jungle:isbn>
    </item>
</channel>
</rss>
```

Implementing this new *ISBN* element as a simple entry level extension would require the following class (using your own class namespace outside of Zend).

```
class My\FeedReader\Extension\JungleBooks\Entry
    extends Zend\Feed\Reader\Extension\AbstractEntry
{
    public function getIsbn()
    {
        if (isset($this->data['isbn'])) {
            return $this->data['isbn'];
        }
        $isbn = $this->xpath->evaluate(
            'string(' . $this->getXpathPrefix() . '/jungle:isbn)'
        );
        if (!$isbn) {
            $isbn = null;
        }
        $this->data['isbn'] = $isbn;
        return $this->data['isbn'];
    }

    protected function registerNamespaces()
    {
        $this->xpath->registerNamespace(
            'jungle', 'http://example.com/junglebooks/rss/module/1.0/'
        );
    }
}
```

This extension is easy enough to follow. It creates a new method `getIsbn()` which runs an XPath query on the current entry to extract the *ISBN* number enclosed by the `<jungle:isbn>` element. It can optionally store this to the internal non-persistent cache (no need to keep querying the *DOM* if it's called again on the same entry). The value is returned to the caller. At the end we have a protected method (it's abstract so it must exist) which registers the Jungle Books namespace for their custom *RSS* module. While we call this an *RSS* module, there's nothing to prevent the same element being used in Atom feeds - and all Extensions which use the prefix provided by `getXpathPrefix()` are actually neutral and work on *RSS* or Atom feeds with no extra code.

Since this Extension is stored outside of Zend Framework, you'll need to register the path prefix for your Extensions so `Zend\Loader\PluginLoader` can find them. After that, it's merely a matter of registering the Extension, if

it's not already loaded, and using it in practice.

```
if (!Zend\Feed\Reader\Reader::isRegistered('JungleBooks')) {
     $extensions = Zend\Feed\Reader\Reader::getExtensionManager();
     $extensions->setInvokableClass('JungleBooksEntry', 'My\FeedReader\Extension\JungleBooks\Entry');
     Zend\Feed\Reader\Reader::registerExtension('JungleBooks');
}
$feed = Zend\Feed\Reader\Reader::import('http://example.com/junglebooks/rss');

// ISBN for whatever book the first entry in the feed was concerned with
$firstIsbn = $feed->current()->getIsbn();
```

Writing a feed level Extension is not much different. The example feed from earlier included an unmentioned `<jungle:dayPopular>` element which Jungle Books have added to their standard to include a link to the day's most popular book (in terms of visitor traffic). Here's an Extension which adds a `getDaysPopularBookLink()` method to the feel level *API*.

```
class My\FeedReader\Extension\JungleBooks\Feed
    extends Zend\Feed\Reader\Extension\AbstractFeed
{
    public function getDaysPopularBookLink()
    {
        if (isset($this->data['dayPopular'])) {
            return $this->data['dayPopular'];
        }
        $dayPopular = $this->xpath->evaluate(
            'string(' . $this->getXpathPrefix() . '/jungle:dayPopular)'
        );
        if (!$dayPopular) {
            $dayPopular = null;
        }
        $this->data['dayPopular'] = $dayPopular;
        return $this->data['dayPopular'];
    }

    protected function registerNamespaces()
    {
        $this->xpath->registerNamespace(
            'jungle', 'http://example.com/junglebooks/rss/module/1.0/'
        );
    }
}
```

Let's repeat the last example using a custom Extension to show the method being used.

```
if (!Zend\Feed\Reader\Reader::isRegistered('JungleBooks')) {
     $extensions = Zend\Feed\Reader\Reader::getExtensionManager();
     $extensions->setInvokableClass('JungleBooksFeed', 'My\FeedReader\Extension\JungleBooks\Feed');
     Zend\Feed\Reader\Reader::registerExtension('JungleBooks');
}
$feed = Zend\Feed\Reader\Reader::import('http://example.com/junglebooks/rss');

// URI to the information page of the day's most popular book with visitors
$daysPopularBookLink = $feed->getDaysPopularBookLink();
```

Going through these examples, you'll note that we don't register feed and entry Extensions separately. Extensions within the same standard may or may not include both a feed and entry class, so `Zend\Feed\Reader\Reader` only requires you to register the overall parent name, e.g. JungleBooks, DublinCore, Slash. Internally, it can check at what level Extensions exist and load them up if found. In our case, we have a full set of Extensions now:

`JungleBooks\Feed` and `JungleBooks\Entry`.

CHAPTER

NINETYSIX

ZEND\FEED\WRITER\WRITER

96.1 Introduction

`Zend\Feed\Writer\Writer` is the sibling component to `Zend\Feed\Reader\Reader` responsible for generating feeds for output. It supports the Atom 1.0 specification (*RFC* 4287) and *RSS* 2.0 as specified by the *RSS* Advisory Board (*RSS* 2.0.11). It does not deviate from these standards. It does, however, offer a simple Extension system which allows for any extension and module for either of these two specifications to be implemented if they are not provided out of the box.

In many ways, `Zend\Feed\Writer\Writer` is the inverse of `Zend\Feed\Reader\Reader`. Where `Zend\Feed\Reader\Reader` focuses on providing an easy to use architecture fronted by getter methods, `Zend\Feed\Writer\Writer` is fronted by similarly named setters or mutators. This ensures the *API* won't pose a learning curve to anyone familiar with `Zend\Feed\Reader\Reader`.

As a result of this design, the rest may even be obvious. Behind the scenes, data set on any `Zend\Feed\Writer\Writer` Data Container object is translated at render time onto a DOMDocument object using the necessary feed elements. For each supported feed type there is both an Atom 1.0 and *RSS* 2.0 renderer. Using a DOMDocument class rather than a templating solution has numerous advantages, the most obvious being the ability to export the DOMDocument for additional processing and relying on *PHP DOM* for correct and valid rendering.

96.2 Architecture

The architecture of `Zend\Feed\Writer\Writer` is very simple. It has two core sets of classes: data containers and renderers.

The containers include the `Zend\Feed\Writer\Feed` and `Zend\Feed\Writer\Entry` classes. The Entry classes can be attached to any Feed class. The sole purpose of these containers is to collect data about the feed to generate using a simple interface of setter methods. These methods perform some data validity testing. For example, it will validate any passed *URI*s, dates, etc. These checks are not tied to any of the feed standards definitions. The container objects also contain methods to allow for fast rendering and export of the final feed, and these can be reused at will.

In addition to the main data container classes, there are two additional Atom 2.0 specific classes. `Zend\Feed\Writer\Source` and `Zend\Feed\Writer\Deleted`. The former implements Atom 2.0 source elements which carry source feed metadata for a specific entry within an aggregate feed (i.e. the current feed is not the entry's original source). The latter implements the Atom Tombstones *RFC* allowing feeds to carry references to entries which have been deleted.

While there are two main data container types, there are four renderers - two matching container renderers per supported feed type. Each renderer accepts a container, and based on its content attempts to generate valid feed markup.

If the renderer is unable to generate valid feed markup, perhaps due to the container missing an obligatory data point, it will report this by throwing an `Exception`. While it is possible to ignore `Exceptions`, this removes the default safeguard of ensuring you have sufficient data set to render a wholly valid feed.

To explain this more clearly, you may construct a set of data containers for a feed where there is a Feed container, into which has been added some Entry containers and a Deleted container. This forms a data hierarchy resembling a normal feed. When rendering is performed, this hierarchy has its pieces passed to relevant renderers and the partial feeds (all DOMDocuments) are then pieced together to create a complete feed. In the case of Source or Deleted (Tomestone) containers, these are rendered only for Atom 2.0 and ignored for *RSS*.

Due to the system being divided between data containers and renderers, it can make Extensions somewhat interesting. A typical Extension offering namespaced feed and entry level elements, must itself reflect the exact same architecture, i.e. offer feed and entry level data containers, and matching renderers. There is, fortunately, no complex integration work required since all Extension classes are simply registered and automatically used by the core classes. We'll meet Extensions in more detail at the end of this section.

96.3 Getting Started

Using `Zend\Feed\Writer\Writer` is as simple as setting data and triggering the renderer. Here is an example to generate a minimal Atom 1.0 feed. As this demonstrates, each feed or entry uses a separate data container.

```
/**
 * Create the parent feed
 */
$feed = new Zend\Feed\Writer\Feed;
$feed->setTitle('Paddy\'s Blog');
$feed->setLink('http://www.example.com');
$feed->setFeedLink('http://www.example.com/atom', 'atom');
$feed->addAuthor(array(
    'name'  => 'Paddy',
    'email' => 'paddy@example.com',
    'uri'   => 'http://www.example.com',
));
$feed->setDateModified(time());
$feed->addHub('http://pubsubhubbub.appspot.com/');

/**
 * Add one or more entries. Note that entries must
 * be manually added once created.
 */
$entry = $feed->createEntry();
$entry->setTitle('All Your Base Are Belong To Us');
$entry->setLink('http://www.example.com/all-your-base-are-belong-to-us');
$entry->addAuthor(array(
    'name'  => 'Paddy',
    'email' => 'paddy@example.com',
    'uri'   => 'http://www.example.com',
));
$entry->setDateModified(time());
$entry->setDateCreated(time());
$entry->setDescription('Exposing the difficultly of porting games to English.');
$entry->setContent(
    'I am not writing the article. The example is long enough as is ;).'
);
$feed->addEntry($entry);

```

```
/**
 * Render the resulting feed to Atom 1.0 and assign to $out.
 * You can substitute "atom" with "rss" to generate an RSS 2.0 feed.
 */
$out = $feed->export('atom');
```

The output rendered should be as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title type="text">Paddy's Blog</title>
    <subtitle type="text">Writing about PC Games since 176 BC.</subtitle>
    <updated>2009-12-14T20:28:18+00:00</updated>
    <generator uri="http://framework.zend.com" version="1.10.0alpha">
        Zend\Feed\Writer
    </generator>
    <link rel="alternate" type="text/html" href="http://www.example.com"/>
    <link rel="self" type="application/atom+xml"
        href="http://www.example.com/atom"/>
    <id>http://www.example.com</id>
    <author>
        <name>Paddy</name>
        <email>paddy@example.com</email>
        <uri>http://www.example.com</uri>
    </author>
    <link rel="hub" href="http://pubsubhubbub.appspot.com/"/>
    <entry>
        <title type="html"><![CDATA[All Your Base Are Belong To
            Us]]></title>
        <summary type="html">
            <![CDATA[Exposing the difficultly of porting games to
                English.]]>
        </summary>
        <published>2009-12-14T20:28:18+00:00</published>
        <updated>2009-12-14T20:28:18+00:00</updated>
        <link rel="alternate" type="text/html"
             href="http://www.example.com/all-your-base-are-belong-to-us"/>
        <id>http://www.example.com/all-your-base-are-belong-to-us</id>
        <author>
            <name>Paddy</name>
            <email>paddy@example.com</email>
            <uri>http://www.example.com</uri>
        </author>
        <content type="html">
            <![CDATA[I am not writing the article.
                     The example is long enough as is ;).]]>
        </content>
    </entry>
</feed>
```

This is a perfectly valid Atom 1.0 example. It should be noted that omitting an obligatory point of data, such as a title, will trigger an `Exception` when rendering as Atom 1.0. This will differ for *RSS* 2.0 since a title may be omitted so long as a description is present. This gives rise to Exceptions that differ between the two standards depending on the renderer in use. By design, `Zend\Feed\Writer\Writer` will not render an invalid feed for either standard unless the end-user deliberately elects to ignore all Exceptions. This built in safeguard was added to ensure users without in-depth knowledge of the relevant specifications have a bit less to worry about.

96.4 Setting Feed Data Points

Before you can render a feed, you must first setup the data necessary for the feed being rendered. This utilises a simple setter style *API* which doubles as an initial method for validating the data being set. By design, the *API* closely matches that for `Zend\Feed\Reader\Reader` to avoid undue confusion and uncertainty.

Note: Users have commented that the lack of a simple array based notation for input data gives rise to lengthy tracts of code. This will be addressed in a future release.

`Zend\Feed\Writer\Writer` offers this *API* via its data container classes `Zend\Feed\Writer\Feed` and `Zend\Feed\Writer\Entry` (not to mention the Atom 2.0 specific and Extension classes). These classes merely store all feed data in a type-agnostic manner, meaning you may reuse any data container with any renderer without requiring additional work. Both classes are also amenable to Extensions, meaning that an Extension may define its own container classes which are registered to the base container classes as extensions, and are checked when any method call triggers the base container's `__call()` method.

Here's a summary of the Core *API* for Feeds. You should note it comprises not only the basic *RSS* and Atom standards, but also accounts for a number of included Extensions bundled with `Zend\Feed\Writer\Writer`. The naming of these Extension sourced methods remain fairly generic - all Extension methods operate at the same level as the Core *API* though we do allow you to retrieve any specific Extension object separately if required.

The Feed Level *API* for data is contained in `Zend\Feed\Writer\Feed`. In addition to the *API* detailed below, the class also implements the `Countable` and `Iterator` interfaces.

Table 96.1: Feed Level API Methods

setId()	Set a unique ID associated with this feed. For Atom 1.0 this is an atom:id element, whereas for RSS 2.0 it is added as a guid element. These are optional so long as a link is added, i.e. the link is set as the ID.
setTitle()	Set the title of the feed.
setDescription()	Set the text description of the feed.
setLink()	Set a URI to the HTML website containing the same or similar information as this feed (i.e. if the feed is from a blog, it should provide the blog's URI where the HTML version of the entries can be read).
setFeedLinks()	Add a link to an XML feed, whether the feed being generated or an alternate URI pointing to the same feed but in a different format. At a minimum, it is recommended to include a link to the feed being generated so it has an identifiable final URI allowing a client to track its location changes without necessitating constant redirects. The parameter is an array of arrays, where each sub-array contains the keys "type" and "uri". The type should be one of "atom", "rss", or "rdf".
addAuthors()	Sets the data for authors. The parameter is an array of arrays where each sub-array may contain the keys "name", "email" and "uri". The "uri" value is only applicable for Atom feeds since RSS contains no facility to show it. For RSS 2.0, rendering will create two elements - an author element containing the email reference with the name in brackets, and a Dublin Core creator element only containing the name.
addAuthor()	Sets the data for a single author following the same array format as described above for a single sub-array.
setDateCreated()	Sets the date on which this feed was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created. The expected parameter may be a UNIX timestamp or a DateTime object.
setDateModified()	Sets the date on which this feed was last modified. The expected parameter may be a UNIX timestamp or a DateTime object.
setLastBuildDate()	Sets the date on which this feed was last build. The expected parameter may be a UNIX timestamp or a DateTime object. This will only be rendered for RSS 2.0 feeds and is automatically rendered as the current date by default when not explicitly set.
setLanguage()	Sets the language of the feed. This will be omitted unless set.
setGenerator()	Allows the setting of a generator. The parameter should be an array containing the keys "name", "version" and "uri". If omitted a default generator will be added referencing ZendFeedWriter, the current Zend Framework version and the Framework's URI.
setCopyright()	Sets a copyright notice associated with the feed.
addHubs()	Accepts an array of Pubsubhubbub Hub Endpoints to be rendered in the feed as Atom links so that PuSH Subscribers may subscribe to your feed. Note that you must implement a Pubsubhubbub Publisher in order for real-time updates to be enabled. A Publisher may be implemented using ZendFeedPubsubhubbubPublisher. The method addHub() allows adding a single hub at a time.
addCategories()	Accepts an array of categories for rendering, where each element is itself an array whose possible keys include "term", "label" and "scheme". The "term" is a typically a category name suitable for inclusion in a URI. The "label" may be a human readable category name supporting special characters (it is HTML encoded during rendering) and is a required key. The "scheme" (called the domain in RSS) is optional but must be a valid URI. The method addCategory() allows adding a single category at a time.
setImage()	Accepts an array of image metadata for an RSS image or Atom logo. Atom 1.0 only requires a URI. RSS 2.0 requires a URI, HTML link, and an image title. RSS 2.0 optionally may send a width, height and image description. The array parameter may contain these using the keys: uri, link, title, description, height and width. The RSS 2.0 HTML link should point to the feed source's HTML page.
createEntry()	Returns a new instance of ZendFeedWriterEntry. This is the Entry level data container. New entries are not automatically assigned to the current feed, so you must explicitly call addEntry() to add the entry for rendering.
addEntry()	Adds an instance of ZendFeedWriterEntry to the current feed container for rendering.
createTombstone()	Returns a new instance of ZendFeedWriterDeleted. This is the Atom 2.0 Tombstone level data container. New entries are not automatically assigned to the current feed, so you must explicitly call addTombstone() to add the deleted entry for rendering.

Note: In addition to these setters, there are also matching getters to retrieve data from the Entry data container. For example, `setImage()` is matched with a `getImage()` method.

96.5 Setting Entry Data Points

Here's a summary of the Core *API* for Entries and Items. You should note it comprises not only the basic *RSS* and Atom standards, but also accounts for a number of included Extensions bundled with `Zend\Feed\Writer\Writer`. The naming of these Extension sourced methods remain fairly generic - all Extension methods operate at the same level as the Core *API* though we do allow you to retrieve any specific Extension object separately if required.

The Entry Level *API* for data is contained in `Zend\Feed\Writer\Entry`.

Table 96.2: Entry Level API Methods

setId()	Set a unique ID associated with this entry. For Atom 1.0 this is an atom:id element, whereas for RSS 2.0 it is added as a guid element. These are optional so long as a link is added, i.e. the link is set as the ID.
setTitle()	Set the title of the entry.
setDescrip-tion()	Set the text description of the entry.
setContent()	Set the content of the entry.
setLink()	Set a URI to the HTML website containing the same or similar information as this entry (i.e. if the feed is from a blog, it should provide the blog article's URI where the HTML version of the entry can be read).
set-FeedLinks()	Add a link to an XML feed, whether the feed being generated or an alternate URI pointing to the same feed but in a different format. At a minimum, it is recommended to include a link to the feed being generated so it has an identifiable final URI allowing a client to track its location changes without necessitating constant redirects. The parameter is an array of arrays, where each sub-array contains the keys "type" and "uri". The type should be one of "atom", "rss", or "rdf". If a type is omitted, it defaults to the type used when rendering the feed.
addAuthors()	Sets the data for authors. The parameter is an array of arrays where each sub-array may contain the keys "name", "email" and "uri". The "uri" value is only applicable for Atom feeds since RSS contains no facility to show it. For RSS 2.0, rendering will create two elements - an author element containing the email reference with the name in brackets, and a Dublin Core creator element only containing the name.
addAuthor()	Sets the data for a single author following the same format as described above for a single sub-array.
setDateCre-ated()	Sets the date on which this feed was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created. The expected parameter may be a UNIX timestamp or a DateTime object. If omitted, the date used will be the current date and time.
setDateModi-fied()	Sets the date on which this feed was last modified. The expected parameter may be a UNIX timestamp or a DateTime object. If omitted, the date used will be the current date and time.
setCopy-right()	Sets a copyright notice associated with the feed.
setCate-gories()	Accepts an array of categories for rendering, where each element is itself an array whose possible keys include "term", "label" and "scheme". The "term" is a typically a category name suitable for inclusion in a URI. The "label" may be a human readable category name supporting special characters (it is encoded during rendering) and is a required key. The "scheme" (called the domain in RSS) is optional but must be a valid URI.
setComment-Count()	Sets the number of comments associated with this entry. Rendering differs between RSS and Atom 2.0 depending on the element or attribute needed.
setCom-mentLink()	Seta a link to a HTML page containing comments associated with this entry.
setComment-FeedLink()	Sets a link to a XML feed containing comments associated with this entry. The parameter is an array containing the keys "uri" and "type", where the type is one of "rdf", "rss" or "atom".
setComment-FeedLinks()	Same as setCommentFeedLink() except it accepts an array of arrays, where each subarray contains the expected parameters of setCommentFeedLink().
setEncoding()	Sets the encoding of entry text. This will default to UTF-8 which is the preferred encoding.

Note: In addition to these setters, there are also matching getters to retrieve data from the Entry data container.

CHAPTER

NINETYSEVEN

ZEND\FEED\PUBSUBHUBBUB

`Zend\Feed\PubSubHubbub` is an implementation of the PubSubHubbub Core 0.2 Specification (Working Draft). It offers implementations of a Pubsubhubbub Publisher and Subscriber suited to Zend Framework and other *PHP* applications.

97.1 What is PubSubHubbub?

Pubsubhubbub is an open, simple web-scale pubsub protocol. A common use case to enable blogs (Publishers) to "push" updates from their *RSS* or Atom feeds (Topics) to end Subscribers. These Subscribers will have subscribed to the blog's *RSS* or Atom feed via a Hub, a central server which is notified of any updates by the Publisher and which then distributes these updates to all Subscribers. Any feed may advertise that it supports one or more Hubs using an Atom namespaced link element with a rel attribute of "hub".

Pubsubhubbub has garnered attention because it is a pubsub protocol which is easy to implement and which operates over *HTTP*. Its philosophy is to replace the traditional model where blog feeds have been polled at regular intervals to detect and retrieve updates. Depending on the frequency of polling, this can take a lot of time to propagate updates to interested parties from planet aggregators to desktop readers. With a pubsub system in place, updates are not simply polled by Subscribers, they are pushed to Subscribers, eliminating any delay. For this reason, Pubsubhubbub forms part of what has been dubbed the real-time web.

The protocol does not exist in isolation. Pubsub systems have been around for a while, such as the familiar Jabber Publish-Subscribe protocol, *XEP-0060*, or the less well known rssCloud (described in 2001). However these have not achieved widespread adoption typically due to either their complexity, poor timing or lack of suitability for web applications. rssCloud, which was recently revived as a response to the appearance of Pubsubhubbub, has also seen its usage increase significantly though it lacks a formal specification and currently does not support Atom 1.0 feeds.

Perhaps surprisingly given its relative early age, Pubsubhubbub is already in use including in Google Reader, Feedburner, and there are plugins available for Wordpress blogs.

97.2 Architecture

`Zend\Feed\PubSubHubbub` implements two sides of the Pubsubhubbub 0.2 Specification: a Publisher and a Subscriber. It does not currently implement a Hub Server though this is in progress for a future Zend Framework release.

A Publisher is responsible for notifying all supported Hubs (many can be supported to add redundancy to the system) of any updates to its feeds, whether they be Atom or *RSS* based. This is achieved by pinging the supported Hub Servers with the *URL* of the updated feed. In Pubsubhubbub terminology, any updatable resource capable of being subscribed to is referred to as a Topic. Once a ping is received, the Hub will request the updated feed, process it for updated items, and forward all updates to all Subscribers subscribed to that feed.

A Subscriber is any party or application which subscribes to one or more Hubs to receive updates from a Topic hosted by a Publisher. The Subscriber never directly communicates with the Publisher since the Hub acts as an intermediary, accepting subscriptions and sending updates to subscribed Subscribers. The Subscriber therefore communicates only with the Hub, either to subscribe or unsubscribe to Topics, or when it receives updates from the Hub. This communication design ("Fat Pings") effectively removes the possibility of a "Thundering Herd" issue. This occurs in a pubsub system where the Hub merely informs Subscribers that an update is available, prompting all Subscribers to immediately retrieve the feed from the Publisher giving rise to a traffic spike. In Pubsubhubbub, the Hub distributes the actual update in a "Fat Ping" so the Publisher is not subjected to any traffic spike.

`Zend\Feed\PubSubHubbub` implements Pubsubhubbub Publishers and Subscribers with the classes `Zend\Feed\PubSubHubbub\Publisher` and `Zend\Feed\PubSubHubbub\Subscriber`. In addition, the Subscriber implementation may handle any feed updates forwarded from a Hub by using `Zend\Feed\PubSubHubbub\Subscriber\Callback`. These classes, their use cases, and *API*s are covered in subsequent sections.

97.3 Zend\Feed\PubSubHubbub\Publisher

In Pubsubhubbub, the Publisher is the party who publishes a live feed and frequently updates it with new content. This may be a blog, an aggregator, or even a web service with a public feed based *API*. In order for these updates to be pushed to Subscribers, the Publisher must notify all of its supported Hubs that an update has occurred using a simple *HTTP POST* request containing the *URI* or the updated Topic (i.e the updated *RSS* or Atom feed). The Hub will confirm receipt of the notification, fetch the updated feed, and forward any updates to any Subscribers who have subscribed to that Hub for updates from the relevant feed.

By design, this means the Publisher has very little to do except send these Hub pings whenever its feeds change. As a result, the Publisher implementation is extremely simple to use and requires very little work to setup and use when feeds are updated.

`Zend\Feed\PubSubHubbub\Publisher` implements a full Pubsubhubbub Publisher. Its setup for use is also simple, requiring mainly that it is configured with the *URI* endpoint for all Hubs to be notified of updates, and the *URI*s of all Topics to be included in the notifications.

The following example shows a Publisher notifying a collection of Hubs about updates to a pair of local *RSS* and Atom feeds. The class retains a collection of errors which include the Hub *URL*s, so the notification can be re-attempted later and/or logged if any notifications happen to fail. Each resulting error array also includes a "response" key containing the related *HTTP* response object. In the event of any errors, it is strongly recommended to attempt the operation for failed Hub Endpoints at least once more at a future time. This may require the use of either a scheduled task for this purpose or a job queue though such extra steps are optional.

```
$publisher = new Zend\Feed\PubSubHubbub\Publisher;
$publisher->addHubUrls(array(
    'http://pubsubhubbub.appspot.com/',
    'http://hubbub.example.com',
));
$publisher->addUpdatedTopicUrls(array(
    'http://www.example.net/rss',
    'http://www.example.net/atom',
));
$publisher->notifyAll();

if (!$publisher->isSuccess()) {
    // check for errors
    $errors     = $publisher->getErrors();
    $failedHubs = array();
    foreach ($errors as $error) {
        $failedHubs[] = $error['hubUrl'];
```

```
18        }
19    }
20
21    // reschedule notifications for the failed Hubs in $failedHubs
```

If you prefer having more concrete control over the Publisher, the methods `addHubUrls()` and `addUpdatedTopicUrls()` pass each array value to the singular `addHubUrl()` and `addUpdatedTopicUrl()` public methods. There are also matching `removeUpdatedTopicUrl()` and `removeHubUrl()` methods.

You can also skip setting Hub *URI*s, and notify each in turn using the `notifyHub()` method which accepts the *URI* of a Hub endpoint as its only argument.

There are no other tasks to cover. The Publisher implementation is very simple since most of the feed processing and distribution is handled by the selected Hubs. It is however important to detect errors and reschedule notifications as soon as possible (with a reasonable maximum number of retries) to ensure notifications reach all Subscribers. In many cases as a final alternative, Hubs may frequently poll your feeds to offer some additional tolerance for failures both in terms of their own temporary downtime or Publisher errors or downtime.

97.4 Zend\Feed\PubSubHubbub\Subscriber

In Pubsubhubbub, the Subscriber is the party who wishes to receive updates to any Topic (*RSS* or Atom feed). They achieve this by subscribing to one or more of the Hubs advertised by that Topic, usually as a set of one or more Atom 1.0 links with a rel attribute of "hub". The Hub from that point forward will send an Atom or *RSS* feed containing all updates to that Subscriber's Callback *URL* when it receives an update notification from the Publisher. In this way, the Subscriber need never actually visit the original feed (though it's still recommended at some level to ensure updates are retrieved if ever a Hub goes offline). All subscription requests must contain the *URI* of the Topic being subscribed and a Callback *URL* which the Hub will use to confirm the subscription and to forward updates.

The Subscriber therefore has two roles. To create and manage subscriptions, including subscribing for new Topics with a Hub, unsubscribing (if necessary), and periodically renewing subscriptions since they may have a limited validity as set by the Hub. This is handled by `Zend\Feed\PubSubHubbub\Subscriber`.

The second role is to accept updates sent by a Hub to the Subscriber's Callback *URL*, i.e. the *URI* the Subscriber has assigned to handle updates. The Callback *URL* also handles events where the Hub contacts the Subscriber to confirm all subscriptions and unsubscriptions. This is handled by using an instance of `Zend\Feed\PubSubHubbub\Subscriber\Callback` when the Callback *URL* is accessed.

Important: `Zend\Feed\PubSubHubbub\Subscriber` implements the Pubsubhubbub 0.2 Specification. As this is a new specification version not all Hubs currently implement it. The new specification allows the Callback *URL* to include a query string which is used by this class, but not supported by all Hubs. In the interests of maximising compatibility it is therefore recommended that the query string component of the Subscriber Callback *URI* be presented as a path element, i.e. recognised as a parameter in the route associated with the Callback *URI* and used by the application's Router.

97.4.1 Subscribing and Unsubscribing

`Zend\Feed\PubSubHubbub\Subscriber` implements a full Pubsubhubbub Subscriber capable of subscribing to, or unsubscribing from, any Topic via any Hub advertised by that Topic. It operates in conjunction with `Zend\Feed\PubSubHubbub\Subscriber\Callback` which accepts requests from a Hub to confirm all subscription or unsubscription attempts (to prevent third-party misuse).

Any subscription (or unsubscription) requires the relevant information before proceeding, i.e. the *URI* of the Topic (Atom or *RSS* feed) to be subscribed to for updates, and the *URI* of the endpoint for the Hub which will handle the subscription and forwarding of the updates. The lifetime of a subscription may be determined by the Hub but most Hubs should support automatic subscription refreshes by checking with the Subscriber. This is supported by `Zend\Feed\PubSubHubbub\Subscriber\Callback` and requires no other work on your part. It is still strongly recommended that you use the Hub sourced subscription time to live (ttl) to schedule the creation of new subscriptions (the process is identical to that for any new subscription) to refresh it with the Hub. While it should not be necessary per se, it covers cases where a Hub may not support automatic subscription refreshing and rules out Hub errors for additional redundancy.

With the relevant information to hand, a subscription can be attempted as demonstrated below:

```
$storage = new Zend\Feed\PubSubHubbub\Model\Subscription;

$subscriber = new Zend\Feed\PubSubHubbub\Subscriber;
$subscriber->setStorage($storage);
$subscriber->addHubUrl('http://hubbub.example.com');
$subscriber->setTopicUrl('http://www.example.net/rss.xml');
$subscriber->setCallbackUrl('http://www.mydomain.com/hubbub/callback');
$subscriber->subscribeAll();
```

In order to store subscriptions and offer access to this data for general use, the component requires a database (a schema is provided later in this section). By default, it is assumed the table name is "subscription" and it utilises `Zend\Db\Table\Abstract` in the background meaning it will use the default adapter you have set for your application. You may also pass a specific custom `Zend\Db\Table\Abstract` instance into the associated model `Zend\Feed\PubSubHubbub\Model\Subscription`. This custom adapter may be as simple in intent as changing the table name to use or as complex as you deem necessary.

While this Model is offered as a default ready-to-roll solution, you may create your own Model using any other backend or database layer (e.g. Doctrine) so long as the resulting class implements the interface `Zend\Feed\PubSubHubbub\Model\SubscriptionInterface`.

An example schema (MySQL) for a subscription table accessible by the provided model may look similar to:

```
CREATE TABLE IF NOT EXISTS `subscription` (
  `id` varchar(32) COLLATE utf8_unicode_ci NOT NULL DEFAULT '',
  `topic_url` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `hub_url` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `created_time` datetime DEFAULT NULL,
  `lease_seconds` bigint(20) DEFAULT NULL,
  `verify_token` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `secret` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `expiration_time` datetime DEFAULT NULL,
  `subscription_state` varchar(12) COLLATE utf8_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Behind the scenes, the Subscriber above will send a request to the Hub endpoint containing the following parameters (based on the previous example):

Table 97.1: Subscription request parameters

Parameter	Value	Explanation
hub.callback	http://www.mydomain.com/hubbub/callback?xhub.subscriptionkey=5536df06b5dcb966edab3a4c4d562b3216a8184	The URI used by a Hub to contact the Subscriber and either request confirmation of a (un)subscription request or send updates from subscribed feeds. The appended query string contains a custom parameter (hence the xhub designation). It is a query string parameter preserved by the Hub and resent with all Subscriber requests. Its purpose is to allow the Subscriber to identify and look up the subscription associated with any Hub request in a backend storage medium. This is a non=standard parameter used by this component in preference to encoding a subscription key in the URI path which is more difficult to implement in a Zend Framework application. Nevertheless, since not all Hubs support query string parameters, we still strongly recommend adding the subscription key as a path component in the form http://www.mydomain.com/hubbub/callback/5536df06b5dcb966edab3a4c4d562 To accomplish this, it requires defining a route capable of parsing out the final value of the key and then retrieving the value and passing it to the Subscriber Callback object. The value would be passed into the method ZendPubSubHubbubSubscriberCallback::setSubscriptionKey(). A detailed example is offered later.
hub.lease_seconds	2592000	The number of seconds for which the Subscriber would like a new subscription to remain valid for (i.e. a TTL). Hubs may enforce their own maximum subscription period. All subscriptions should be renewed by simply re-subscribing before the subscription period ends to ensure continuity of updates. Hubs should additionally attempt to automatically refresh subscriptions before they expire by contacting Subscribers (handled automatically by the Callback class).
hub.mode	subscribe	Simple value indicating this is a subscription request. Unsubscription requests would use the "unsubscribe" value.
hub.topic	http://www.example.net/rss.xml	The URI of the topic (i.e. Atom or RSS feed) which the Subscriber wishes to subscribe to for updates.
hub.verify	sync	Indicates to the Hub the preferred mode of verifying subscriptions or unsubscriptions. It is repeated twice in order of preference. Technically this component does not distinguish between the two modes and treats both equally.
hub.verify	async	Indicates to the Hub the preferred mode of verifying subscriptions or unsubscriptions. It is repeated twice in order of preference. Technically this component does not distinguish between the two modes and treats both equally.
hub.verify_token	3065919804abcaa7212ae89.879827871253878386	A verification token returned to the Subscriber by the Hub when it is confirming a subscription or unsubscription. Offers a measure of reliance that the confirmation request originates from the correct Hub to prevent misuse.

You can modify several of these parameters to indicate a different preference. For example, you can set a different lease seconds value using

`Zend\Feed\PubSubHubbub\Subscriber::setLeaseSeconds()` or show a preference for the async verify mode by using `setPreferredVerificationMode(Zend\Feed\PubSubHubbub\PubSubHubbub::VERIFICATION_`
However the Hubs retain the capability to enforce their own preferences and for this reason the component is deliberately designed to work across almost any set of options with minimum end-user configuration required. Conventions are great when they work!

Note: While Hubs may require the use of a specific verification mode (both are supported by `Zend\Feed\PubSubHubbub`), you may indicate a specific preference using the `setPreferredVerificationMode()` method. In "sync" (synchronous) mode, the Hub attempts to confirm a subscription as soon as it is received, and before responding to the subscription request. In "async" (asynchronous) mode, the Hub will return a response to the subscription request immediately, and its verification request may occur at a later time. Since `Zend\Feed\PubSubHubbub` implements the Subscriber verification role as a separate callback class and requires the use of a backend storage medium, it actually supports both transparently though in terms of end-user performance, asynchronous verification is very much preferred to eliminate the impact of a poorly performing Hub tying up end-user server resources and connections for too long.

Unsubscribing from a Topic follows the exact same pattern as the previous example, with the exception that we should call `unsubscribeAll()` instead. The parameters included are identical to a subscription request with the exception that "`hub.mode`" is set to "unsubscribe".

By default, a new instance of `Zend\PubSubHubbub\Subscriber` will attempt to use a database backed storage medium which defaults to using the default `Zend\Db` adapter with a table name of "subscription". It is recommended to set a custom storage solution where these defaults are not apt either by passing in a new Model supporting the required interface or by passing a new instance of `Zend\Db\Table\Abstract` to the default Model's constructor to change the used table name.

97.4.2 Handling Subscriber Callbacks

Whenever a subscription or unsubscription request is made, the Hub must verify the request by forwarding a new verification request to the Callback *URL* set in the subscription or unsubscription parameters. To handle these Hub requests, which will include all future communications containing Topic (feed) updates, the Callback *URL* should trigger the execution of an instance of `Zend\Feed\PubSubHubbub\Subscriber\Callback` to handle the request.

The Callback class should be configured to use the same storage medium as the Subscriber class. Using it is quite simple since most of its work is performed internally.

```
$storage = new Zend\Feed\PubSubHubbub\Model\Subscription;
$callback = new Zend\Feed\PubSubHubbub\Subscriber\Callback;
$callback->setStorage($storage);
$callback->handle();
$callback->sendResponse();

/**
 * Check if the callback resulting in the receipt of a feed update.
 * Otherwise it was either a (un)sub verification request or invalid request.
 * Typically we need do nothing other than add feed update handling - the rest
 * is handled internally by the class.
 */
if ($callback->hasFeedUpdate()) {
    $feedString = $callback->getFeedUpdate();
    /**
     *  Process the feed update asynchronously to avoid a Hub timeout.
     */
}
```

Note: It should be noted that `Zend\Feed\PubSubHubbub\Subscriber\Callback` may independently parse any incoming query string and other parameters. This is necessary since *PHP* alters the structure and keys of a query string when it is parsed into the `$_GET` or `$_POST` superglobals. For example, all duplicate keys are ignored and periods are converted to underscores. Pubsubhubbub features both of these in the query strings it generates.

Important: It is essential that developers recognise that Hubs are only concerned with sending requests and receiving a response which verifies its receipt. If a feed update is received, it should never be processed on the spot since this leaves the Hub waiting for a response. Rather, any processing should be offloaded to another process or deferred until after a response has been returned to the Hub. One symptom of a failure to promptly complete Hub requests is that a Hub may continue to attempt delivery of the update or verification request leading to duplicated update attempts being processed by the Subscriber. This appears problematic - but in reality a Hub may apply a timeout of just a few seconds, and if no response is received within that time it may disconnect (assuming a delivery failure) and retry later. Note that Hubs are expected to distribute vast volumes of updates so their resources are stretched - please do process feeds asynchronously (e.g. in a separate process or a job queue or even a cron scheduled task) as much as possible.

97.4.3 Setting Up And Using A Callback URL Route

As noted earlier, the `Zend\Feed\PubSubHubbub\Subscriber\Callback` class receives the combined key associated with any subscription from the Hub via one of two methods. The technically preferred method is to add this key to the Callback *URL* employed by the Hub in all future requests using a query string parameter with the key "xhub.subscription". However, for historical reasons, primarily that this was not supported in Pubsubhubbub 0.1 (it was recently added in 0.2 only), it is strongly recommended to use the most compatible means of adding this key to the Callback *URL* by appending it to the *URL*'s path.

Thus the *URL* http://www.example.com/callback?xhub.subscription=key would become http://www.example.com/callback/key.

Since the query string method is the default in anticipation of a greater level of future support for the full 0.2 specification, this requires some additional work to implement.

The first step to make the `Zend\Feed\PubSubHubbub\Subscriber\Callback` class aware of the path contained subscription key. It's manually injected therefore since it also requires manually defining a route for this purpose. This is achieved simply by called the method `Zend\Feed\PubSubHubbub\Subscriber\Callback::setSubscriptionKey()` with the parameter being the key value available from the Router. The example below demonstrates this using a Zend Framework controller.

```
use Zend\Mvc\Controller\AbstractActionController;

class CallbackController extends AbstractActionController
{

    public function indexAction()
    {
        $storage = new Zend\Feed\PubSubHubbub\Model\Subscription;
        $callback = new Zend\Feed\PubSubHubbub\Subscriber\Callback;
        $callback->setStorage($storage);
        /**
         * Inject subscription key parsing from URL path using
         * a parameter from Router.
         */
        $subscriptionKey = $this->params()->fromRoute('subkey');
        $callback->setSubscriptionKey($subscriptionKey);
```

```
        $callback->handle();
        $callback->sendResponse();

        /**
         * Check if the callback resulting in the receipt of a feed update.
         * Otherwise it was either a (un)sub verification request or invalid
         * request. Typically we need do nothing other than add feed update
         * handling - the rest is handled internally by the class.
         */
        if ($callback->hasFeedUpdate()) {
            $feedString = $callback->getFeedUpdate();
            /**
             *  Process the feed update asynchronously to avoid a Hub timeout.
             */
        }
    }

}
```

Actually adding the route which would map the path-appended key to a parameter for retrieval from a controller can be accomplished using a Route like in the example below.

```
// Callback Route to enable appending a PuSH Subscription's lookup key
$route = Zend\Mvc\Router\Http\Segment::factory(array(
   'route' => '/callback/:subkey',
   'constraints' => array(
      'subkey' => '[a-z0-9]+'
   ),
   'defaults' => array(
      'controller' => 'application-index',
      'action' => 'index'
   )
));
```

CHAPTER

NINETYEIGHT

ZEND\FILE\CLASSFILELOCATOR

98.1 Overview

TODO

98.2 Available Methods

TODO

98.3 Examples

TODO

CHAPTER

NINETYNINE

INTRODUCTION TO ZEND\FILTER

The `Zend\Filter` component provides a set of commonly needed data filters. It also provides a simple filter chaining mechanism by which multiple filters may be applied to a single datum in a user-defined order.

99.1 What is a filter?

In the physical world, a filter is typically used for removing unwanted portions of input, and the desired portion of the input passes through as filter output (e.g., coffee). In such scenarios, a filter is an operator that produces a subset of the input. This type of filtering is useful for web applications - removing illegal input, trimming unnecessary white space, etc.

This basic definition of a filter may be extended to include generalized transformations upon input. A common transformation applied in web applications is the escaping of *HTML* entities. For example, if a form field is automatically populated with untrusted input (e.g., from a web browser), this value should either be free of *HTML* entities or contain only escaped *HTML* entities, in order to prevent undesired behavior and security vulnerabilities. To meet this requirement, *HTML* entities that appear in the input must either be removed or escaped. Of course, which approach is more appropriate depends on the situation. A filter that removes the *HTML* entities operates within the scope of the first definition of filter - an operator that produces a subset of the input. A filter that escapes the *HTML* entities, however, transforms the input (e.g., "&" is transformed to "&"). Supporting such use cases for web developers is important, and "to filter," in the context of using `Zend\Filter`, means to perform some transformations upon input data.

99.2 Basic usage of filters

Having this filter definition established provides the foundation for `Zend\Filter\FilterInterface`, which requires a single method named `filter()` to be implemented by a filter class.

Following is a basic example of using a filter upon two input data, the ampersand (**&**) and double quote (**"**) characters:

```
$htmlEntities = new Zend\Filter\HtmlEntities();

echo $htmlEntities->filter('&'); // &amp;
echo $htmlEntities->filter('"'); // &quot;
```

Also, if a Filter inherits from `Zend\Filter\AbstractFilter` (just like all out-of-the-box Filters) you can also use them as such:

```
$strtolower = new Zend\Filter\StringToLower;

```

```
3 echo $strtolower('I LOVE ZF2!'); // i love zf2!
4 $zf2love = $strtolower('I LOVE ZF2!');
```

CHAPTER

USING THE STATICFILTER

If it is inconvenient to load a given filter class and create an instance of the filter, you can use `StaticFilter` with it's method `execute()` as an alternative invocation style. The first argument of this method is a data input value, that you would pass to the `filter()` method. The second argument is a string, which corresponds to the basename of the filter class, relative to the `Zend\Filter` namespace. The `execute()` method automatically loads the class, creates an instance, and applies the `filter()` method to the data input.

```
echo StaticFilter::execute('&', 'HtmlEntities');
```

You can also pass an array of constructor arguments, if they are needed for the filter class.

```
echo StaticFilter::execute('"',
                           'HtmlEntities',
                           array('quotestyle' => ENT_QUOTES));
```

The static usage can be convenient for invoking a filter ad hoc, but if you have the need to run a filter for multiple inputs, it's more efficient to follow the first example above, creating an instance of the filter object and calling its `filter()` method.

Also, the `FilterChain` class allows you to instantiate and run multiple filter and validator classes on demand to process sets of input data. See *FilterChain*.

You can set and receive the `FilterPluginManager` for the `StaticFilter` to amend the standard filter classes.

```
$pluginManager = StaticFilter::getPluginManager()->setInvokableClass(
    'myNewFilter', 'MyCustom\Filter\MyNewFilter'
);

StaticFilter::setPluginManager(new MyFilterPluginManager());
```

This is useful when adding custom filters to be used by the `StaticFilter`.

100.1 Double filtering

When using two filters after each other you have to keep in mind that it is often not possible to get the original output by using the opposite filter. Take the following example:

```
$original = "my_original_content";

// Attach a filter
$filter   = new Zend\Filter\Word\UnderscoreToCamelCase();
$filtered = $filter->filter($original);

// Use it's opposite
$filter2  = new Zend\Filter\Word\CamelCaseToUnderscore();
$filtered = $filter2->filter($filtered)
```

The above code example could lead to the impression that you will get the original output after the second filter has been applied. But thinking logically this is not the case. After applying the first filter **my_original_content** will be changed to **MyOriginalContent**. But after applying the second filter the result is **My_Original_Content**.

As you can see it is not always possible to get the original output by using a filter which seems to be the opposite. It depends on the filter and also on the given input.

CHAPTER

ONE

STANDARD FILTER CLASSES

Zend Framework comes with a standard set of filters, which are ready for you to use.

101.1 Alnum

The `Alnum` filter can be used to return only alphabetic characters and digits in the unicode "letter" and "number" categories, respectively. All other characters are suppressed.

Supported Options for Alnum Filter

The following options are supported for `Alnum`:

`Alnum([ boolean $allowWhiteSpace [, string $locale ]])`

- `$allowWhiteSpace`: If set to true then whitespace characters are allowed. Otherwise they are suppressed. Default is "false" (whitespace is not allowed).

  Methods for getting/setting the allowWhiteSpace option are also available: `getAllowWhiteSpace()` and `setAllowWhiteSpace()`

- `$locale`: The locale string used in identifying the characters to filter (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`).

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

Alnum Filter Usage

```
// Default settings, deny whitespace
$filter = new \Zend\I18n\Filter\Alnum();
echo $filter->filter("This is (my) content: 123");
// Returns "Thisismycontent123"

// First param in constructor is $allowWhiteSpace
$filter = new \Zend\I18n\Filter\Alnum(true);
echo $filter->filter("This is (my) content: 123");
// Returns "This is my content 123"
```

Note: `Alnum` works on almost all languages, except: Chinese, Japanese and Korean. Within these languages the english alphabet is used instead of the characters from these languages. The language itself is detected using the `Locale`.

101.2 Alpha

The `Alpha` filter can be used to return only alphabetic characters in the unicode "letter" category. All other characters are suppressed.

Supported Options for Alpha Filter

The following options are supported for `Alpha`:

`Alpha([ boolean $allowWhiteSpace [, string $locale ]])`

- `$allowWhiteSpace`: If set to true then whitespace characters are allowed. Otherwise they are suppressed. Default is "false" (whitespace is not allowed).

  Methods for getting/setting the allowWhiteSpace option are also available: `getAllowWhiteSpace()` and `setAllowWhiteSpace()`

- `$locale`: The locale string used in identifying the characters to filter (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`).

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

Alpha Filter Usage

```
// Default settings, deny whitespace
$filter = new \Zend\I18n\Filter\Alpha();
echo $filter->filter("This is (my) content: 123");
// Returns "Thisismycontent"

// Allow whitespace
$filter = new \Zend\I18n\Filter\Alpha(true);
echo $filter->filter("This is (my) content: 123");
// Returns "This is my content "
```

Note: `Alpha` works on almost all languages, except: Chinese, Japanese and Korean. Within these languages the english alphabet is used instead of the characters from these languages. The language itself is detected using the `Locale`.

101.3 BaseName

`Zend\Filter\BaseName` allows you to filter a string which contains the path to a file and it will return the base name of this file.

Supported Options

There are no additional options for `Zend\Filter\BaseName`.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\BaseName();

print $filter->filter('/vol/tmp/filename');
```

This will return 'filename'.

```
$filter = new Zend\Filter\BaseName();

print $filter->filter('/vol/tmp/filename.txt');
```

This will return '`filename.txt`'.

101.4 Boolean

This filter changes a given input to be a `BOOLEAN` value. This is often useful when working with databases or when processing form values.

Supported Options

The following options are supported for `Zend\Filter\Boolean`:

- **casting**: When this option is set to `TRUE` then any given input will be casted to boolean. This option defaults to `TRUE`.
- **locale**: This option sets the locale which will be used to detect localized input.
- **type**: The `type` option sets the boolean type which should be used. Read the following for details.

Default Behavior

By default, this filter works by casting the input to a `BOOLEAN` value; in other words, it operates in a similar fashion to calling `(boolean) $value`.

```
$filter = new Zend\Filter\Boolean();
$value  = '';
$result = $filter->filter($value);
// returns false
```

This means that without providing any configuration, `Zend\Filter\Boolean` accepts all input types and returns a `BOOLEAN` just as you would get by type casting to `BOOLEAN`.

Changing the Default Behavior

Sometimes casting with `(boolean)` will not suffice. `Zend\Filter\Boolean` allows you to configure specific types to convert, as well as which to omit.

The following types can be handled:

- **boolean**: Returns a boolean value as is.
- **integer**: Converts an integer **0** value to `FALSE`.

- **float**: Converts a float **0.0** value to `FALSE`.
- **string**: Converts an empty string **''** to `FALSE`.
- **zero**: Converts a string containing the single character zero (**'0'**) to `FALSE`.
- **empty_array**: Converts an empty **array** to `FALSE`.
- **null**: Converts a `NULL` value to `FALSE`.
- **php**: Converts values according to *PHP* when casting them to `BOOLEAN`.
- **false_string**: Converts a string containing the word "false" to a boolean `FALSE`.
- **yes**: Converts a localized string which contains the word "no" to `FALSE`.
- **all**: Converts all above types to `BOOLEAN`.

All other given values will return `TRUE` by default.

There are several ways to select which of the above types are filtered. You can give one or multiple types and add them, you can give an array, you can use constants, or you can give a textual string. See the following examples:

```
// converts 0 to false
$filter = new Zend\Filter\Boolean(Zend\Filter\Boolean::INTEGER);

// converts 0 and '0' to false
$filter = new Zend\Filter\Boolean(
    Zend\Filter\Boolean::INTEGER + Zend\Filter\Boolean::ZERO
);

// converts 0 and '0' to false
$filter = new Zend\Filter\Boolean(array(
    'type' => array(
        Zend\Filter\Boolean::INTEGER,
        Zend\Filter\Boolean::ZERO,
    ),
));

// converts 0 and '0' to false
$filter = new Zend\Filter\Boolean(array(
    'type' => array(
        'integer',
        'zero',
    ),
));
```

You can also give an instance of `Zend\Config\Config` to set the desired types. To set types after instantiation, use the `setType()` method.

Localized Booleans

As mentioned previously, `Zend\Filter\Boolean` can also recognise localized "yes" and "no" strings. This means that you can ask your customer in a form for "yes" or "no" within his native language and `Zend\Filter\Boolean` will convert the response to the appropriate boolean value.

To set the desired locale, you can either use the `locale` option, or the method `setLocale()`.

```
$filter = new Zend\Filter\Boolean(array(
    'type'   => Zend\Filter\Boolean::ALL,
    'locale' => 'de',
```

```
));

// returns false
echo $filter->filter('nein');

$filter->setLocale('en');

// returns true
$filter->filter('yes');
```

Disable Casting

Sometimes it is necessary to recognise only `TRUE` or `FALSE` and return all other values without changes. `Zend\Filter\Boolean` allows you to do this by setting the `casting` option to `FALSE`.

In this case `Zend\Filter\Boolean` will work as described in the following table, which shows which values return `TRUE` or `FALSE`. All other given values are returned without change when `casting` is set to `FALSE`

Table 101.1: Usage without casting

Type	True	False
Zend\Filter\Boolean::BOOLEAN	TRUE	FALSE
Zend\Filter\Boolean::INTEGER	0	1
Zend\Filter\Boolean::FLOAT	0.0	1.0
Zend\Filter\Boolean::STRING	“”	
Zend\Filter\Boolean::ZERO	“0”	“1”
Zend\Filter\Boolean::EMPTY_ARRAY	array()	
Zend\Filter\Boolean::NULL	NULL	
Zend\Filter\Boolean::FALSE_STRING	“false” (case independently)	“true” (case independently)
Zend\Filter\Boolean::YES	localized “yes” (case independently)	localized “no” (case independently)

The following example shows the behaviour when changing the `casting` option:

```
$filter = new Zend\Filter\Boolean(array(
    'type'    => Zend\Filter\Boolean::ALL,
    'casting' => false,
));

// returns false
echo $filter->filter(0);

// returns true
echo $filter->filter(1);

// returns the value
echo $filter->filter(2);
```

101.5 Callback

This filter allows you to use own methods in conjunction with `Zend\Filter`. You don't have to create a new filter when you already have a method which does the job.

Supported Options

The following options are supported for `Zend\Filter\Callback`:

- **callback**: This sets the callback which should be used.
- **options**: This property sets the options which are used when the callback is processed.

Basic Usage

The usage of this filter is quite simple. Let's expect we want to create a filter which reverses a string.

```
$filter = new Zend\Filter\Callback('strrev');

print $filter->filter('Hello!');
// returns "!olleH"
```

As you can see it's really simple to use a callback to define a own filter. It is also possible to use a method, which is defined within a class, by giving an array as callback.

```
// Our classdefinition
class MyClass
{
    public function Reverse($param);
}

// The filter definition
$filter = new Zend\Filter\Callback(array('MyClass', 'Reverse'));
print $filter->filter('Hello!');
```

To get the actual set callback use `getCallback()` and to set another callback use `setCallback()`.

Note: Possible exceptions

You should note that defining a callback method which can not be called will raise an exception.

Default Parameters Within a Callback

It is also possible to define default parameters, which are given to the called method as array when the filter is executed. This array will be concatenated with the value which will be filtered.

```
$filter = new Zend\Filter\Callback(
    array(
        'callback' => 'MyMethod',
        'options'  => array('key' => 'param1', 'key2' => 'param2')
    )
);
$filter->filter(array('value' => 'Hello'));
```

When you would call the above method definition manually it would look like this:

```
$value = MyMethod('Hello', 'param1', 'param2');
```

101.6 Compress and Decompress

These two filters are capable of compressing and decompressing strings, files, and directories.

Supported Options

The following options are supported for `Zend\Filter\Compress` and `Zend\Filter\Decompress`:

- **adapter**: The compression adapter which should be used. It defaults to `Gz`.
- **options**: Additional options which are given to the adapter at initiation. Each adapter supports it's own options.

Supported Compression Adapters

The following compression formats are supported by their own adapter:

- **Bz2**
- **Gz**
- **Lzf**
- **Rar**
- **Tar**
- **Zip**

Each compression format has different capabilities as described below. All compression filters may be used in approximately the same ways, and differ primarily in the options available and the type of compression they offer (both algorithmically as well as string vs. file vs. directory)

Generic Handling

To create a compression filter you need to select the compression format you want to use. The following description takes the **Bz2** adapter. Details for all other adapters are described after this section.

The two filters are basically identical, in that they utilize the same backends. `Zend\Filter\Compress` should be used when you wish to compress items, and `Zend\Filter\Decompress` should be used when you wish to decompress items.

For instance, if we want to compress a string, we have to initiate `Zend\Filter\Compress` and indicate the desired adapter.

```
$filter = new Zend\Filter\Compress('Bz2');
```

To use a different adapter, you simply specify it to the constructor.

You may also provide an array of options or a Traversable object. If you do, provide minimally the key "adapter", and then either the key "options" or "adapterOptions" (which should be an array of options to provide to the adapter on instantiation).

```
$filter = new Zend\Filter\Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'blocksize' => 8,
    ),
));
```

Note: Default compression Adapter

When no compression adapter is given, then the **Gz** adapter will be used.

Almost the same usage is we want to decompress a string. We just have to use the decompression filter in this case.

```
$filter = new Zend\Filter\Decompress('Bz2');
```

To get the compressed string, we have to give the original string. The filtered value is the compressed version of the original string.

```
$filter     = new Zend\Filter\Compress('Bz2');
$compressed = $filter->filter('Uncompressed string');
// Returns the compressed string
```

Decompression works the same way.

```
$filter     = new Zend\Filter\Decompress('Bz2');
$compressed = $filter->filter('Compressed string');
// Returns the uncompressed string
```

Note: Note on string compression

Not all adapters support string compression. Compression formats like **Rar** can only handle files and directories. For details, consult the section for the adapter you wish to use.

Creating an Archive

Creating an archive file works almost the same as compressing a string. However, in this case we need an additional parameter which holds the name of the archive we want to create.

```
$filter     = new Zend\Filter\Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'archive' => 'filename.bz2',
    ),
));
$compressed = $filter->filter('Uncompressed string');
// Returns true on success and creates the archive file
```

In the above example the uncompressed string is compressed, and is then written into the given archive file.

Note: Existing archives will be overwritten

The content of any existing file will be overwritten when the given filename of the archive already exists.

When you want to compress a file, then you must give the name of the file with its path.

```
$filter     = new Zend\Filter\Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'archive' => 'filename.bz2'
    ),
));
```

```
$compressed = $filter->filter('C:\temp\compressme.txt');
// Returns true on success and creates the archive file
```

You may also specify a directory instead of a filename. In this case the whole directory with all its files and subdirectories will be compressed into the archive.

```
$filter     = new Zend\Filter\Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'archive' => 'filename.bz2'
    ),
));
$compressed = $filter->filter('C:\temp\somedir');
// Returns true on success and creates the archive file
```

Note: Do not compress large or base directories

You should never compress large or base directories like a complete partition. Compressing a complete partition is a very time consuming task which can lead to massive problems on your server when there is not enough space or your script takes too much time.

Decompressing an Archive

Decompressing an archive file works almost like compressing it. You must specify either the `archive` parameter, or give the filename of the archive when you decompress the file.

```
$filter     = new Zend\Filter\Decompress('Bz2');
$decompressed = $filter->filter('filename.bz2');
// Returns true on success and decompresses the archive file
```

Some adapters support decompressing the archive into another subdirectory. In this case you can set the `target` parameter.

```
$filter     = new Zend\Filter\Decompress(array(
    'adapter' => 'Zip',
    'options' => array(
        'target' => 'C:\temp',
    )
));
$decompressed = $filter->filter('filename.zip');
// Returns true on success and decompresses the archive file
// into the given target directory
```

Note: Directories to extract to must exist

When you want to decompress an archive into a directory, then that directory must exist.

Bz2 Adapter

The Bz2 Adapter can compress and decompress:

- Strings
- Files

- Directories

This adapter makes use of *PHP*'s Bz2 extension.

To customize compression, this adapter supports the following options:

- **Archive**: This parameter sets the archive file which should be used or created.
- **Blocksize**: This parameter sets the blocksize to use. It can be from '0' to '9'. The default value is '4'.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Blocksize' are `getBlocksize()` and `setBlocksize()`. You can also use the `setOptions()` method which accepts all options as array.

Gz Adapter

The Gz Adapter can compress and decompress:

- Strings
- Files
- Directories

This adapter makes use of *PHP*'s Zlib extension.

To customize the compression this adapter supports the following options:

- **Archive**: This parameter sets the archive file which should be used or created.
- **Level**: This compression level to use. It can be from '0' to '9'. The default value is '9'.
- **Mode**: There are two supported modes. 'compress' and 'deflate'. The default value is 'compress'.

All options can be set at initiation or by using a related method. For example, the related methods for 'Level' are `getLevel()` and `setLevel()`. You can also use the `setOptions()` method which accepts all options as array.

Lzf Adapter

The Lzf Adapter can compress and decompress:

- Strings

Note: Lzf supports only strings

The Lzf adapter can not handle files and directories.

This adapter makes use of *PHP*'s Lzf extension.

There are no options available to customize this adapter.

Rar Adapter

The Rar Adapter can compress and decompress:

- Files
- Directories

Note: Rar does not support strings

The Rar Adapter can not handle strings.

This adapter makes use of *PHP*'s Rar extension.

Note: Rar compression not supported

Due to restrictions with the Rar compression format, there is no compression available for free. When you want to compress files into a new Rar archive, you must provide a callback to the adapter that can invoke a Rar compression program.

To customize the compression this adapter supports the following options:

- **Archive**: This parameter sets the archive file which should be used or created.
- **Callback**: A callback which provides compression support to this adapter.
- **Password**: The password which has to be used for decompression.
- **Target**: The target where the decompressed files will be written to.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Target' are `getTarget()` and `setTarget()`. You can also use the `setOptions()` method which accepts all options as array.

Tar Adapter

The Tar Adapter can compress and decompress:

- Files
- Directories

Note: Tar does not support strings

The Tar Adapter can not handle strings.

This adapter makes use of *PEAR*'s `Archive_Tar` component.

To customize the compression this adapter supports the following options:

- **Archive**: This parameter sets the archive file which should be used or created.
- **Mode**: A mode to use for compression. Supported are either '`NULL`' which means no compression at all, 'Gz' which makes use of *PHP*'s Zlib extension and 'Bz2' which makes use of *PHP*'s Bz2 extension. The default value is '`NULL`'.
- **Target**: The target where the decompressed files will be written to.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Target' are `getTarget()` and `setTarget()`. You can also use the `setOptions()` method which accepts all options as array.

Note: Directory usage

When compressing directories with Tar then the complete file path is used. This means that created Tar files will not only have the subdirectory but the complete path for the compressed file.

Zip Adapter

The Zip Adapter can compress and decompress:

- Strings
- Files
- Directories

Note: Zip does not support string decompression

The Zip Adapter can not handle decompression to a string; decompression will always be written to a file.

This adapter makes use of *PHP*'s `Zip` extension.

To customize the compression this adapter supports the following options:

- **Archive**: This parameter sets the archive file which should be used or created.
- **Target**: The target where the decompressed files will be written to.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Target' are `getTarget()` and `setTarget()`. You can also use the `setOptions()` method which accepts all options as array.

101.7 Digits

Returns the string `$value`, removing all but digits.

Supported Options

There are no additional options for `Zend\Filter\Digits`.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Digits();

print $filter->filter('October 2012');
```

This returns "2012".

```
$filter = new Zend\Filter\Digits();

print $filter->filter('HTML 5 for Dummies');
```

This returns "5".

101.8 Dir

Given a string containing a path to a file, this function will return the name of the directory.

Supported Options

There are no additional options for `Zend\Filter\Dir`.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Dir();

print $filter->filter('/etc/passwd');
```

This returns "`/etc`".

```
$filter = new Zend\Filter\Dir();

print $filter->filter('C:/Temp/x');
```

This returns "`C:/Temp`".

101.9 Encrypt and Decrypt

These filters allow to encrypt and decrypt any given string. Therefor they make use of Adapters. Actually there are adapters for the `Zend\Crypt\BlockCipher` class and the `OpenSSL` extension of *PHP*.

Supported Options

The following options are supported for `Zend\Filter\Encrypt` and `Zend\Filter\Decrypt`:

- **adapter**: This sets the encryption adapter which should be used
- **algorithm**: Only `BlockCipher`. The algorithm which has to be used by the adapter

  `Zend\Crypt\Symmetric\Mcrypt`. It should be one of the algorithm ciphers supported by `Zend\Crypt\Symmetric\Mcrypt` (see the `getSupportedAlgorithms()` method). If not set it defaults to `aes`, the Advanced Encryption Standard (see *Zend\Crypt\BlockCipher* for more details).
- **compression**: If the encrypted value should be compressed. Default is no compression.
- **envelope**: Only `OpenSSL`. The encrypted envelope key from the user who encrypted the content. You can either provide the path and filename of the key file, or just the content of the key file itself. When the `package` option has been set, then you can omit this parameter.
- **key**: Only `BlockCipher`. The encryption key with which the input will be encrypted. You need the same key for decryption.
- **mode**: Only `BlockCipher`. The encryption mode which has to be used. It should be one of the modes which can be found under PHP's mcrypt modes. If not set it defaults to 'cbc'.

- **mode_directory**: Only `BlockCipher`. The directory where the mode can be found. If not set it defaults to the path set within the `Mcrypt` extension.
- **package**: Only `OpenSSL`. If the envelope key should be packed with the encrypted value. Default is `FALSE`.
- **private**: Only `OpenSSL`. Your private key which will be used for encrypting the content. Also the private key can be either a filename with path of the key file, or just the content of the key file itself.
- **public**: Only `OpenSSL`. The public key of the user whom you want to provide the encrypted content. You can give multiple public keys by using an array. You can either provide the path and filename of the key file, or just the content of the key file itself.
- **vector**: Only `BlockCipher`. The initialization vector which shall be used. If not set it will be a random vector.

Adapter Usage

As these two encryption methodologies work completely different, also the usage of the adapters differ. You have to select the adapter you want to use when initiating the filter.

```
// Use the BlockCipher adapter
$filter1 = new Zend\Filter\Encrypt(array('adapter' => 'BlockCipher'));

// Use the OpenSSL adapter
$filter2 = new Zend\Filter\Encrypt(array('adapter' => 'openssl'));
```

To set another adapter you can also use `setAdapter()`, and the `getAdapter()` method to receive the actual set adapter.

```
// Use the OpenSSL adapter
$filter = new Zend\Filter\Encrypt();
$filter->setAdapter('openssl');
```

Note: When you do not supply the `adapter` option or do not use `setAdapter()`, then the `BlockCipher` adapter will be used per default.

Encryption with BlockCipher

To encrypt a string using the `BlockCipher` you have to specify the encryption key using the `setKey()` method or passing it during the constructor.

```
// Use the default AES encryption algorithm
$filter = new Zend\Filter\Encrypt(array('adapter' => 'BlockCipher'));
$filter->setKey('encryption key');

// or
// $filter = new Zend\Filter\Encrypt(array(
//     'adapter' => 'BlockCipher',
//     'key'     => 'encryption key'
// ));

$encrypted = $filter->filter('text to be encrypted');
printf ("Encrypted text: %s\n", $encrypted);
```

You can get and set the encryption values also afterwards with the `getEncryption()` and `setEncryption()` methods.

```
// Use the default AES encryption algorithm
$filter = new Zend\Filter\Encrypt(array('adapter' => 'BlockCipher'));
$filter->setKey('encryption key');
var_dump($filter->getEncryption());

// Will print:
//array(4) {
//  ["key_iteration"]=>
//  int(5000)
//  ["algorithm"]=>
//  string(3) "aes"
//  ["hash"]=>
//  string(6) "sha256"
//  ["key"]=>
//  string(14) "encryption key"
//}
```

Note: The `BlockCipher` adapter uses the Mcrypt PHP extension by default. That means you will need to install the *Mcrypt* module in your PHP environment.

If you don't specify an initialization Vector (*salt* or *iv*), the BlockCipher will generate a random value during each encryption. If you try to execute the following code the output will be always different (note that even if the output is always different you can decrypt it using the same key).

```
$key  = 'encryption key';
$text = 'message to encrypt';

// use the default adapter that is BlockCipher
$filter = new \Zend\Filter\Encrypt();
$filter->setKey('encryption key');
for ($i=0; $i < 10; $i++) {
   printf("%d) %s\n", $i, $filter->filter($text));
}
```

If you want to obtain the same output you need to specify a fixed Vector, using the *setVector()* method. This script will produce always the same encryption output.

```
// use the default adapter that is BlockCipher
$filter = new \Zend\Filter\Encrypt();
$filter->setKey('encryption key');
$filter->setVector('12345678901234567890');
printf("%s\n", $filter->filter('message'));

// output:
// 04636a6cb8276fad0787a2e187803b6557f77825d5ca6ed4392be702b9754bb3MTIzNDU2Nzg5MDEyMzQ1NgZ+zPwTGpV6g(
```

Note: For a security reason it's always better to use a different Vector on each encryption. We suggest to use the *setVector()* method only if you really need it.

Decryption with BlockCipher

For decrypting content which was previously encrypted with `BlockCipher` you need to have the options with which the encryption has been called.

If you used only the encryption key, you can just use it to decrypt the content. As soon as you have provided all options decryption is as simple as encryption.

```
$content = '04636a6cb8276fad0787a2e187803b6557f77825d5ca6ed4392be702b9754bb3MTIzNDU2Nzg5MDEyMzQ1NgZ+
// use the default adapter that is BlockCipher
$filter = new Zend\Filter\Decrypt();
$filter->setKey('encryption key');
printf("Decrypt: %s\n", $filter->filter($content));

// output:
// Decrypt: message
```

Note that even if we did not specify the same Vector, the `BlockCipher` is able to decrypt the message because the Vector is stored in the encryption string itself (note that the Vector can be stored in plaintext, it is not a secret, the Vector is only used to improve the randomness of the encryption algorithm).

Note: You should also note that all settings which be checked when you create the instance or when you call `setEncryption()`.

Encryption with OpenSSL

When you have installed the `OpenSSL` extension you can use the `OpenSSL` adapter. You can get or set the public keys also afterwards with the `getPublicKey()` and `setPublicKey()` methods. The private key can also be get and set with the related `getPrivateKey()` and `setPrivateKey()` methods.

```
// Use openssl and provide a private key
$filter = new Zend\Filter\Encrypt(array(
   'adapter' => 'openssl',
   'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the public keys at initiation
$filter->setPublicKey(array(
   '/public/key/path/first.pem',
   '/public/key/path/second.pem'
));
```

Note: Note that the `OpenSSL` adapter will not work when you do not provide valid keys.

When you want to encode also the keys, then you have to provide a passphrase with the `setPassphrase()` method. When you want to decode content which was encoded with a passphrase you will not only need the public key, but also the passphrase to decode the encrypted key.

```
// Use openssl and provide a private key
$filter = new Zend\Filter\Encrypt(array(
   'adapter' => 'openssl',
   'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the public keys at initiation
$filter->setPublicKey(array(
   '/public/key/path/first.pem',
   '/public/key/path/second.pem'
));
$filter->setPassphrase('mypassphrase');
```

At last, when you use OpenSSL you need to give the receiver the encrypted content, the passphrase when have provided one, and the envelope keys for decryption.

This means for you, that you have to get the envelope keys after the encryption with the `getEnvelopeKey()` method.

So our complete example for encrypting content with `OpenSSL` look like this.

```
// Use openssl and provide a private key
$filter = new Zend\Filter\Encrypt(array(
   'adapter' => 'openssl',
   'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the public keys at initiation
$filter->setPublicKey(array(
   '/public/key/path/first.pem',
   '/public/key/path/second.pem'
));
$filter->setPassphrase('mypassphrase');

$encrypted = $filter->filter('text_to_be_encoded');
$envelope  = $filter->getEnvelopeKey();
print $encrypted;

// For decryption look at the Decrypt filter
```

Simplified usage with Openssl

As seen before, you need to get the envelope key to be able to decrypt the previous encrypted value. This can be very annoying when you work with multiple values.

To have a simplified usage you can set the `package` option to `TRUE`. The default value is `FALSE`.

```
// Use openssl and provide a private key
$filter = new Zend\Filter\Encrypt(array(
   'adapter' => 'openssl',
   'private' => '/path/to/mykey/private.pem',
   'public'  => '/public/key/path/public.pem',
   'package' => true
));

$encrypted = $filter->filter('text_to_be_encoded');
print $encrypted;

// For decryption look at the Decrypt filter
```

Now the returned value contains the encrypted value and the envelope. You don't need to get them after the compression. But, and this is the negative aspect of this feature, the encrypted value can now only be decrypted by using `Zend\Filter\Encrypt`.

Compressing Content

Based on the original value, the encrypted value can be a very large string. To reduce the value `Zend\Filter\Encrypt` allows the usage of compression.

The `compression` option can either be set to the name of a compression adapter, or to an array which sets all wished options for the compression adapter.

```
// Use basic compression adapter
$filter1 = new Zend\Filter\Encrypt(array(
   'adapter'     => 'openssl',
   'private'     => '/path/to/mykey/private.pem',
   'public'      => '/public/key/path/public.pem',
   'package'     => true,
   'compression' => 'bz2'
));

// Use basic compression adapter
$filter2 = new Zend\Filter\Encrypt(array(
   'adapter'     => 'openssl',
   'private'     => '/path/to/mykey/private.pem',
   'public'      => '/public/key/path/public.pem',
   'package'     => true,
   'compression' => array('adapter' => 'zip', 'target' => '\usr\tmp\tmp.zip')
));
```

Note: Decryption with same settings

When you want to decrypt a value which is additionally compressed, then you need to set the same compression settings for decryption as for encryption. Otherwise the decryption will fail.

Decryption with OpenSSL

Decryption with `OpenSSL` is as simple as encryption. But you need to have all data from the person who encrypted the content. See the following example:

```
// Use openssl and provide a private key
$filter = new Zend\Filter\Decrypt(array(
   'adapter' => 'openssl',
   'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the envelope keys at initiation
$filter->setEnvelopeKey(array(
   '/key/from/encoder/first.pem',
   '/key/from/encoder/second.pem'
));
```

Note: Note that the `OpenSSL` adapter will not work when you do not provide valid keys.

Optionally it could be necessary to provide the passphrase for decrypting the keys themself by using the `setPassphrase()` method.

```
// Use openssl and provide a private key
$filter = new Zend\Filter\Decrypt(array(
   'adapter' => 'openssl',
   'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the envelope keys at initiation
```

```
$filter->setEnvelopeKey(array(
   '/key/from/encoder/first.pem',
   '/key/from/encoder/second.pem'
));
$filter->setPassphrase('mypassphrase');
```

At last, decode the content. Our complete example for decrypting the previously encrypted content looks like this.

```
// Use openssl and provide a private key
$filter = new Zend\Filter\Decrypt(array(
   'adapter' => 'openssl',
   'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the envelope keys at initiation
$filter->setEnvelopeKey(array(
   '/key/from/encoder/first.pem',
   '/key/from/encoder/second.pem'
));
$filter->setPassphrase('mypassphrase');

$decrypted = $filter->filter('encoded_text_normally_unreadable');
print $decrypted;
```

101.10 HtmlEntities

Returns the string `$value`, converting characters to their corresponding *HTML* entity equivalents where they exist.

Supported Options

The following options are supported for `Zend\Filter\HtmlEntities`:

- **quotestyle**: Equivalent to the *PHP* htmlentities native function parameter **quote_style**. This allows you to define what will be done with 'single' and "double" quotes. The following constants are accepted: `ENT_COMPAT`, `ENT_QUOTES` `ENT_NOQUOTES` with the default being `ENT_COMPAT`.
- **charset**: Equivalent to the *PHP* htmlentities native function parameter **charset**. This defines the character set to be used in filtering. Unlike the *PHP* native function the default is 'UTF-8'. See "http://php.net/htmlentities" for a list of supported character sets.

  Note: This option can also be set via the `$options` parameter as a Traversable object or array. The option key will be accepted as either charset or encoding.

- **doublequote**: Equivalent to the *PHP* htmlentities native function parameter **double_encode**. If set to false existing html entities will not be encoded. The default is to convert everything (true).

  Note: This option must be set via the `$options` parameter or the `setDoubleEncode()` method.

Basic Usage

See the following example for the default behavior of this filter.

```
$filter = new Zend\Filter\HtmlEntities();

print $filter->filter('<');
```

Quote Style

`Zend\Filter\HtmlEntities` allows changing the quote style used. This can be useful when you want to leave double, single, or both types of quotes un-filtered. See the following example:

```
$filter = new Zend\Filter\HtmlEntities(array('quotestyle' => ENT_QUOTES));

$input  = "A 'single' and " . '"double"';
print $filter->filter($input);
```

The above example returns `A 'single' and "double"`. Notice that `'single'` as well as `"double"` quotes are filtered.

```
$filter = new Zend\Filter\HtmlEntities(array('quotestyle' => ENT_COMPAT));

$input  = "A 'single' and " . '"double"';
print $filter->filter($input);
```

The above example returns `A 'single' and "double"`. Notice that `"double"` quotes are filtered while `'single'` quotes are not altered.

```
$filter = new Zend\Filter\HtmlEntities(array('quotestyle' => ENT_NOQUOTES));

$input  = "A 'single' and " . '"double"';
print $filter->filter($input);
```

The above example returns `A 'single' and "double"`. Notice that neither `"double"` or `'single'` quotes are altered.

Helper Methods

To change or retrieve the `quotestyle` after instantiation, the two methods `setQuoteStyle()` and `getQuoteStyle()` may be used respectively. `setQuoteStyle()` accepts one parameter `$quoteStyle`. The following constants are accepted: `ENT_COMPAT`, `ENT_QUOTES`, `ENT_NOQUOTES`

```
$filter = new Zend\Filter\HtmlEntities();

$filter->setQuoteStyle(ENT_QUOTES);
print $filter->getQuoteStyle(ENT_QUOTES);
```

To change or retrieve the `charset` after instantiation, the two methods `setCharSet()` and `getCharSet()` may be used respectively. `setCharSet()` accepts one parameter `$charSet`. See "http://php.net/htmlentities" for a list of supported character sets.

```
$filter = new Zend\Filter\HtmlEntities();

$filter->setQuoteStyle(ENT_QUOTES);
print $filter->getQuoteStyle(ENT_QUOTES);
```

To change or retrieve the `doublequote` option after instantiation, the two methods `setDoubleQuote()` and `getDoubleQuote()` may be used respectively. `setDoubleQuote()` accepts one boolean parameter `$doubleQuote`.

```
$filter = new Zend\Filter\HtmlEntities();

$filter->setQuoteStyle(ENT_QUOTES);
print $filter->getQuoteStyle(ENT_QUOTES);
```

101.11 Int

`Zend\Filter\Int` allows you to transform a scalar value which contains into an integer.

Supported Options

There are no additional options for `Zend\Filter\Int`.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Int();

print $filter->filter('-4 is less than 0');
```

This will return '-4'.

101.12 Null

This filter will change the given input to be `NULL` if it meets specific criteria. This is often necessary when you work with databases and want to have a `NULL` value instead of a boolean or any other type.

Supported Options

The following options are supported for `Zend\Filter\Null`:

- **type**: The variable type which should be supported.

Default Behavior

Per default this filter works like *PHP*'s `empty()` method; in other words, if `empty()` returns a boolean `TRUE`, then a `NULL` value will be returned.

```
$filter = new Zend\Filter\Null();
$value  = '';
$result = $filter->filter($value);
// returns null instead of the empty string
```

This means that without providing any configuration, `Zend\Filter\Null` will accept all input types and return `NULL` in the same cases as `empty()`.

Any other value will be returned as is, without any changes.

Changing the Default Behavior

Sometimes it's not enough to filter based on `empty()`. Therefor `Zend\Filter\Null` allows you to configure which type will be converted and which not.

The following types can be handled:

- **boolean**: Converts a boolean **FALSE** value to `NULL`.
- **integer**: Converts an integer **0** value to `NULL`.
- **empty_array**: Converts an empty **array** to `NULL`.
- **float**: Converts an float **0.0** value to `NULL`.
- **string**: Converts an empty string **''** to `NULL`.
- **zero**: Converts a string containing the single character zero (**'0'**) to `NULL`.
- **all**: Converts all above types to `NULL`. (This is the default behavior.)

There are several ways to select which of the above types are filtered. You can give one or multiple types and add them, you can give an array, you can use constants, or you can give a textual string. See the following examples:

```
// converts false to null
$filter = new Zend\Filter\Null(Zend\Filter\Null::BOOLEAN);

// converts false and 0 to null
$filter = new Zend\Filter\Null(
    Zend\Filter\Null::BOOLEAN + Zend\Filter\Null::INTEGER
);

// converts false and 0 to null
$filter = new Zend\Filter\Null( array(
    Zend\Filter\Null::BOOLEAN,
    Zend\Filter\Null::INTEGER
));

// converts false and 0 to null
$filter = new Zend\Filter\Null(array(
    'boolean',
    'integer',
));
```

You can also give a Traversable or an array to set the wished types. To set types afterwards use `setType()`.

101.13 NumberFormat

The `NumberFormat` filter can be used to return locale-specific number and percentage strings. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl).

Supported Options for NumberFormat Filter

The following options are supported for `NumberFormat`:

`NumberFormat([ string $locale [, int $style [, int $type ]]])`

- `$locale`: (Optional) Locale in which the number would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

- `$style`: (Optional) Style of the formatting, one of the format style constants. If unset, it will use `NumberFormatter::DEFAULT_STYLE` as the default style.

  Methods for getting/setting the format style are also available: `getStyle()` and `setStyle()`

- `$type`: (Optional) The formatting type to use. If unset, it will use `NumberFormatter::TYPE_DOUBLE` as the default type.

  Methods for getting/setting the format type are also available: `getType()` and `setType()`

NumberFormat Filter Usage

```
$filter = new \Zend\I18n\Filter\NumberFormat("de_DE");
echo $filter->filter(1234567.8912346);
// Returns "1.234.567,891"

$filter = new \Zend\I18n\Filter\NumberFormat("en_US", NumberFormatter::PERCENT);
echo $filter->filter(0.80);
// Returns "80%"

$filter = new \Zend\I18n\Filter\NumberFormat("fr_FR", NumberFormatter::SCIENTIFIC);
echo $filter->filter(0.00123456789);
// Returns "1,23456789E-3"
```

101.14 PregReplace

`Zend\Filter\PregReplace` performs a search using regular expressions and replaces all found elements.

Supported Options

The following options are supported for `Zend\Filter\PregReplace`:

- **pattern**: The pattern which will be searched for.
- **replacement**: The string which is used as replacement for the matches.

Basic Usage

To use this filter properly you must give two options:

The option `pattern` has to be given to set the pattern which will be searched for. It can be a string for a single pattern, or an array of strings for multiple pattern.

To set the pattern which will be used as replacement the option `replacement` has to be used. It can be a string for a single pattern, or an array of strings for multiple pattern.

```
$filter = new Zend\Filter\PregReplace(array(
    'pattern'     => '/bob/',
    'replacement' => 'john',
));
```

```
$input  = 'Hy bob!";

$filter->filter($input);
// returns 'Hy john!'
```

You can use `getPattern()` and `setPattern()` to set the matching pattern afterwards. To set the replacement pattern you can use `getReplacement()` and `setReplacement()`.

```
$filter = new Zend\Filter\PregReplace();
$filter->setMatchPattern(array('bob', 'Hy'))
      ->setReplacement(array('john', 'Bye'));
$input  = 'Hy bob!";

$filter->filter($input);
// returns 'Bye john!'
```

For a more complex usage take a look into *PHP*'s PCRE Pattern Chapter.

101.15 RealPath

This filter will resolve given links and pathnames and returns canonicalized absolute pathnames.

Supported Options

The following options are supported for `Zend\Filter\RealPath`:

- **exists**: This option defaults to `TRUE` which checks if the given path really exists.

Basic Usage

For any given link of pathname its absolute path will be returned. References to '`/./`', '`/../`' and extra '`/`' characters in the input path will be stripped. The resulting path will not have any symbolic link, '`/./`' or '`/../`' character.

`Zend\Filter\RealPath` will return `FALSE` on failure, e.g. if the file does not exist. On *BSD* systems `Zend\Filter\RealPath` doesn't fail if only the last path component doesn't exist, while other systems will return `FALSE`.

```
$filter = new Zend\Filter\RealPath();
$path   = '/www/var/path/../../mypath';
$filtered = $filter->filter($path);

// returns '/www/mypath'
```

Non-Existing Paths

Sometimes it is useful to get also paths when they don't exist, f.e. when you want to get the real path for a path which you want to create. You can then either give a `FALSE` at initiation, or use `setExists()` to set it.

```
$filter = new Zend\Filter\RealPath(false);
$path   = '/www/var/path/../../non/existing/path';
$filtered = $filter->filter($path);

```

```
// returns '/www/non/existing/path'
// even when file_exists or realpath would return false
```

101.16 StringToLower

This filter converts any input to be lowercased.

Supported Options

The following options are supported for `Zend\Filter\StringToLower`:

- **encoding**: This option can be used to set an encoding which has to be used.

Basic Usage

This is a basic example:

```
$filter = new Zend\Filter\StringToLower();

print $filter->filter('SAMPLE');
// returns "sample"
```

Different Encoded Strings

Per default it will only handle characters from the actual locale of your server. Characters from other charsets would be ignored. Still, it's possible to also lowercase them when the mbstring extension is available in your environment. Simply set the wished encoding when initiating the `StringToLower` filter. Or use the `setEncoding()` method to change the encoding afterwards.

```
// using UTF-8
$filter = new Zend\Filter\StringToLower('UTF-8');

// or give an array which can be useful when using a configuration
$filter = new Zend\Filter\StringToLower(array('encoding' => 'UTF-8'));

// or do this afterwards
$filter->setEncoding('ISO-8859-1');
```

Note: Setting wrong encodings

Be aware that you will get an exception when you want to set an encoding and the mbstring extension is not available in your environment.

Also when you are trying to set an encoding which is not supported by your mbstring extension you will get an exception.

101.17 StringToUpper

This filter converts any input to be uppercased.

Supported Options

The following options are supported for `Zend\Filter\StringToUpper`:

- **encoding**: This option can be used to set an encoding which has to be used.

Basic Usage

This is a basic example for using the `StringToUpper` filter:

```
$filter = new Zend\Filter\StringToUpper();

print $filter->filter('Sample');
// returns "SAMPLE"
```

Different Encoded Strings

Like the `StringToLower` filter, this filter handles only characters from the actual locale of your server. Using different character sets works the same as with `StringToLower`.

```
$filter = new Zend\Filter\StringToUpper(array('encoding' => 'UTF-8'));

// or do this afterwards
$filter->setEncoding('ISO-8859-1');
```

101.18 StringTrim

This filter modifies a given string such that certain characters are removed from the beginning and end.

Supported Options

The following options are supported for `Zend\Filter\StringTrim`:

- **charlist**: List of characters to remove from the beginning and end of the string. If this is not set or is null, the default behavior will be invoked, which is to remove only whitespace from the beginning and end of the string.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\StringTrim();

print $filter->filter(' This is (my) content: ');
```

The above example returns 'This is (my) content:'. Notice that the whitespace characters have been removed.

Default Behavior

```
$filter = new Zend\Filter\StringTrim(':');
// or new Zend\Filter\StringTrim(array('charlist' => ':'));

print $filter->filter(' This is (my) content:');
```

The above example returns 'This is (my) content'. Notice that the whitespace characters and colon are removed. You can also provide a Traversable or an array with a 'charlist' key. To set the desired character list after instantiation, use the `setCharList()` method. The `getCharList()` return the values set for charlist.

101.19 StripNewLines

This filter modifies a given string and removes all new line characters within that string.

Supported Options

There are no additional options for `Zend\Filter\StripNewLines`:

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\StripNewLines();

print $filter->filter(' This is (my)``\n\r``content: ');
```

The above example returns 'This is (my) content:'. Notice that all newline characters have been removed.

101.20 StripTags

This filter can strip XML and HTML tags from given content.

> **Warning:** Be warned that Zend\Filter\StripTags should only be used to strip all available tags.
> Using Zend\Filter\StripTags to make your site secure by stripping some unwanted tags will lead to unsecure and dangerous code.
> Zend\Filter\StripTags must not be used to prevent XSS attacks. This filter is no replacement for using Tidy or HtmlPurifier.

Supported Options

The following options are supported for `Zend\Filter\StripTags`:

- **allowAttribs**: This option sets the attributes which are accepted. All other attributes are

stripped from the given content.

- **allowTags**: This option sets the tags which are accepted. All other tags will be stripped

from the given content.

Basic Usage

See the following example for the default behaviour of this filter:

```
$filter = new Zend\Filter\StripTags();

print $filter->filter('<B>My content</B>');
```

As result you will get the stripped content 'My content'.

When the content contains broken or partial tags then the complete following content will be erased. See the following example:

```
$filter = new Zend\Filter\StripTags();

print $filter->filter('This contains <a href="http://example.com">no ending tag');
```

The above will return 'This contains' with the rest being stripped.

Allowing Defined Tags

`Zend\Filter\StripTags` allows stripping of all but defined tags. This can be used for example to strip all tags but links from a text.

```
$filter = new Zend\Filter\StripTags(array('allowTags' => 'a'));

$input  = "A text with <br/> a <a href='link.com'>link</a>";
print $filter->filter($input);
```

The above will return 'A text with a <a href='link.com'>link</a>' as result. It strips all tags but the link. By providing an array you can set multiple tags at once.

> **Warning:** Do not use this feature to get a probably secure content. This component does not replace the use of a proper configured html filter.

Allowing Defined Attributes

It is also possible to strip all but allowed attributes from a tag.

```
$filter = new Zend\Filter\StripTags(array('allowAttribs' => 'src'));

$input  = "A text with <br/> a <img src='picture.com' width='100'>picture</img>";
print $filter->filter($input);
```

The above will return 'A text with a <img src='picture.com'>picture</img>' as result. It strips all tags but img. Additionally from the img tag all attributes but src will be stripped. By providing an array you can set multiple attributes at once.

101.21 UriNormalize

This filter can set a scheme on an URI, if a scheme is not present. If a scheme is present, that scheme will not be affected, even if a different scheme is enforced. The following options are supported for `Zend\Filter\UriNormalize`:

- **defaultScheme**: This option can be used to set the default scheme to use when parsing scheme-less URIs.
- **enforcedScheme**: Set a URI scheme to enforce on schemeless URIs.

Basic Usage

See the following example for the default behaviour of this filter:

```
$filter = new Zend\Filter\UriNormalize(array(
    'enforcedScheme' => 'https'
));

echo $filter->filter('www.example.com');
```

As the result the string `https://www.example.com` will be output.

CHAPTER

TWO

WORD FILTERS

In addition to the standard set of filters, there are several classes specific to filtering word strings.

102.1 CamelCaseToDash

This filter modifies a given string such that 'CamelCaseWords' are converted to 'camel-case-words'.

Supported Options

There are no additional options for `Zend\Filter\Word\CamelCaseToDash`:

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\CamelCaseToDash();

print $filter->filter('ThisIsMyContent');
```

The above example returns 'this-is-my-content'.

102.2 CamelCaseToSeparator

This filter modifies a given string such that 'CamelCaseWords' are converted to 'camel case words'.

Supported Options

The following options are supported for `Zend\Filter\Word\CamelCaseToSeparator`:

- **separator**: A separator char. If this is not set the separator will be a space character.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\CamelCaseToSeparator(':');
// or new Zend\Filter\Word\CamelCaseToSeparator(array('separator' => ':'));

print $filter->filter('ThisIsMyContent');
```

The above example returns 'this:is:my:content'.

Default Behavior

```
$filter = new Zend\Filter\Word\CamelCaseToSeparator();

print $filter->filter('ThisIsMyContent');
```

The above example returns 'this is my content'.

102.3 CamelCaseToUnderscore

This filter modifies a given string such that 'CamelCaseWords' are converted to 'camel_case_words'.

Supported Options

There are no additional options for `Zend\Filter\Word\CamelCaseToUnderscore`:

Basic usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\CamelCaseToUnderscore();

print $filter->filter('ThisIsMyContent');
```

The above example returns 'this_is_my_content'.

102.4 DashToCamelCase

This filter modifies a given string such that 'words-with-dashes' are converted to 'WordsWithDashes'.

Supported Options

There are no additional options for `Zend\Filter\Word\DashToCamelCase`:

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\DashToCamelCase();

print $filter->filter('this-is-my-content');
```

The above example returns 'ThisIsMyContent'.

102.5 DashToSeparator

This filter modifies a given string such that 'words-with-dashes' are converted to 'words with dashes'.

Supported Options

The following options are supported for `Zend\Filter\Word\DashToSeparator`:

- **separator**: A separator char. If this is not set the separator will be a space character.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\DashToSeparator('+');
// or new Zend\Filter\Word\CamelCaseToSeparator(array('separator' => '+'));

print $filter->filter('this-is-my-content');
```

The above example returns 'this+is+my+content'.

Default Behavior

```
$filter = new Zend\Filter\Word\DashToSeparator();

print $filter->filter('this-is-my-content');
```

The above example returns 'this is my content'.

102.6 DashToUnderscore

This filter modifies a given string such that 'words-with-dashes' are converted to 'words_with_dashes'.

Supported Options

There are no additional options for `Zend\Filter\Word\DashToUnderscore`:

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\DashToUnderscore();

print $filter->filter('this-is-my-content');
```

The above example returns 'this_is_my_content'.

102.7 SeparatorToCamelCase

This filter modifies a given string such that 'words with separators' are converted to 'WordsWithSeparators'.

Supported Options

The following options are supported for `Zend\Filter\Word\SeparatorToCamelCase`:

- **separator**: A separator char. If this is not set the separator will be a space character.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\SeparatorToCamelCase(':');
// or new Zend\Filter\Word\SeparatorToCamelCase(array('separator' => ':'));

print $filter->filter('this:is:my:content');
```

The above example returns 'ThisIsMyContent'.

Default Behavior

```
$filter = new Zend\Filter\Word\SeparatorToCamelCase();

print $filter->filter('this is my content');
```

The above example returns 'ThisIsMyContent'.

102.8 SeparatorToDash

This filter modifies a given string such that 'words with separators' are converted to 'words-with-separators'.

Supported Options

The following options are supported for `Zend\Filter\Word\SeparatorToDash`:

- **separator**: A separator char. If this is not set the separator will be a space character.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\SeparatorToDash(':');
// or new Zend\Filter\Word\SeparatorToDash(array('separator' => ':'));

print $filter->filter('this:is:my:content');
```

The above example returns 'this-is-my-content'.

Default Behavior

```
$filter = new Zend\Filter\Word\SeparatorToDash();

print $filter->filter('this is my content');
```

The above example returns 'this-is-my-content'.

102.9 SeparatorToSeparator

This filter modifies a given string such that 'words with separators' are converted to 'words-with-separators'.

Supported Options

The following options are supported for `Zend\Filter\Word\SeparatorToSeparator`:

- **searchSeparator**: The search separator char. If this is not set the separator will be a space character.
- **replaceSeparator**: The replace separator char. If this is not set the separator will be a dash.

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\SeparatorToSeparator(':', '+');

print $filter->filter('this:is:my:content');
```

The above example returns 'this+is+my+content'.

Default Behaviour

```
$filter = new Zend\Filter\Word\SeparatorToSeparator();

print $filter->filter('this is my content');
```

The above example returns 'this-is-my-content'.

102.10 UnderscoreToCamelCase

This filter modifies a given string such that 'words_with_underscores' are converted to 'WordsWithUnderscores'.

Supported Options

There are no additional options for `Zend\Filter\Word\UnderscoreToCamelCase`:

Basic Usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\UnderscoreToCamelCase();

print $filter->filter('this_is_my_content');
```

The above example returns 'ThisIsMyContent'.

102.11 UnderscoreToSeparator

This filter modifies a given string such that 'words_with_underscores' are converted to 'words with underscores'.

Supported Options

The following options are supported for `Zend\Filter\Word\UnderscoreToSeparator`:

- **separator**: A separator char. If this is not set the separator will be a space character.

Basic usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\UnderscoreToSeparator('+');
// or new Zend\Filter\Word\CamelCaseToSeparator(array('separator' => '+'));

print $filter->filter('this_is_my_content');
```

The above example returns 'this+is+my+content'.

Default Behavior

```
$filter = new Zend\Filter\Word\UnderscoreToSeparator();

print $filter->filter('this_is_my_content');
```

The above example returns 'this is my content'.

102.12 UnderscoreToDash

This filter modifies a given string such that 'words_with_underscores' are converted to 'words-with-underscores'.

Supported Options

There are no additional options for `Zend\Filter\Word\UnderscoreToDash`:

Basic usage

A basic example of usage is below:

```
$filter = new Zend\Filter\Word\UnderscoreToDash();

print $filter->filter('this_is_my_content');
```

The above example returns 'this-is-my-content'.

CHAPTER

THREE

FILE FILTER CLASSES

Zend Framework comes with a set of classes for filtering file contents as well as performing other actions, such as file renaming.

Note: All of the File Filter Classes' `filter()` methods support both a file path string *or* a `$_FILES` array as the supplied argument. When a `$_FILES` array is passed in, the `tmp_name` is used for the file path.

103.1 Decrypt

TODO

103.2 Encrypt

TODO

103.3 Lowercase

TODO

103.4 Rename

`Zend\Filter\File\Rename` can be used to rename a file and/or move a file to a new path.

Supported Options

The following set of options are supported:

- **target (string) default: "*"** Target filename or directory, the new name of the source file.
- **source (string) default: "*"** Source filename or directory which will be renamed.

  Used to match the filtered file with an options set.

- **overwrite (boolean) default: false** Shall existing files be overwritten?

  If the file is unable to be moved into the target path, a `Zend\Filter\Exception\RuntimeException` will be thrown.

- **randomize (boolean) default: false** Shall target files have a random postfix attached? The random postfix will be a `uniqid('_')` after the file name and before the extension.

  For example, `"file.txt"` will be randomized to `"file_4b3403665fea6.txt"`

An array of option sets is also supported, where a single `Rename` filter instance can filter several files using different options. The options used for the filtered file will be matched from the `source` option in the options set.

Usage Examples

Move all filtered files to a different directory:

```
// 'target' option is assumed if param is a string
$filter = \Zend\Filter\File\Rename("/tmp/");
echo $filter->filter("./myfile.txt");
// File has been moved to "/tmp/myfile.txt"
```

Rename all filtered files to a new name:

```
$filter = \Zend\Filter\File\Rename("/tmp/newfile.txt");
echo $filter->filter("./myfile.txt");
// File has been renamed to "/tmp/newfile.txt"
```

Move to a new path and randomize file names:

```
$filter = \Zend\Filter\File\Rename(array(
    "target"    => "/tmp/newfile.txt",
    "randomize" => true,
));
echo $filter->filter("./myfile.txt");
// File has been renamed to "/tmp/newfile_4b3403665fea6.txt"
```

Configure different options for several possible source files:

```
$filter = \Zend\Filter\File\Rename(array(
    array(
        "source"    => "fileA.txt"
        "target"    => "/dest1/newfileA.txt",
        "overwrite" => true,
    ),
    array(
        "source"    => "fileB.txt"
        "target"    => "/dest2/newfileB.txt",
        "randomize" => true,
    ),
));
echo $filter->filter("fileA.txt");
// File has been renamed to "/dest1/newfileA.txt"
echo $filter->filter("fileB.txt");
// File has been renamed to "/dest2/newfileB_4b3403665fea6.txt"
```

Public Methods

The specific public methods for the `Rename` filter, besides the common `filter()` method, are as follows:

getFile()
Returns the files to rename and their new name and location

Return type `array`

setFile(*string|array $options*)
Sets the file options for renaming. Removes any previously set file options.

Parameters **$options** – See *Supported Options* section for more information.

addFile(*string|array $options*)
Adds file options for renaming to the current list of file options.

Parameters **$options** – See *Supported Options* section for more information.

103.5 RenameUpload

`Zend\Filter\File\RenameUpload` can be used to rename or move an uploaded file to a new path.

Supported Options

The following set of options are supported:

- **target (string) default: `"*"`** Target directory or full filename path.
- **overwrite (boolean) default: `false`** Shall existing files be overwritten?

  If the file is unable to be moved into the target path, a `Zend\Filter\Exception\RuntimeException` will be thrown.
- **randomize (boolean) default: `false`** Shall target files have a random postfix attached? The random postfix will be a `uniqid('_')` after the file name and before the extension.

  For example, `"file.txt"` will be randomized to `"file_4b3403665fea6.txt"`
- **use_upload_name (boolean) default: `false`** When true, this filter will use the $_FILES['name'] as the target filename. Otherwise, the default `target` rules and the `$_FILES['tmp_name']` will be used.

> **Warning:** Be **very** careful when using the `use_upload_name` option. For instance, extremely bad things could happen if you were to allow uploaded `.php` files (or other CGI files) to be moved into the `DocumentRoot`.
> It is generally a better idea to supply an internal filename to avoid security risks.

`RenameUpload` does not support an array of options like the‘‘Rename‘‘ filter. When filtering HTML5 file uploads with the `multiple` attribute set, all files will be filtered with the same option settings.

Usage Examples

Move all filtered files to a different directory:

```
use Zend\Http\PhpEnvironment\Request;

$request = new Request();
$files   = $request->getFiles();
// i.e. $files['my-upload']['tmp_name'] === '/tmp/php5Wx0aJ'
// i.e. $files['my-upload']['name'] === 'myfile.txt'

// 'target' option is assumed if param is a string
$filter = \Zend\Filter\File\RenameUpload("./data/uploads/");
echo $filter->filter($files['my-upload']);
// File has been moved to "./data/uploads/php5Wx0aJ"

// ... or retain the uploaded file name
$filter->setUseUploadName(true);
echo $filter->filter($files['my-upload']);
// File has been moved to "./data/uploads/myfile.txt"
```

Rename all filtered files to a new name:

```
use Zend\Http\PhpEnvironment\Request;

$request = new Request();
$files   = $request->getFiles();
// i.e. $files['my-upload']['tmp_name'] === '/tmp/php5Wx0aJ'

$filter = \Zend\Filter\File\Rename("./data/uploads/newfile.txt");
echo $filter->filter($files['my-upload']);
// File has been renamed to "./data/uploads/newfile.txt"
```

Move to a new path and randomize file names:

```
use Zend\Http\PhpEnvironment\Request;

$request = new Request();
$files   = $request->getFiles();
// i.e. $files['my-upload']['tmp_name'] === '/tmp/php5Wx0aJ'

$filter = \Zend\Filter\File\Rename(array(
    "target"    => "./data/uploads/newfile.txt",
    "randomize" => true,
));
echo $filter->filter($files['my-upload']);
// File has been renamed to "./data/uploads/newfile_4b3403665fea6.txt"
```

103.6 Uppercase

TODO

CHAPTER

FOUR

FILTER CHAINS

Often multiple filters should be applied to some value in a particular order. For example, a login form accepts a username that should be only lowercase, alphabetic characters. `Zend\Filter\FilterChain` provides a simple method by which filters may be chained together. The following code illustrates how to chain together two filters for the submitted username:

```
// Create a filter chain and add filters to the chain
$filterChain = new Zend\Filter\FilterChain();
$filterChain->attach(new Zend\Filter\Alpha())
            ->attach(new Zend\Filter\StringToLower());

// Filter the username
$username = $filterChain->filter($_POST['username']);
```

Filters are run in the order they were added to `Zend\Filter\FilterChain`. In the above example, the username is first removed of any non-alphabetic characters, and then any uppercase characters are converted to lowercase.

Any object that implements `Zend\Filter\FilterInterface` may be used in a filter chain.

104.1 Setting Filter Chain Order

For each filter added to the `FilterChain` you can set a priority to define the chain order. The default value is `1000`. In the following example, any uppercase characters are converted to lowercase before any non-alphabetic characters are removed.

```
// Create a filter chain and add filters to the chain
$filterChain = new Zend\Filter\FilterChain();
$filterChain->attach(new Zend\Filter\Alpha())
            ->attach(new Zend\Filter\StringToLower(), 500);
```

104.2 Using the Plugin Manager

To every `FilterChain` object an instance of the `FilterPluginManager` is attached. Every filter that is used in a `FilterChain` must be know by this `FilterPluginManager`. To add a filter that is known by the `FilterPluginManager` you can use the `attachByName()` method. The first parameter is the name of the filter within the `FilterPluginManager`. The second parameter takes any options for creating the filter instance. The third parameter is the priority.

```
// Create a filter chain and add filters to the chain
$filterChain = new Zend\Filter\FilterChain();
$filterChain->attachByName('alpha')
            ->attachByName('stringtolower', array('encoding' => 'utf-8'), 500);
```

The following example shows how to add a custom filter to the `FilterPluginManager` and the `FilterChain`.

```
$filterChain = new Zend\Filter\FilterChain();
$filterChain->getPluginManager()->setInvokableClass(
    'myNewFilter', 'MyCustom\Filter\MyNewFilter'
);
$filterChain->attachByName('alpha')
            ->attachByName('myNewFilter');
```

You can also add your own `FilterPluginManager` implementation.

```
$filterChain = new Zend\Filter\FilterChain();
$filterChain->setPluginManager(new MyFilterPluginManager());
$filterChain->attach(new Zend\Filter\Alpha())
            ->attach(new MyCustom\Filter\MyNewFilter());
```

CHAPTER

FIVE

ZEND\FILTER\INFLECTOR

`Zend\Filter\Inflector` is a general purpose tool for rules-based inflection of strings to a given target.

As an example, you may find you need to transform MixedCase or camelCasedWords into a path; for readability, OS policies, or other reasons, you also need to lower case this, and you want to separate the words using a dash ('-'). An inflector can do this for you.

`Zend\Filter\Inflector` implements `Zend\Filter\FilterInterface`; you perform inflection by calling `filter()` on the object instance.

Transforming MixedCase and camelCaseText to another format

```
$inflector = new Zend\Filter\Inflector('pages/:page.:suffix');
$inflector->setRules(array(
    ':page'  => array('Word\CamelCaseToDash', 'StringToLower'),
    'suffix' => 'html',
));

$string   = 'camelCasedWords';
$filtered = $inflector->filter(array('page' => $string));
// pages/camel-cased-words.html

$string   = 'this_is_not_camel_cased';
$filtered = $inflector->filter(array('page' => $string));
// pages/this_is_not_camel_cased.html
```

105.1 Operation

An inflector requires a **target** and one or more **rules**. A target is basically a string that defines placeholders for variables you wish to substitute. These are specified by prefixing with a ':': **:script**.

When calling `filter()`, you then pass in an array of key and value pairs corresponding to the variables in the target.

Each variable in the target can have zero or more rules associated with them. Rules may be either **static** or refer to a `Zend\Filter` class. Static rules will replace with the text provided. Otherwise, a class matching the rule provided will be used to inflect the text. Classes are typically specified using a short name indicating the filter name stripped of any common prefix.

As an example, you can use any `Zend\Filter` concrete implementations; however, instead of referring to them as '`Zend\Filter\Alpha`' or '`Zend\Filter\StringToLower`', you'd specify only '`Alpha`' or '`StringToLower`'.

105.2 Using Custom Filters

`Zend\Filter\Inflector` uses `Zend\Filter\FilterPluginManager` to manage loading filters to use with inflection. By default, filters registered with `Zend\Filter\FilterPluginManager` are available. To access filters with that prefix but which occur deeper in the hierarchy, such as the various `Word` filters, simply strip off the `Zend\Filter` prefix:

```
// use Zend\Filter\Word\CamelCaseToDash as a rule
$inflector->addRules(array('script' => 'Word\CamelCaseToDash'));
```

To use custom filters, you have two choices: reference them by fully qualified class name (e.g., `My\Custom\Filter\Mungify`), or manipulate the composed `FilterPluginManager` instance.

```
$filters = $inflector->getPluginManager();
$filters->addInvokableClass('mungify', 'My\Custom\Filter\Mungify');
```

105.3 Setting the Inflector Target

The inflector target is a string with some placeholders for variables. Placeholders take the form of an identifier, a colon (':') by default, followed by a variable name: ':script', ':path', etc. The `filter()` method looks for the identifier followed by the variable name being replaced.

You can change the identifier using the `setTargetReplacementIdentifier()` method, or passing it as the third argument to the constructor:

```
// Via constructor:
$inflector = new Zend\Filter\Inflector('#foo/#bar.#sfx', null, '#');

// Via accessor:
$inflector->setTargetReplacementIdentifier('#');
```

Typically, you will set the target via the constructor. However, you may want to re-set the target later. `setTarget()` can be used for this purpose:

```
$inflector->setTarget('layouts/:script.phtml');
```

Additionally, you may wish to have a class member for your class that you can use to keep the inflector target updated – without needing to directly update the target each time (thus saving on method calls). `setTargetReference()` allows you to do this:

```
class Foo
{
    /**
     * @var string Inflector target
     */
    protected $_target = 'foo/:bar/:baz.:suffix';

    /**
     * Constructor
     * @return void
     */
    public function __construct()
    {
        $this->_inflector = new Zend\Filter\Inflector();
        $this->_inflector->setTargetReference($this->_target);
    }
```

```

    /**
     * Set target; updates target in inflector
     *
     * @param  string $target
     * @return Foo
     */
    public function setTarget($target)
    {
        $this->_target = $target;
        return $this;
    }
}
```

105.4 Inflection Rules

As mentioned in the introduction, there are two types of rules: static and filter-based.

Note: It is important to note that regardless of the method in which you add rules to the inflector, either one-by-one, or all-at-once; the order is very important. More specific names, or names that might contain other rule names, must be added before least specific names. For example, assuming the two rule names 'moduleDir' and 'module', the 'moduleDir' rule should appear before module since 'module' is contained within 'moduleDir'. If 'module' were added before 'moduleDir', 'module' will match part of 'moduleDir' and process it leaving 'Dir' inside of the target uninflected.

105.4.1 Static Rules

Static rules do simple string substitution; use them when you have a segment in the target that will typically be static, but which you want to allow the developer to modify. Use the `setStaticRule()` method to set or modify the rule:

```
$inflector = new Zend\Filter\Inflector(':script.:suffix');
$inflector->setStaticRule('suffix', 'phtml');

// change it later:
$inflector->setStaticRule('suffix', 'php');
```

Much like the target itself, you can also bind a static rule to a reference, allowing you to update a single variable instead of require a method call; this is often useful when your class uses an inflector internally, and you don't want your users to need to fetch the inflector in order to update it. The `setStaticRuleReference()` method is used to accomplish this:

```
class Foo
{
    /**
     * @var string Suffix
     */
    protected $_suffix = 'phtml';

    /**
     * Constructor
     * @return void
     */
    public function __construct()
```

```
    {
        $this->_inflector = new Zend\Filter\Inflector(':script.:suffix');
        $this->_inflector->setStaticRuleReference('suffix', $this->_suffix);
    }

    /**
     * Set suffix; updates suffix static rule in inflector
     *
     * @param  string $suffix
     * @return Foo
     */
    public function setSuffix($suffix)
    {
        $this->_suffix = $suffix;
        return $this;
    }
}
```

105.4.2 Filter Inflector Rules

`Zend\Filter` filters may be used as inflector rules as well. Just like static rules, these are bound to a target variable; unlike static rules, you may define multiple filters to use when inflecting. These filters are processed in order, so be careful to register them in an order that makes sense for the data you receive.

Rules may be added using `setFilterRule()` (which overwrites any previous rules for that variable) or `addFilterRule()` (which appends the new rule to any existing rule for that variable). Filters are specified in one of the following ways:

- **String**. The string may be a filter class name, or a class name segment minus any prefix set in the inflector's plugin loader (by default, minus the '`Zend\Filter`' prefix).
- **Filter object**. Any object instance implementing `Zend\Filter\FilterInterface` may be passed as a filter.
- **Array**. An array of one or more strings or filter objects as defined above.

```
$inflector = new Zend\Filter\Inflector(':script.:suffix');

// Set rule to use Zend\Filter\Word\CamelCaseToDash filter
$inflector->setFilterRule('script', 'Word\CamelCaseToDash');

// Add rule to lowercase string
$inflector->addFilterRule('script', new Zend\Filter\StringToLower());

// Set rules en-masse
$inflector->setFilterRule('script', array(
    'Word\CamelCaseToDash',
    new Zend\Filter\StringToLower()
));
```

105.4.3 Setting Many Rules At Once

Typically, it's easier to set many rules at once than to configure a single variable and its inflection rules at a time. `Zend\Filter\Inflector`'s `addRules()` and `setRules()` method allow this.

Each method takes an array of variable and rule pairs, where the rule may be whatever the type of rule accepts (string, filter object, or array). Variable names accept a special notation to allow setting static rules and filter rules, according to the following notation:

- **':' prefix**: filter rules.
- **No prefix**: static rule.

Setting Multiple Rules at Once

```
// Could also use setRules() with this notation:
$inflector->addRules(array(
    // filter rules:
    ':controller' => array('CamelCaseToUnderscore','StringToLower'),
    ':action'     => array('CamelCaseToUnderscore','StringToLower'),

    // Static rule:
    'suffix'      => 'phtml'
));
```

105.5 Utility Methods

`Zend\Filter\Inflector` has a number of utility methods for retrieving and setting the plugin loader, manipulating and retrieving rules, and controlling if and when exceptions are thrown.

- `setPluginManager()` can be used when you have configured your own `Zend\Filter\FilterPluginManager` instance and wish to use it with `Zend\Filter\Inflector`; `getPluginManager()` retrieves the currently set one.
- `setThrowTargetExceptionsOn()` can be used to control whether or not `filter()` throws an exception when a given replacement identifier passed to it is not found in the target. By default, no exceptions are thrown. `isThrowTargetExceptionsOn()` will tell you what the current value is.
- `getRules($spec = null)` can be used to retrieve all registered rules for all variables, or just the rules for a single variable.
- `getRule($spec, $index)` fetches a single rule for a given variable; this can be useful for fetching a specific filter rule for a variable that has a filter chain. `$index` must be passed.
- `clearRules()` will clear all currently registered rules.

105.6 Using a Traversable or an array with Zend\Filter\Inflector

You can use a `Traversable` or an array to set rules and other object state in your inflectors, either by passing a `Traversable` object or an array to the constructor or `setOptions()`. The following settings may be specified:

- `target` specifies the inflection target.
- `pluginManager` specifies the `Zend\Filter\FilterPluginManager` instance or extension to use for obtaining plugins; alternately, you may specify a class name of a class that extends the `FilterPluginManager`.
- `throwTargetExceptionsOn` should be a boolean indicating whether or not to throw exceptions when a replacement identifier is still present after inflection.

- `targetReplacementIdentifier` specifies the character to use when identifying replacement variables in the target string.
- `rules` specifies an array of inflection rules; it should consist of keys that specify either values or arrays of values, consistent with `addRules()`.

Using a Traversable or an array with `Zend\Filter\Inflector`

```
// With the constructor:
$options; // implements Traversable
$inflector = new Zend\Filter\Inflector($options);

// Or with setOptions():
$inflector = new Zend\Filter\Inflector();
$inflector->setOptions($options);
```

CHAPTER

SIX

WRITING FILTERS

`Zend\Filter` supplies a set of commonly needed filters, but developers will often need to write custom filters for their particular use cases. The task of writing a custom filter is facilitated by implementing `Zend\Filter\FilterInterface`.

`Zend\Filter\FilterInterface` defines a single method, `filter()`, that may be implemented by user classes.

The following example demonstrates how to write a custom filter:

```
namespace Application\Filter;

use Zend\Filter\FilterInterface;

class MyFilter implements FilterInterface
{
    public function filter($value)
    {
        // perform some transformation upon $value to arrive on $valueFiltered

        return $valueFiltered;
    }
}
```

To attach an instance of the filter defined above to a filter chain:

```
$filterChain = new Zend\Filter\FilterChain();
$filterChain->attach(new Application\Filter\MyFilter());
```

CHAPTER

SEVEN

INTRODUCTION TO ZEND\FORM

`Zend\Form` is intended primarily as a bridge between your domain models and the View Layer. It composes a thin layer of objects representing form elements, an *InputFilter*, and a small number of methods for binding data to and from the form and attached objects.

The `Zend\Form` component consists of the following objects:

- `Elements`, which simply consist of a name and attributes.
- `Fieldsets`, which extend from `Elements`, but allow composing other fieldsets and elements.
- `Forms`, which extend from `Fieldsets` (and thus `Elements`). They provide data and object binding, and compose *InputFilters*. Data binding is done via *ZendStdlibHydrator*.

To facilitate usage with the view layer, the `Zend\Form` component also aggregates a number of form-specific view helpers. These accept elements, fieldsets, and/or forms, and use the attributes they compose to render markup.

A small number of specialized elements are provided for accomplishing application-centric tasks. These include the `Csrf` element, used to prevent Cross Site Request Forgery attacks, and the `Captcha` element, used to display and validate *CAPTCHAs*.

A `Factory` is provided to facilitate creation of elements, fieldsets, forms, and the related input filter. The default `Form` implementation is backed by a factory to facilitate extension and ease the process of form creation.

The code related to forms can often spread between a variety of concerns: a form definition, an input filter definition, a domain model class, and one or more hydrator implementations. As such, finding the various bits of code and how they relate can become tedious. To simplify the situation, you can also annotate your domain model class, detailing the various input filter definitions, attributes, and hydrators that should all be used together. `Zend\Form\Annotation\AnnotationBuilder` can then be used to build the various objects you need.

CHAPTER

EIGHT

FORM QUICK START

Forms are relatively easy to create. At the bare minimum, each element or fieldset requires a name; typically, you'll also provide some attributes to hint to the view layer how it might render the item. The form itself will also typically compose an `InputFilter`– which you can also conveniently create directly in the form via a factory. Individual elements can hint as to what defaults to use when generating a related input for the input filter.

Form validation is as easy as providing an array of data to the `setData()` method. If you want to simplify your work even more, you can bind an object to the form; on successful validation, it will be populated from the validated values.

108.1 Programmatic Form Creation

If nothing else, you can simply start creating elements, fieldsets, and forms and wiring them together.

```
use Zend\Captcha;
use Zend\Form\Element;
use Zend\Form\Fieldset;
use Zend\Form\Form;
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;

$name = new Element('name');
$name->setLabel('Your name');
$name->setAttributes(array(
    'type'  => 'text'
));

$email = new Element\Email('email');
$email->setLabel('Your email address');

$subject = new Element('subject');
$subject->setLabel('Subject');
$subject->setAttributes(array(
    'type'  => 'text'
));

$message = new Element\Textarea('message');
$message->setLabel('Message');

$captcha = new Element\Captcha('captcha');
$captcha->setCaptcha(new Captcha\Dumb());
$captcha->setLabel('Please verify you are human');
```

```

$csrf = new Element\Csrf('security');

$send = new Element('send');
$send->setValue('Submit');
$send->setAttributes(array(
    'type'  => 'submit'
));


$form = new Form('contact');
$form->add($name);
$form->add($email);
$form->add($subject);
$form->add($message);
$form->add($captcha);
$form->add($csrf);
$form->add($send);

$nameInput = new Input('name');
// configure input... and all others
$inputFilter = new InputFilter();
// attach all inputs

$form->setInputFilter($inputFilter);
```

As a demonstration of fieldsets, let's alter the above slightly. We'll create two fieldsets, one for the sender information, and another for the message details.

```
$sender = new Fieldset('sender');
$sender->add($name);
$sender->add($email);

$details = new Fieldset('details');
$details->add($subject);
$details->add($message);

$form = new Form('contact');
$form->add($sender);
$form->add($details);
$form->add($captcha);
$form->add($csrf);
$form->add($send);
```

Regardless of approach, as you can see, this can be tedious.

108.2 Creation via Factory

You can create the entire form, and input filter, using the `Factory`. This is particularly nice if you want to store your forms as pure configuration; you can simply pass the configuration to the factory and be done.

```
use Zend\Form\Factory;

$factory = new Factory();
$form    = $factory->createForm(array(
    'hydrator' => 'Zend\Stdlib\Hydrator\ArraySerializable',
    'elements' => array(
```

```
        array(
            'spec' => array(
                'name' => 'name',
                'options' => array(
                    'label' => 'Your name',
                ),
                'type'  => 'Text',
            )
        ),
        array(
            'spec' => array(
                'type' => 'Zend\Form\Element\Email',
                'name' => 'email',
                'options' => array(
                    'label' => 'Your email address',
                )
            ),
        ),
        array(
            'spec' => array(
                'name' => 'subject',
                'options' => array(
                    'label' => 'Subject',
                ),
                'type'  => 'Text',
            ),
        ),
        array(
            'spec' => array(
                'type' => 'Zend\Form\Element\Textarea',
                'name' => 'message',
                'options' => array(
                    'label' => 'Message',
                )
            ),
        ),
        array(
            'spec' => array(
                'type' => 'Zend\Form\Element\Captcha',
                'name' => 'captcha',
                'options' => array(
                    'label' => 'Please verify you are human.',
                    'captcha' => array(
                        'class' => 'Dumb',
                    ),
                ),
            ),
        ),
        array(
            'spec' => array(
                'type' => 'Zend\Form\Element\Csrf',
                'name' => 'security',
            ),
        ),
        array(
            'spec' => array(
                'name' => 'send',
                'type'  => 'Submit',
```

```
                'attributes' => array(
                    'value' => 'Submit',
                ),
            ),
        ),
    ),
    /* If we had fieldsets, they'd go here; fieldsets contain
     * "elements" and "fieldsets" keys, and potentially a "type"
     * key indicating the specific FieldsetInterface
     * implementation to use.
    'fieldsets' => array(
    ),
     */

    // Configuration to pass on to
    // Zend\InputFilter\Factory::createInputFilter()
    'input_filter' => array(
        /* ... */
    ),
));
```

If we wanted to use fieldsets, as we demonstrated in the previous example, we could do the following:

```
use Zend\Form\Factory;

$factory = new Factory();
$form    = $factory->createForm(array(
    'hydrator'  => 'Zend\Stdlib\Hydrator\ArraySerializable'
    'fieldsets' => array(
        array(
            'name' => 'sender',
            'elements' => array(
                array(
                    'name' => 'name',
                    'options' => array(
                        'label' => 'Your name',
                        ),
                    'type'  => 'Text'
                ),
                array(
                    'type' => 'Zend\Form\Element\Email',
                    'name' => 'email',
                    'options' => array(
                        'label' => 'Your email address',
                        ),
                ),
            ),
        ),
        array(
            'name' => 'details',
            'elements' => array(
                array(
                    'name' => 'subject',
                    'options' => array(
                        'label' => 'Subject',
                        ),
                    'type'  => 'Text',
                ),
                array(
```

```
                    'name' => 'message',
                    'type' => 'Zend\Form\Element\Textarea',
                    'options' => array(
                        'label' => 'Message',
                        ),
                ),
            ),
        ),
    ),
    'elements' => array(
        array(
            'type' => 'Zend\Form\Element\Captcha',
            'name' => 'captcha',
            'options' => array(
                'label' => 'Please verify you are human. ',
                'captcha' => array(
                    'class' => 'Dumb',
                ),
            ),
        ),
        array(
            'type' => 'Zend\Form\Element\Csrf',
            'name' => 'security',
        ),
        array(
            'name' => 'send',
            'type'  => 'Submit',
            'attributes' => array(
                'value' => 'Submit',
            ),
        ),
    ),

    // Configuration to pass on to
    // Zend\InputFilter\Factory::createInputFilter()
    'input_filter' => array(
        /* ... */
    ),
));
```

Note that the chief difference is nesting; otherwise, the information is basically the same.

The chief benefits to using the `Factory` are allowing you to store definitions in configuration, and usage of significant whitespace.

108.3 Factory-backed Form Extension

The default `Form` implementation is backed by the `Factory`. This allows you to extend it, and define your form internally. This has the benefit of allowing a mixture of programmatic and factory-backed creation, as well as defining a form for re-use in your application.

```
namespace Contact;

use Zend\Captcha\AdapterInterface as CaptchaAdapter;
use Zend\Form\Element;
use Zend\Form\Form;

```

```
7  class ContactForm extends Form
8  {
9      protected $captcha;
10
11     public function __construct(CaptchaAdapter $captcha)
12     {
13         $this->captcha = $captcha;
14
15         // add() can take either an Element/Fieldset instance,
16         // or a specification, from which the appropriate object
17         // will be built.
18
19         $this->add(array(
20             'name' => 'name',
21             'options' => array(
22                 'label' => 'Your name',
23             ),
24             'type'  => 'Text',
25         ));
26         $this->add(array(
27             'type' => 'Zend\Form\Element\Email',
28             'name' => 'email',
29             'options' => array(
30                 'label' => 'Your email address',
31             ),
32         ));
33         $this->add(array(
34             'name' => 'subject',
35             'options' => array(
36                 'label' => 'Subject',
37             ),
38             'type'  => 'Text',
39         ));
40         $this->add(array(
41             'type' => 'Zend\Form\Element\Textarea',
42             'name' => 'message',
43             'options' => array(
44                 'label' => 'Message',
45             ),
46         ));
47         $this->add(array(
48             'type' => 'Zend\Form\Element\Captcha',
49             'name' => 'captcha',
50             'options' => array(
51                 'label' => 'Please verify you are human.',
52                 'captcha' => $this->captcha,
53             ),
54         ));
55         $this->add(new Element\Csrf('security'));
56         $this->add(array(
57             'name' => 'send',
58             'type'  => 'Submit',
59             'attributes' => array(
60                 'value' => 'Submit',
61             ),
62         ));
63
64         // We could also define the input filter here, or
```

```
        // lazy-create it in the getInputFilter() method.
    }
}
```

You'll note that this example, the elements are added in the constructor. This is done to allow altering and/or configuring either the form or input filter factory instances, which could then have bearing on how elements, inputs, etc. are created. In this case, it also allows injection of the CAPTCHA adapter, allowing us to configure it elsewhere in our application and inject it into the form.

108.4 Validating Forms

Validating forms requires three steps. First, the form must have an input filter attached. Second, you must inject the data to validate into the form. Third, you validate the form. If invalid, you can retrieve the error messages, if any.

```
$form = new Contact\ContactForm();

// If the form doesn't define an input filter by default, inject one.
$form->setInputFilter(new Contact\ContactFilter());

// Get the data. In an MVC application, you might try:
$data = $request->getPost();  // for POST data
$data = $request->getQuery(); // for GET (or query string) data

$form->setData($data);

// Validate the form
if ($form->isValid()) {
    $validatedData = $form->getData();
} else {
    $messages = $form->getMessages();
}
```

You can get the raw data if you want, by accessing the composed input filter.

```
$filter = $form->getInputFilter();

$rawValues    = $filter->getRawValues();
$nameRawValue = $filter->getRawValue('name');
```

108.5 Hinting to the Input Filter

Often, you'll create elements that you expect to behave in the same way on each usage, and for which you'll want specific filters or validation as well. Since the input filter is a separate object, how can you achieve these latter points?

Because the default form implementation composes a factory, and the default factory composes an input filter factory, you can have your elements and/or fieldsets hint to the input filter. If no input or input filter is provided in the input filter for that element, these hints will be retrieved and used to create them.

To do so, one of the following must occur. For elements, they must implement `Zend\InputFilter\InputProviderInterface`, which defines a `getInputSpecification()` method; for fieldsets, they must implement `Zend\InputFilter\InputFilterProviderInterface`, which defines a `getInputFilterSpecification()` method.

In the case of an element, the `getInputSpecification()` method should return data to be used by the input filter factory to create an input. Every HTML5 (email, url, color...) elements have a built-in element that use this logic. For instance, here is how the `Zend\Form\Element\Color` element is defined:

```
namespace Zend\Form\Element;

use Zend\Form\Element;
use Zend\InputFilter\InputProviderInterface;
use Zend\Validator\Regex as RegexValidator;
use Zend\Validator\ValidatorInterface;

/**
 * @category   Zend
 * @package    Zend_Form
 * @subpackage Element
 */
class Color extends Element implements InputProviderInterface
{
    /**
     * Seed attributes
     *
     * @var array
     */
    protected $attributes = array(
        'type' => 'color',
    );

    /**
     * @var ValidatorInterface
     */
    protected $validator;

    /**
     * Get validator
     *
     * @return ValidatorInterface
     */
    protected function getValidator()
    {
        if (null === $this->validator) {
            $this->validator = new RegexValidator('/^#[0-9a-fA-F]{6}$/');
        }
        return $this->validator;
    }

    /**
     * Provide default input rules for this element
     *
     * Attaches an email validator.
     *
     * @return array
     */
    public function getInputSpecification()
    {
        return array(
            'name' => $this->getName(),
            'required' => true,
            'filters' => array(
                array('name' => 'Zend\Filter\StringTrim'),
```

```
                    array('name' => 'Zend\Filter\StringToLower'),
                ),
                'validators' => array(
                    $this->getValidator(),
                ),
            );
        }
    }
```

The above would hint to the input filter to create and attach an input named after the element, marking it as required, and giving it a `StringTrim` and `StringToLower` filters and a `Regex` validator. Note that you can either rely on the input filter to create filters and validators, or directly instantiate them.

For fieldsets, you do very similarly; the difference is that `getInputFilterSpecification()` must return configuration for an input filter.

```
namespace Contact\Form;

use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;
use Zend\Validator;

class SenderFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
                'filters'  => array(
                    array('name' => 'Zend\Filter\StringTrim'),
                ),
            ),
            'email' => array(
                'required' => true,
                'filters'  => array(
                    array('name' => 'Zend\Filter\StringTrim'),
                ),
                'validators' => array(
                    new Validator\EmailAddress(),
                ),
            ),
        );
    }
}
```

Specifications are a great way to make forms, fieldsets, and elements re-usable trivially in your applications. In fact, the `Captcha` and `Csrf` elements define specifications in order to ensure they can work without additional user configuration!

108.6 Binding an object

As noted in the intro, forms in Zend Framework bridge the domain model and the view layer. Let's see that in action.

When you `bind()` an object to the form, the following happens:

- The composed `Hydrator` calls `extract()` on the object, and uses the values returned, if any, to populate

the `value` attributes of all elements. If a form contains a fieldset that itself contains another fieldset, the form will recursively extract the values.

- When `isValid()` is called, if `setData()` has not been previously set, the form uses the composed `Hydrator` to extract values from the object, and uses those during validation.
- If `isValid()` is successful (and the `bindOnValidate` flag is enabled, which is true by default), then the `Hydrator` will be passed the validated values to use to hydrate the bound object. (If you do not want this behavior, call `setBindOnValidate(FormInterface::BIND_MANUAL)`).
- If the object implements `Zend\InputFilter\InputFilterAwareInterface`, the input filter it composes will be used instead of the one composed on the form.

This is easier to understand in practice.

```
$contact = new ArrayObject;
$contact['subject'] = '[Contact Form] ';
$contact['message'] = 'Type your message here';

$form    = new Contact\ContactForm;

$form->bind($contact); // form now has default values for
                       // 'subject' and 'message'

$data = array(
    'name'    => 'John Doe',
    'email'   => 'j.doe@example.tld',
    'subject' => '[Contact Form] \'sup?',
);
$form->setData($data);

if ($form->isValid()) {
    // $contact now looks like:
    // array(
    //     'name'    => 'John Doe',
    //     'email'   => 'j.doe@example.tld',
    //     'subject' => '[Contact Form] \'sup?',
    //     'message' => 'Type your message here',
    // )
    // only as an ArrayObject
}
```

When an object is bound to the form, calling `getData()` will return that object by default. If you want to return an associative array instead, you can pass the `FormInterface::VALUES_AS_ARRAY` flag to the method.

```
use Zend\Form\FormInterface;
$data = $form->getData(FormInterface::VALUES_AS_ARRAY);
```

Zend Framework ships several standard *hydrators*, and implementation is as simple as implementing `Zend\Stdlib\Hydrator\HydratorInterface`, which looks like this:

```
namespace Zend\Stdlib\Hydrator;

interface HydratorInterface
{
    /** @return array */
    public function extract($object);
    public function hydrate(array $data, $object);
}
```

108.7 Rendering

As noted previously, forms are meant to bridge the domain model and view layer. We've discussed the domain model binding, but what about the view?

The form component ships a set of form-specific view helpers. These accept the various form objects, and introspect them in order to generate markup. Typically, they will inspect the attributes, but in special cases, they may look at other properties and composed objects.

When preparing to render, you will likely want to call `prepare()`. This method ensures that certain injections are done, and will likely in the future munge names to allow for `scoped[array][notation]`.

The simplest view helpers available are `Form`, `FormElement`, `FormLabel`, and `FormElementErrors`. Let's use them to display the contact form.

```
<?php
// within a view script
$form = $this->form;
$form->prepare();

// Assuming the "contact/process" route exists...
$form->setAttribute('action', $this->url('contact/process'));

// Set the method attribute for the form
$form->setAttribute('method', 'post');

// Get the form label plugin
$formLabel = $this->plugin('formLabel');

// Render the opening tag
echo $this->form()->openTag($form);
?>
<div class="form_element">
<?php
    $name = $form->get('name');
    echo $formLabel->openTag() . $name->getOption('label');
    echo $this->formInput($name);
    echo $this->formElementErrors($name);
    echo $formLabel->closeTag();
?></div>

<div class="form_element">
<?php
    $subject = $form->get('subject');
    echo $formLabel->openTag() . $subject->getOption('label');
    echo $this->formInput($subject);
    echo $this->formElementErrors($subject);
    echo $formLabel->closeTag();
?></div>

<div class="form_element">
<?php
    $message = $form->get('message');
    echo $formLabel->openTag() . $message->getOption('label');
    echo $this->formTextarea($message);
    echo $this->formElementErrors($message);
    echo $formLabel->closeTag();
?></div>

```

```
<div class="form_element">
<?php
    $captcha = $form->get('captcha');
    echo $formLabel->openTag() . $captcha->getOption('label');
    echo $this->formCaptcha($captcha);
    echo $this->formElementErrors($captcha);
    echo $formLabel->closeTag();
?></div>

<?php echo $this->formElement($form->get('security')) ?>
<?php echo $this->formElement($form->get('send')) ?>

<?php echo $this->form()->closeTag() ?>
```

There are a few things to note about this. First, to prevent confusion in IDEs and editors when syntax highlighting, we use helpers to both open and close the form and label tags. Second, there's a lot of repetition happening here; we could easily create a partial view script or a composite helper to reduce boilerplate. Third, note that not all elements are created equal – the CSRF and submit elements don't need labels or error messages necessarily. Finally, note that the `FormElement` helper tries to do the right thing – it delegates actual markup generation to other view helpers; however, it can only guess what specific form helper to delegate to based on the list it has. If you introduce new form view helpers, you'll need to extend the `FormElement` helper, or create your own.

However, your view files can quickly become long and repetitive to write. While we do not currently provide a single-line form view helper (as this reduces the form customization), the simplest and most recommended way to render your form is by using the `FormRow` view helper. This view helper automatically renders a label (if present), the element itself using the `FormElement` helper, as well as any errors that could arise. Here is the previous form, rewritten to take advantage of this helper :

```
<?php
// within a view script
$form = $this->form;
$form->prepare();

// Assuming the "contact/process" route exists...
$form->setAttribute('action', $this->url('contact/process'));

// Set the method attribute for the form
$form->setAttribute('method', 'post');

// Render the opening tag
echo $this->form()->openTag($form);
?>
<div class="form_element">
<?php
    $name = $form->get('name');
    echo $this->formRow($name);
?></div>

<div class="form_element">
<?php
    $subject = $form->get('subject');
    echo $this->formRow($subject);
?></div>

<div class="form_element">
<?php
    $message = $form->get('message');
    echo $this->formRow($message);
```

```
?></div>

<div class="form_element">
<?php
    $captcha = $form->get('captcha');
    echo $this->formRow($captcha);
?></div>

<?php echo $this->formElement($form->get('security')) ?>
<?php echo $this->formElement($form->get('send')) ?>

<?php echo $this->form()->closeTag() ?>
```

Note that `FormRow` helper automatically prepends the label. If you want it to be rendered after the element itself, you can pass an optional parameter to the `FormRow` view helper :

```
<div class="form_element">
<?php
    $name = $form->get('name');
    echo $this->formRow($name, **'append'**);
?></div>
```

108.7.1 Taking advantage of HTML5 input attributes

HTML5 brings a lot of exciting features, one of them being a simplified client form validations. Adding HTML5 attributes is simple as you just need to add specify the attributes. However, please note that adding those attributes does not automatically add Zend validators to the form's input filter. You still need to manually add them.

```
$form->add(array(
    'name' => 'phoneNumber',
    'options' => array(
        'label' => 'Your phone number'
    ),
    'attributes' => array(
        'type' => 'tel'
        'required' => 'required',
        'pattern'  => '^0[1-68]([-. ]?[0-9]{2}){4}$'
    )
));
```

View helpers will automatically render those attributes, and hence allowing modern browsers to perform automatic validation.

Note: Although client validation is nice from a user experience point of view, it has to be used in addition with server validation, as client validation can be easily fooled.

108.8 Validation Groups

Sometimes you want to validate only a subset of form elements. As an example, let's say we're re-using our contact form over a web service; in this case, the `Csrf`, `Captcha`, and submit button elements are not of interest, and shouldn't be validated.

`Zend\Form` provides a proxy method to the underlying `InputFilter`'s `setValidationGroup()` method, allowing us to perform this operation.

```
$form->setValidationGroup('name', 'email', 'subject', 'message');
$form->setData($data);
if ($form->isValid()) {
    // Contains only the "name", "email", "subject", and "message" values
    $data = $form->getData();
}
```

If you later want to reset the form to validate all, simply pass the `FormInterface::VALIDATE_ALL` flag to the `setValidationGroup()` method.

```
use Zend\Form\FormInterface;
$form->setValidationGroup(FormInterface::VALIDATE_ALL);
```

When your form contains nested fieldsets, you can use an array notation to validate only a subset of the fieldsets :

```
$form->setValidationGroup(array(
            'profile' => array(
            'firstname',
            'lastname'
            )
));
$form->setData($data);
if ($form->isValid()) {
    // Contains only the "firstname" and "lastname" values from the
    // "profile" fieldset
    $data = $form->getData();
}
```

108.9 Using Annotations

Creating a complete forms solution can often be tedious: you'll create some domain model object, an input filter for validating it, a form object for providing a representation for it, and potentially a hydrator for mapping the form elements and fieldsets to the domain model. Wouldn't it be nice to have a central place to define all of these?

Annotations allow us to solve this problem. You can define the following behaviors with the shipped annotations in `Zend\Form`:

- *AllowEmpty*: mark an input as allowing an empty value. This annotation does not require a value.
- *Attributes*: specify the form, fieldset, or element attributes. This annotation requires an associative array of values, in a JSON object format: `@Attributes({"class":"zend_form","type":"text"})`.
- *ComposedObject*: specify another object with annotations to parse. Typically, this is used if a property references another object, which will then be added to your form as an additional fieldset. Expects a string value indicating the class for the object being composed.
- *ErrorMessage*: specify the error message to return for an element in the case of a failed validation. Expects a string value.
- *Exclude*: mark a property to exclude from the form or fieldset. This annotation does not require a value.
- *Filter*: provide a specification for a filter to use on a given element. Expects an associative array of values, with a "name" key pointing to a string filter name, and an "options" key pointing to an associative array of filter options for the constructor: `@Filter({"name": "Boolean", "options": {"casting":true}})`. This annotation may be specified multiple times.
- *Flags*: flags to pass to the fieldset or form composing an element or fieldset; these are usually used to specify the name or priority. The annotation expects an associative array: `@Flags({"priority": 100})`.

- *Hydrator*: specify the hydrator class to use for this given form or fieldset. A string value is expected.
- *InputFilter*: specify the input filter class to use for this given form or fieldset. A string value is expected.
- *Input*: specify the input class to use for this given element. A string value is expected.
- *Name*: specify the name of the current element, fieldset, or form. A string value is expected.
- *Options*: options to pass to the fieldset or form that are used to inform behavior – things that are not attributes; e.g. labels, CAPTCHA adapters, etc. The annotation expects an associative array: `@Options({"label": "Username:"})`.
- *Required*: indicate whether an element is required. A boolean value is expected. By default, all elements are required, so this annotation is mainly present to allow disabling a requirement.
- *Type*: indicate the class to use for the current element, fieldset, or form. A string value is expected.
- *Validator*: provide a specification for a validator to use on a given element. Expects an associative array of values, with a "name" key pointing to a string validator name, and an "options" key pointing to an associative array of validator options for the constructor: `@Validator({"name": "StringLength", "options": {"min":3, "max": 25}})`. This annotation may be specified multiple times.

To use annotations, you simply include them in your class and/or property docblocks. Annotation names will be resolved according to the import statements in your class; as such, you can make them as long or as short as you want depending on what you import.

Note: Form annotations require `Doctrine\Common`, which contains an annotation parsering engine. The simplest way to install `Doctrine\Common` is if you are using `Composer`; simply update your `composer.json` and add the following line to the `require` section:

```
"doctrine/common": ">=2.1",
```

Then run `php composer.phar update` to install the dependency.

If you're not using `Composer`, visit the Doctrine project website for more details on installation.

Here's a simple example.

```
use Zend\Form\Annotation;

/**
 * @Annotation\Name("user")
 * @Annotation\Hydrator("Zend\Stdlib\Hydrator\ObjectProperty")
 */
class User
{
    /**
     * @Annotation\Exclude()
     */
    public $id;

    /**
     * @Annotation\Filter({"name":"StringTrim"})
     * @Annotation\Validator({"name":"StringLength", "options":{"min":1, "max":25}})
     * @Annotation\Validator({"name":"Regex", "options":{"pattern":"/^[a-zA-Z][a-zA-Z0-9_-]{0,24}$/"
     * @Annotation\Attributes({"type":"text"})
     * @Annotation\Options({"label":"Username:"})
     */
    public $username;

```

```
    /**
     * @Annotation\Type("Zend\Form\Element\Email")
     * @Annotation\Options({"label":"Your email address:"})
     */
    public $email;
}
```

The above will hint to the annotation build to create a form with name "user", which uses the hydrator `Zend\Stdlib\Hydrator\ObjectProperty`. That form will have two elements, "username" and "email". The "username" element will have an associated input that has a `StringTrim` filter, and two validators: a `StringLength` validator indicating the username is between 1 and 25 characters, and a `Regex` validator asserting it follows a specific accepted pattern. The form element itself will have an attribute "type" with value "text" (a text element), and a label "Username:". The "email" element will be of type `Zend\Form\Element\Email`, and have the label "Your email address:".

To use the above, we need `Zend\Form\Annotation\AnnotationBuilder`:

```
use Zend\Form\Annotation\AnnotationBuilder;

$builder = new AnnotationBuilder();
$form    = $builder->createForm('User');
```

At this point, you have a form with the appropriate hydrator attached, an input filter with the appropriate inputs, and all elements.

Note: You're not done

In all likelihood, you'll need to add some more elements to the form you construct. For example, you'll want a submit button, and likely a CSRF-protection element. We recommend creating a fieldset with common elements such as these that you can then attach to the form you build via annotations.

CHAPTER

NINE

FORM COLLECTIONS

Often, fieldsets or elements in your forms will correspond to other domain objects. In some cases, they may correspond to collections of domain objects. In this latter case, in terms of user interfaces, you may want to add items dynamically in the user interface – a great example is adding tasks to a task list.

This document is intended to demonstrate these features. To do so, we first need to define some domain objects that we'll be using.

```
namespace Application\Entity;

class Product
{
    /**
     * @var string
     */
    protected $name;

    /**
     * @var int
     */
    protected $price;

    /**
     * @var Brand
     */
    protected $brand;

    /**
     * @var array
     */
    protected $categories;

    /**
     * @param string $name
     * @return Product
     */
    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    /**
     * @return string
     */
```

```
    public function getName()
    {
        return $this->name;
    }

    /**
     * @param int $price
     * @return Product
     */
    public function setPrice($price)
    {
        $this->price = $price;
        return $this;
    }

    /**
     * @return int
     */
    public function getPrice()
    {
        return $this->price;
    }

    /**
     * @param Brand $brand
     * @return Product
     */
    public function setBrand(Brand $brand)
    {
        $this->brand = $brand;
        return $this;
    }

    /**
     * @return Brand
     */
    public function getBrand()
    {
        return $this->brand;
    }

    /**
     * @param array $categories
     * @return Product
     */
    public function setCategories(array $categories)
    {
        $this->categories = $categories;
        return $this;
    }

    /**
     * @return array
     */
    public function getCategories()
    {
        return $this->categories;
    }
```

```
}

class Brand
{
    /**
     * @var string
     */
    protected $name;

    /**
     * @var string
     */
    protected $url;

    /**
     * @param string $name
     * @return Brand
     */
    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    /**
     * @return string
     */
    public function getName()
    {
        return $this->name;
    }

    /**
     * @param string $url
     * @return Brand
     */
    public function setUrl($url)
    {
        $this->url = $url;
        return $this;
    }

    /**
     * @return string
     */
    public function getUrl()
    {
        return $this->url;
    }
}

class Category
{
    /**
     * @var string
     */
    protected $name;
```

```
    /**
     * @param string $name
     * @return Category
     */
    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    /**
     * @return string
     */
    public function getName()
    {
        return $this->name;
    }
}
```

As you can see, this is really simple code. A Product has two scalar properties (name and price), a OneToOne relationship (one product has one brand), and a OneToMany relationship (one product has many categories).

109.1 Creating Fieldsets

The first step is to create three fieldsets. Each fieldset will contain all the fields and relationships for a specific entity.

Here is the `Brand` fieldset:

```
namespace Application\Form;

use Application\Entity\Brand;
use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class BrandFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function __construct()
    {
        parent::__construct('brand');
        $this->setHydrator(new ClassMethodsHydrator(false))
            ->setObject(new Brand());

        $this->add(array(
            'name' => 'name',
            'options' => array(
                'label' => 'Name of the brand'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));

        $this->add(array(
            'name' => 'url',
            'type' => 'Zend\Form\Element\Url',
            'options' => array(
```

```
                'label' => 'Website of the brand'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));
    }

    /**
     * @return array
     */
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
            )
        );
    }
}
```

We can discover some new things here. As you can see, the fieldset calls the method `setHydrator()`, giving it a `ClassMethods` hydrator, and the `setObject()` method, giving it an empty instance of a concrete `Brand` object.

When the data will be validated, the `Form` will automatically iterate through all the field sets it contains, and automatically populate the sub-objects, in order to return a complete entity.

Also notice that the `Url` element has a type of `Zend\Form\Element\Url`. This information will be used to validate the input field. You don't need to manually add filters or validators for this input as that element provides a reasonable input specification.

Finally, `getInputFilterSpecification()` gives the specification for the remaining input ("name"), indicating that this input is required. Note that *required* in the array "attributes" (when elements are added) is only meant to add the "required" attribute to the form markup (and therefore has semantic meaning only).

Here is the `Category` fieldset:

```
namespace Application\Form;

use Application\Entity\Category;
use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class CategoryFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function __construct()
    {
        parent::__construct('category');
        $this->setHydrator(new ClassMethodsHydrator(false))
             ->setObject(new Category());

        $this->setLabel('Category');

        $this->add(array(
            'name' => 'name',
            'options' => array(
                'label' => 'Name of the category'
            ),
```

```
            'attributes' => array(
                'required' => 'required'
            )
        ));
    }

    /**
     * @return array
     */
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
            )
        );
    }
}
```

Nothing new here.

And finally the `Product` fieldset:

```
namespace Application\Form;

use Application\Entity\Product;
use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class ProductFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function __construct()
    {
        parent::__construct('product');
        $this->setHydrator(new ClassMethodsHydrator(false))
             ->setObject(new Product());

        $this->add(array(
            'name' => 'name',
            'options' => array(
                'label' => 'Name of the product'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));

        $this->add(array(
            'name' => 'price',
            'options' => array(
                'label' => 'Price of the product'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));

        $this->add(array(
```

```
            'type' => 'Application\Form\BrandFieldset',
            'name' => 'brand',
            'options' => array(
                'label' => 'Brand of the product'
            )
        ));

        $this->add(array(
            'type' => 'Zend\Form\Element\Collection',
            'name' => 'categories',
            'options' => array(
                'label' => 'Please choose categories for this product',
                'count' => 2,
                'should_create_template' => true,
                'allow_add' => true,
                'target_element' => array(
                    'type' => 'Application\Form\CategoryFieldset'
                )
            )
        ));
    }

    /**
     * Should return an array specification compatible with
     * {@link Zend\InputFilter\Factory::createInputFilter()}.
     *
     * @return array
     */
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
            ),
            'price' => array(
                'required' => true,
                'validators' => array(
                    array(
                        'name' => 'Float'
                    )
                )
            )
        );
    }
}
```

We have a lot of new things here!

First, notice how the brand element is added: we specify it to be of type `Application\Form\BrandFieldset`. This is how you handle a OneToOne relationship. When the form is validated, the `BrandFieldset` will first be populated, and will return a `Brand` entity (as we have specified a `ClassMethods` hydrator, and bound the fieldset to a `Brand` entity using the `setObject()` method). This `Brand` entity will then be used to populate the `Product` entity by calling the `setBrand()` method.

The next element shows you how to handle OneToMany relationship. The type is `Zend\Form\Element\Collection`, which is a specialized element to handle such cases. As you can see, the name of the element ("categories") perfectly matches the name of the property in the `Product` entity.

This element has a few interesting options:

- `count`: this is how many times the element (in this case a category) has to be rendered. We've set it to two in this examples.
- `should_create_template`: if set to `true`, it will generate a template markup in a `<span>` element, in order to simplify adding new element on the fly (we will speak about this one later).
- `allow_add`: if set to `true` (which is the default), dynamically added elements will be retrieved and validated; otherwise, they will be completely ignored. This, of course, depends on what you want to do.
- `target_element`: this is either an element or, as this is the case in this example, an array that describes the element or fieldset that will be used in the collection. In this case, the `target_element` is a `Category` fieldset.

109.2 The Form Element

So far, so good. We now have our field sets in place. But those are field sets, not forms. And only `Form` instances can be validated. So here is the form :

```
namespace Application\Form;

use Zend\Form\Form;
use Zend\InputFilter\InputFilter;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class CreateProduct extends Form
{
    public function __construct()
    {
        parent::__construct('create_product');

        $this->setAttribute('method', 'post')
             ->setHydrator(new ClassMethodsHydrator(false))
             ->setInputFilter(new InputFilter());

        $this->add(array(
            'type' => 'Application\Form\ProductFieldset',
            'options' => array(
                'use_as_base_fieldset' => true
            )
        ));

        $this->add(array(
            'type' => 'Zend\Form\Element\Csrf',
            'name' => 'csrf'
        ));

        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type' => 'submit',
                'value' => 'Send'
            )
        ));
    }
}
```

`CreateProduct` is quite simple, as it only defines a `Product` fieldset, as well as some other useful fields (`CSRF`

for security, and a `Submit` button).

Notice the `use_as_base_fieldset` option. This option is here to say to the form: "hey, the object I bind to you is, in fact, bound to the fieldset that is the base fieldset." This will be to true most of the times.

What's cool with this approach is that each entity can have its own `Fieldset` and can be reused. You describe the elements, the filters, and validators for each entity only once, and the concrete `Form` instance will only compose those fieldsets. You no longer have to add the "username" input to every form that deals with users!

109.3 The Controller

Now, let's create the action in the controller:

```
/**
  * @return array
  */
 public function indexAction()
 {
     $form = new CreateProduct();
     $product = new Product();
     $form->bind($product);

     $request = $this->getRequest();
     if ($request->isPost()) {
         $form->setData($request->getPost());

         if ($form->isValid()) {
             var_dump($product);
         }
     }

     return array(
         'form' => $form
     );
 }
```

This is super easy. Nothing to do in the controllers. All the magic is done behind the scene.

109.4 The View

And finally, the view:

```
<?php
$form->setAttribute('action', $this->url('home'))
     ->prepare();

echo $this->form()->openTag($form);

$product = $form->get('product');

echo $this->formRow($product->get('name'));
echo $this->formRow($product->get('price'));
echo $this->formCollection($product->get('categories'));

$brand = $product->get('brand');
```

```
echo $this->formRow($brand->get('name'));
echo $this->formRow($brand->get('url'));

echo $this->formHidden($form->get('csrf'));
echo $this->formElement($form->get('submit'));

echo $this->form()->closeTag();
```

A few new things here :

- the `prepare()` method. You *must* call it prior to rendering anything in the view (this function is only meant to be called in views, not in controllers).
- the `FormRow` helper renders a label (if present), the input itself, and errors.
- the `FormCollection` helper will iterate through every element in the collection, and render every element with the FormRow helper (you may specify an alternate helper if desired, using the `setElementHelper()` method on that `FormCollection` helper instance). If you need more control about the way you render your forms, you can iterate through the elements in the collection, and render them manually one by one.

Here is the result:

As you can see, collections are wrapped inside a fieldset, and every item in the collection is itself wrapped in the fieldset. In fact, the `Collection` element uses label for each item in the collection, while the label of the `Collection` element itself is used as the legend of the fieldset. You must have a label on every element in order to use this feature. If you don't want the fieldset created, but just the elements within it, simply add a boolean `false` as the second parameter of the `FormCollection` view helper.

If you validate, all elements will show errors (this is normal, as we've marked them as required). As soon as the form is valid, this is what we get :

As you can see, the bound object is completely filled, not with arrays, but with objects!

But that's not all.

109.5 Adding New Elements Dynamically

Remember the `should_create_template`? We are going to use it now.

Often, forms are not completely static. In our case, let's say that we don't want only two categories, but we want the user to be able to add other ones at runtime. `Zend\Form` has this capability. First, let's see what it generates when we ask it to create a template:

As you can see, the collection generates two fieldsets (the two categories) *plus* a span with a `data-template` attribute that contains the full HTML code to copy to create a new element in the collection. Of course `__index__` (this is the placeholder generated) has to be changed to a valid value. Currently, we have 2 elements (`categories[0]` and `categories[1]`, so `__index__` has to be changed to 2.

If you want, this placeholder (`__index__` is the default) can be changed using the `template_placeholder` option key:

```
$this->add(array(
    'type' => 'Zend\Form\Element\Collection',
    'name' => 'categories',
    'options' => array(
        'label' => 'Please choose categories for this product',
        'count' => 2,
        'should_create_template' => true,
```

```
        'template_placeholder' => '__placeholder__',
        'target_element' => array(
            'type' => 'Application\Form\CategoryFieldset'
        )
    )
));
```

First, let's add a small button "Add new category" anywhere in the form:

```
<button onclick="return add_category()">Add a new category</button>
```

The `add_category` function is fairly simple:

1. First, count the number of elements we already have.
2. Get the template from the `span`'s `data-template` attribute.
3. Change the placeholder to a valid index.
4. Add the element to the DOM.

Here is the code:

```
<script>
    function add_category() {
        var currentCount = $('form > fieldset > fieldset').length;
        var template = $('form > fieldset > span').data('template');
        template = template.replace(/__index__/g, currentCount);

        $('form > fieldset').append(template);

        return false;
    }
</script>
```

(Note: the above example assumes `$()` is defined, and equivalent to jQuery's `$()` function, Dojo's `dojo.query`, etc.)

One small remark about the `template.replace`: the example uses `currentCount` and not `currentCount + 1`, as the indices are zero-based (so, if we have two elements in the collection, the third one will have the index `2`).

Now, if we validate the form, it will automatically take into account this new element by validating it, filtering it and retrieving it:

Of course, if you don't want to allow adding elements in a collection, you must set the option `allow_add` to `false`. This way, even if new elements are added, they won't be validated and hence, not added to the entity. Also, if we don't want elements to be added, we don't need the data template, either. Here's how you do it:

```
$this->add(array(
    'type' => 'Zend\Form\Element\Collection',
    'name' => 'categories',
    'options' => array(
        'label' => 'Please choose categories for this product',
        'count' => 2,
        'should_create_template' => false,
        'allow_add' => false,
        'target_element' => array(
            'type' => 'Application\Form\CategoryFieldset'
        )
    )
));
```

There are some limitations to this capability:

- Although you can add new elements and remove them, you *CANNOT* remove more elements in a collection than the initial count (for instance, if your code specifies `count == 2`, you will be able to add a third one and remove it, but you won't be able to remove any others. If the initial count is 2, you *must* have at least two elements.
- Dynamically added elements have to be added at the end of the collection. They can be added anywhere (these elements will still be validated and inserted into the entity), but if the validation fails, this newly added element will be automatically be replaced at the end of the collection.

109.6 Validation groups for fieldsets and collection

Validation groups allow you to validate a subset of fields.

As an example, although the `Brand` entity has a `URL` property, we don't want the user to specify it in the creation form (but may wish to later in the "Edit Product" form, for instance). Let's update the view to remove the `URL` input:

```
<?php
$form->setAttribute('action', $this->url('home'))
     ->prepare();

echo $this->form()->openTag($form);

$product = $form->get('product');

echo $this->formRow($product->get('name'));
echo $this->formRow($product->get('price'));
echo $this->formCollection($product->get('categories'));

$brand = $product->get('brand');

echo $this->formRow($brand->get('name'));

echo $this->formHidden($form->get('csrf'));
echo $this->formElement($form->get('submit'));

echo $this->form()->closeTag();
```

This is what we get:

The `URL` input has disappeared, but even if we fill every input, the form won't validate. In fact, this is normal. We specified in the input filter that the `URL` is a *required* field, so if the form does not have it, it won't validate, even though we didn't add it to the view!

Of course, you could create a `BrandFieldsetWithoutURL` fieldset, but of course this is not recommended, as a lot of code will be duplicated.

The solution: validation groups. A validation group is specified in a `Form` object (hence, in our case, in the `CreateProduct` form) by giving an array of all the elements we want to validate. Our `CreateProduct` now looks like this:

```
namespace Application\Form;

use Zend\Form\Form;
use Zend\InputFilter\InputFilter;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;
```

```
class CreateProduct extends Form
{
    public function __construct()
    {
        parent::__construct('create_product');

        $this->setAttribute('method', 'post')
             ->setHydrator(new ClassMethodsHydrator())
             ->setInputFilter(new InputFilter());

        $this->add(array(
            'type' => 'Application\Form\ProductFieldset',
            'options' => array(
                'use_as_base_fieldset' => true
            )
        ));

        $this->add(array(
            'type' => 'Zend\Form\Element\Csrf',
            'name' => 'csrf'
        ));

        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type' => 'submit',
                'value' => 'Send'
            )
        ));

        $this->setValidationGroup(array(
            'csrf',
            'product' => array(
                'name',
                'price',
                'brand' => array(
                    'name'
                ),
                'categories' => array(
                    'name'
                )
            )
        ));
    }
}
```

Of course, don't forget to add the `CSRF` element, as we want it to be validated too (but notice that I didn't write the submit element, as we don't care about it). You can recursively select the elements you want.

There is one simple limitation currently: validation groups for collections are set on a per-collection basis, not per-element in a collection basis. This means you cannot say, "validate the name input for the first element of the categories collection, but don't validate it for the second one." But, honestly, this is really an edge-case.

Now, the form validates (and the `URL` is set to null as we didn't specify it).

CHAPTER

FILE UPLOADING

Zend Framework provides support for file uploading by using features in `Zend\Form`, `Zend\InputFilter`, `Zend\Validator`, `Zend\Filter`, and `Zend\ProgressBar`. These reusable framework components provide a convenient and secure way for handling file uploads in your projects.

Note: If the reader has experience with file uploading in Zend Framework v1.x, he/she will notice some major differences. `Zend_File_Transfer` has been deprecated in favor of using the standard ZF2 `Zend\Form` and `Zend\InputFilter` features.

Note: The file upload features described here are specifically for forms using the `POST` method. Zend Framework itself does not currently provide specific support for handling uploads via the `PUT` method, but it is possible with PHP. See the PUT Method Support in the PHP documentation for more information.

110.1 Standard Example

Handling file uploads is *essentially* the same as how you would use `Zend\Form` for form processing, but with some slight caveats that will be described below.

In this example we will:

- Define a **Form** for backend validation and filtering.
- Create a **view template** with a `<form>` containing a file input.
- Process the form within a **Controller action**.

110.1.1 The Form and InputFilter

Here we define a `Zend\Form\Element\File` input in a Form class named `UploadForm`.

```
// File: UploadForm.php

use Zend\Form\Element;
use Zend\Form\Form;

class UploadForm extends Form
{
    public function __construct($name = null, $options = array())
    {
        parent::__construct($name, $options);
        $this->addElements();
```

```
    }

    public function addElements()
    {
        // File Input
        $file = new Element\File('image-file');
        $file->setLabel('Avatar Image Upload')
             ->setAttribute('id', 'image-file');
        $this->add($file);
    }
}
```

The `File` element provides some automatic features that happen behind the scenes:

- The form's `enctype` will automatically be set to `multipart/form-data` when the form `prepare()` method is called.
- The file element's default input specification will create the correct `Input` type: *Zend\InputFilter\FileInput*.
- The `FileInput` will automatically prepend an *UploadFile Validator*, to securely validate that the file is actually an uploaded file, and to report other types of upload errors to the user.

110.1.2 The View Template

In the view template we render the `<form>`, a file input (with label and errors), and a submit button.

```
// File: upload-form.phtml
<?php $form->prepare(); // The correct enctype is set here ?>
<?php echo $this->form()->openTag($form); ?>

    <div class="form-element">
        <?php $fileElement = $form->get('image-file'); ?>
        <?php echo $this->formLabel($fileElement); ?>
        <?php echo $this->formFile($fileElement); ?>
        <?php echo $this->formElementErrors($fileElement); ?>
    </div>

    <button>Submit</button>

<?php echo $this->form()->closeTag(); ?>
```

When rendered, the HTML should look similar to:

```
<form name="upload-form" id="upload-form" method="post" enctype="multipart/form-data">
    <div class="form-element">
        <label for="image-file">Avatar Image Upload</label>
        <input type="file" name="image-file" id="image-file">
    </div>

    <button>Submit</button>
</form>
```

110.1.3 The Controller Action

For the final step, we will instantiate the `UploadForm` and process any postbacks in a Controller action.

The form processing in the controller action will be similar to normal forms, *except* that you **must** merge the `$_FILES` information in the request with the other post data.

```
// File: MyController.php

public function uploadFormAction()
{
    $form = new UploadForm('upload-form');

    $request = $this->getRequest();
    if ($request->isPost()) {
        // Make certain to merge the files info!
        $post = array_merge_recursive(
            $request->getPost()->toArray(),
            $request->getFiles()->toArray()
        );

        $form->setData($post);
        if ($form->isValid()) {
            $data = $form->getData();
            // Form is valid, save the form!
            return $this->redirect()->toRoute('upload-form/success');
        }
    }

    return array('form' => $form);
}
```

Upon a successful file upload, `$form->getData()` would return:

```
array(1) {
    ["image-file"] => array(5) {
        ["name"]     => string(11) "myimage.png"
        ["type"]     => string(9)  "image/png"
        ["tmp_name"] => string(22) "/private/tmp/phpgRXd58"
        ["error"]    => int(0)
        ["size"]     => int(14908679)
    }
}
```

Note: It is suggested that you always use the `Zend\Http\PhpEnvironment\Request` object to retrieve and merge the `$_FILES` information with the form, instead of using `$_FILES` directly.

This is due to how the file information is mapped in the `$_FILES` array:

```
// A $_FILES array with single input and multiple files:
array(1) {
    ["image-file"]=>array(2) {
        ["name"]=>array(2) {
            [0]=>string(9)"file0.txt"
            [1]=>string(9)"file1.txt"
        }
        ["type"]=>array(2) {
            [0]=>string(10)"text/plain"
            [1]=>string(10)"text/html"
        }
    }
}

// How Zend\Http\PhpEnvironment\Request remaps the $_FILES array:
array(1) {
```

```
    ["image-file"]=>array(2) {
        [0]=>array(2) {
            ["name"]=>string(9)"file0.txt"
            ["type"]=>string(10)"text/plain"
        },
        [1]=>array(2) {
            ["name"]=>string(9)"file1.txt"
            ["type"]=>string(10)"text/html"
        }
    }
}
```

Zend\InputFilter\FileInput expects the file data be in this re-mapped array format.

110.2 File Post-Redirect-Get Plugin

When using other standard form inputs (i.e. `text`, `checkbox`, `select`, etc.) along with file inputs in a Form, you can encounter a situation where some inputs may become invalid and the user must re-select the file and re-upload. PHP will delete uploaded files from the temporary directory at the end of the request if it has not been moved away or renamed. Re-uploading a valid file each time another form input is invalid is inefficient and annoying to users.

One strategy to get around this is to split the form into multiple forms. One form for the file upload inputs and another for the other standard inputs.

When you cannot separate the forms, the *File Post-Redirect-Get Controller Plugin* can be used to manage the file inputs and save off valid uploads until the entire form is valid.

Changing our earlier example to use the `fileprg` plugin will require two changes.

1. Adding a `RenameUpload` filter to our form's file input, with details on where the valid files should be stored:

```
// File: UploadForm.php

use Zend\InputFilter;
use Zend\Form\Element;
use Zend\Form\Form;

class UploadForm extends Form
{
    public function __construct($name = null, $options = array())
    {
        parent::__construct($name, $options);
        $this->addElements();
        $this->addInputFilter();
    }

    public function addElements()
    {
        // File Input
        $file = new Element\File('image-file');
        $file->setLabel('Avatar Image Upload')
             ->setAttribute('id', 'image-file');
        $this->add($file);
    }

    public function addInputFilter()
```

```
    {
        $inputFilter = new InputFilter\InputFilter();

        // File Input
        $fileInput = new InputFilter\FileInput('image-file');
        $fileInput->setRequired(true);
        $fileInput->getFilterChain()->attachByName(
            'filerenameupload',
            array(
                'target'    => './data/tmpuploads/avatar.png',
                'randomize' => true,
            )
        );
        $inputFilter->add($fileInput);

        $this->setInputFilter($inputFilter);
    }
}
```

The `filerenameupload` options above would cause an uploaded file to be renamed and moved to: `./data/tmpuploads/avatar_4b3403665fea6.png`.

See the *RenameUpload filter* documentation for more information on its supported options.

2. And, changing the Controller action to use the `fileprg` plugin:

```
// File: MyController.php

public function uploadFormAction()
{
    $form     = new UploadForm('upload-form');
    $tempFile = null;

    $prg = $this->fileprg($form);
    if ($prg instanceof \Zend\Http\PhpEnvironment\Response) {
        return $prg; // Return PRG redirect response
    } elseif (is_array($prg)) {
        if ($form->isValid()) {
            $data = $form->getData();
            // Form is valid, save the form!
            return $this->redirect()->toRoute('upload-form/success');
        } else {
            // Form not valid, but file uploads might be valid...
            // Get the temporary file information to show the user in the view
            $fileErrors = $form->get('image-file')->getMessages();
            if (empty($fileErrors)) {
                $tempFile = $form->get('image-file')->getValue();
            }
        }
    }

    return array(
        'form'     => $form,
        'tempFile' => $tempFile,
    );
}
```

Behind the scenes, the `FilePRG` plugin will:

- Run the Form's filters, namely the `RenameUpload` filter, to move the files out of temporary storage.

- Store the valid POST data in the session across requests.
- Change the `required` flag of any file inputs that had valid uploads to `false`. This is so that form re-submissions without uploads will not cause validation errors.

Note: In the case of a partially valid form, it is up to the developer whether to notify the user that files have been uploaded or not. For example, you may wish to hide the form input and/or display the file information. These things would be implementation details in the view or in a custom view helper. Just note that neither the `FilePRG` plugin nor the `formFile` view helper will do any automatic notifications or view changes when files have been successfully uploaded.

110.3 HTML5 Multi-File Uploads

With HTML5 we are able to select multiple files from a single file input using the `multiple` attribute. Not all browsers support multiple file uploads, but the file input will safely remain a single file upload for those browsers that do not support the feature.

To enable multiple file uploads in Zend Framework, just set the file element's `multiple` attribute to true:

```
// File: UploadForm.php

use Zend\InputFilter;
use Zend\Form\Element;
use Zend\Form\Form;

class UploadForm extends Form
{
    public function __construct($name = null, $options = array())
    {
        parent::__construct($name, $options);
        $this->addElements();
        $this->addInputFilter();
    }

    public function addElements()
    {
        // File Input
        $file = new Element\File('image-file');
        $file->setLabel('Avatar Image Upload')
             ->setAttribute('id', 'image-file')
             ->setAttribute('multiple', true);   // That's it
        $this->add($file);
    }

    public function addInputFilter()
    {
        $inputFilter = new InputFilter\InputFilter();

        // File Input
        $fileInput = new InputFilter\FileInput('image-file');
        $fileInput->setRequired(true);

        // You only need to define validators and filters
        // as if only one file was being uploaded. All files
        // will be run through the same validators and filters
```

```
        // automatically.
        $fileInput->getValidatorChain()
            ->attachByName('filesize',      array('max' => 204800))
            ->attachByName('filemimetype',  array('mimeType' => 'image/png,image/x-png'))
            ->attachByName('fileimagesize', array('maxWidth' => 100, 'maxHeight' => 100));

        // All files will be renamed, i.e.:
        //   ./data/tmpuploads/avatar_4b3403665fea6.png,
        //   ./data/tmpuploads/avatar_5c45147660fb7.png
        $fileInput->getFilterChain()->attachByName(
            'filerenameupload',
            array(
                'target'    => './data/tmpuploads/avatar.png',
                'randomize' => true,
            )
        );
        $inputFilter->add($fileInput);

        $this->setInputFilter($inputFilter);
    }
}
```

You do not need to do anything special with the validators and filters to support multiple file uploads. All of the files that are uploaded will have the same validators and filters run against them automatically (from logic within `FileInput`). You only need to define them as if one file was being uploaded.

110.4 Upload Progress

While pure client-based upload progress meters are starting to become available with HTML5's Progress Events, not all browsers have XMLHttpRequest level 2 support. For upload progress to work in a greater number of browsers (IE9 and below), you must use a server-side progress solution.

`Zend\ProgressBar\Upload` provides handlers that can give you the actual state of a file upload in progress. To use this feature you need to choose one of the *Upload Progress Handlers* (APC, uploadprogress, or Session) and ensure that your server setup has the appropriate extension or feature enabled.

Note: For this example we will use PHP **5.4**'s Session progress handler

PHP 5.4 is required and you may need to verify these php.ini settings for it to work:

```
file_uploads = On
post_max_size = 50M
upload_max_filesize = 50M
session.upload_progress.enabled = On
session.upload_progress.freq =  "1%"
session.upload_progress.min_freq = "1"
; Also make certain 'upload_tmp_dir' is writable
```

When uploading a file with a form POST, you must also include the progress identifier in a hidden input. The *File Upload Progress View Helpers* provide a convenient way to add the hidden input based on your handler type.

```
// File: upload-form.phtml
<?php $form->prepare(); ?>
<?php echo $this->form()->openTag($form); ?>
    <?php echo $this->formFileSessionProgress(); // Must come before the file input! ?>
```

```

    <div class="form-element">
        <?php $fileElement = $form->get('image-file'); ?>
        <?php echo $this->formLabel($fileElement); ?>
        <?php echo $this->formFile($fileElement); ?>
        <?php echo $this->formElementErrors($fileElement); ?>
    </div>

    <button>Submit</button>

<?php echo $this->form()->closeTag(); ?>
```

When rendered, the HTML should look similar to:

```
<form name="upload-form" id="upload-form" method="post" enctype="multipart/form-data">
    <input type="hidden" id="progress_key" name="PHP_SESSION_UPLOAD_PROGRESS" value="12345abcde">

    <div class="form-element">
        <label for="image-file">Avatar Image Upload</label>
        <input type="file" name="image-file" id="image-file">
    </div>

    <button>Submit</button>
</form>
```

There are a few different methods for getting progress information to the browser (long vs. short polling). Here we will use short polling since it is simpler and less taxing on server resources, though keep in mind it is not as responsive as long polling.

When our form is submitted via AJAX, the browser will continuously poll the server for upload progress.

The following is an example Controller action which provides the progress information:

```
// File: MyController.php

public function uploadProgressAction()
{
    $id = $this->params()->fromQuery('id', null);
    $progress = new \Zend\ProgressBar\Upload\SessionProgress();
    return new \Zend\View\Model\JsonModel($progress->getProgress($id));
}

// Returns JSON
//{
//    "total"    : 204800,
//    "current"  : 10240,
//    "rate"     : 1024,
//    "message"  : "10kB / 200kB",
//    "done"     : false
//}
```

> **Warning:** This is *not* the most efficient way of providing upload progress, since each polling request must go through the Zend Framework bootstrap process. A better example would be to use a standalone php file in the public folder that bypasses the MVC bootstrapping and only uses the essential `Zend\ProgressBar` adapters.

Back in our view template, we will add the JavaScript to perform the AJAX POST of the form data, and to start a timeout interval for the progress polling. To keep the example code relatively short, we are using the jQuery Form plugin to do the AJAX form POST. If your project uses a different JavaScript framework (or none at all), this will hopefully at least illustrate the necessary high-level logic that would need to be performed.

```
// File: upload-form.phtml
// ...after the form...

<!-- Twitter Bootstrap progress bar styles:
     http://twitter.github.com/bootstrap/components.html#progress -->
<div id="progress" class="help-block">
    <div class="progress progress-info progress-striped">
        <div class="bar"></div>
    </div>
    <p></p>
</div>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js"></script>
<script src="/js/jquery.form.js"></script>
<script>
var progressInterval;

function getProgress() {
    // Poll our controller action with the progress id
    var url = '/upload-form/upload-progress?id=' + $('#progress_key').val();
    $.getJSON(url, function(data) {
        if (data.status && !data.status.done) {
            var value = Math.floor((data.status.current / data.status.total) * 100);
            showProgress(value, 'Uploading...');
        } else {
            showProgress(100, 'Complete!');
            clearInterval(progressInterval);
        }
    });
}

function startProgress() {
    showProgress(0, 'Starting upload...');
    progressInterval = setInterval(getProgress, 900);
}

function showProgress(amount, message) {
    $('#progress').show();
    $('#progress .bar').width(amount + '%');
    $('#progress > p').html(message);
    if (amount < 100) {
        $('#progress .progress')
            .addClass('progress-info active')
            .removeClass('progress-success');
    } else {
        $('#progress .progress')
            .removeClass('progress-info active')
            .addClass('progress-success');
    }
}

$(function() {
    // Register a 'submit' event listener on the form to perform the AJAX POST
    $('#upload-form').on('submit', function(e) {
        e.preventDefault();

        if ($('#image-file').val() == '') {
            // No files selected, abort
```

```
            return;
        }

        // Perform the submit
        //$.fn.ajaxSubmit.debug = true;
        $(this).ajaxSubmit({
            beforeSubmit: function(arr, $form, options) {
                // Notify backend that submit is via ajax
                arr.push({ name: "isAjax", value: "1" });
            },
            success: function (response, statusText, xhr, $form) {
                clearInterval(progressInterval);
                showProgress(100, 'Complete!');

                // TODO: You'll need to do some custom logic here to handle a successful
                // form post, and when the form is invalid with validation errors.
                if (response.status) {
                    // TODO: Do something with a successful form post, like redirect
                    // window.location.replace(response.redirect);
                } else {
                    // Clear the file input, otherwise the same file gets re-uploaded
                    // http://stackoverflow.com/a/1043969
                    var fileInput = $('#image-file');
                    fileInput.replaceWith( fileInput.val('').clone( true ) );

                    // TODO: Do something with these errors
                    // showErrors(response.formErrors);
                }
            },
            error: function(a, b, c) {
                // NOTE: This callback is *not* called when the form is invalid.
                // It is called when the browser is unable to initiate or complete the ajax submit.
                // You will need to handle validation errors in the 'success' callback.
                console.log(a, b, c);
            }
        });
        // Start the progress polling
        startProgress();
    });
});
</script>
```

And finally, our Controller action can be modified to return form status and validation messages in JSON format if we see the 'isAjax' post parameter (which was set in the JavaScript just before submit):

```
// File: MyController.php

public function uploadFormAction()
{
    $form = new UploadForm('upload-form');

    $request = $this->getRequest();
    if ($request->isPost()) {
        // Make certain to merge the files info!
        $post = array_merge_recursive(
            $request->getPost()->toArray(),
            $request->getFiles()->toArray()
        );
```

```

        $form->setData($post);
        if ($form->isValid()) {
            $data = $form->getData();
            // Form is valid, save the form!
            if (!empty($post['isAjax'])) {
                return new JsonModel(array(
                    'status'   => true,
                    'redirect' => $this->url()->fromRoute('upload-form/success'),
                    'formData' => $data,
                ));
            } else {
                // Fallback for non-JS clients
                return $this->redirect()->toRoute('upload-form/success');
            }
        } else {
            if (!empty($post['isAjax'])) {
                 // Send back failure information via JSON
                 return new JsonModel(array(
                     'status'     => false,
                     'formErrors' => $form->getMessages(),
                     'formData'   => $form->getData(),
                 ));
            }
        }
    }

    return array('form' => $form);
}
```

110.5 Additional Info

Related documentation:

- *Form File Element*
- *Form File View Helper*
- *List of File Validators*
- *List of File Filters*
- *File Post-Redirect-Get Controller Plugin*
- *Zend\InputFilter\FileInput*
- *Upload Progress Handlers*
- *Upload Progress View Helpers*

External resources and blog posts from the community:

- ZF2FileUploadExamples : A ZF2 module with several file upload examples.

CHAPTER

ONE

ADVANCED USE OF FORMS

Beginning with Zend Framework 2.1, forms elements can be registered using a designated plugin manager of *Zend\ServiceManager*. This is similar to how view helpers, controller plugins, and filters are registered. This new feature has a number of benefits, especially when you need to handle complex dependencies in forms/fieldsets. This section describes all the benefits of this new architecture in ZF 2.1.

111.1 Short names

The first advantage of pulling form elements from the service manager is that now you can use short names to create new elements through the factory. Therefore, this code:

```
$form->add(array(
    'type' => 'Zend\Form\Element\Email',
    'name' => 'email'
));
```

can now be replaced by:

```
$form->add(array(
     'type' => 'Email',
     'name' => 'email'
));
```

Each element provided out-of-the-box by Zend Framework 2 support this natively, so you can now make your initialization code more compact.

111.2 Creating custom elements

`Zend\Form` also supports custom form elements.

To create a custom form element, make it extend the `Zend\Form\Element` class, or if you need a more specific one, extend one of the `Zend\Form\Element` classes.

In the following we will show how to create a custom `Phone` element for entering phone numbers. It will extend `Zend\Form\Element` class and provide some default input rules.

Our custom phone element could look something like this:

```
namespace Application\Form\Element;

use Zend\Form\Element;
```

```

use Zend\Form\Element;
use Zend\InputFilter\InputProviderInterface;
use Zend\Validator\Regex as RegexValidator;

class Phone extends Element implements InputProviderInterface
{
    /**
     * @var ValidatorInterface
     */
    protected $validator;

    /**
    * Get a validator if none has been set.
    *
    * @return ValidatorInterface
    */
    public function getValidator()
    {
        if (null === $this->validator) {
            $validator = new RegexValidator('/^\+?\d{11,12}$/');
            $validator->setMessage('Please enter 11 or 12 digits only!',
                                   RegexValidator::NOT_MATCH);

            $this->validator = $validator;
        }

        return $this->validator;
    }

    /**
     * Sets the validator to use for this element
     *
     * @param  ValidatorInterface $validator
     * @return Application\Form\Element\Phone
     */
    public function setValidator(ValidatorInterface $validator)
    {
        $this->validator = $validator;
        return $this;
    }

    /**
     * Provide default input rules for this element
     *
     * Attaches a phone number validator.
     *
     * @return array
     */
    public function getInputSpecification()
    {
        return array(
            'name' => $this->getName(),
            'required' => true,
            'filters' => array(
                array('name' => 'Zend\Filter\StringTrim'),
            ),
            'validators' => array(
```

```
                $this->getValidator(),
            ),
        );
    }
}
```

By implementing the `Zend\InputFilter\InputProviderInterface` interface, we are hinting to our form object that this element provides some default input rules for filtering and/or validating values. In this example the default input specification provides a `Zend\Filter\StringTrim` filter and a `Zend\Validator\Regex` validator that validates that the value optionally has a + sign at the beginning and is followed by 11 or 12 digits.

The easiest way of start using your new custom element in your forms is to use the custom element's FCQN:

```
$form = new Zend\Form\Form();
$form->add(array(
    'name' => 'phone',
    'type' => 'Application\Form\Element\Phone',
));
```

Or, if you are extending `Zend\Form\Form`:

```
namespace Application\Form;

use Zend\Form\Form;

class MyForm extends Form
{
    public function __construct($name = null)
    {
        parent::__construct($name);

        $this->add(array(
            'name' => 'phone',
            'type' => 'Application\Form\Element\Phone',
        ))
    }
}
```

If you don't want to use the custom element's FCQN, but rather a short name, as of Zend Framework 2.1 you can do so by adding them to the `Zend\Form\FormElementManager` plugin manager by utilising the `getFormElementConfig` function.

> **Warning:** To use custom elements with the FormElementManager needs a bit more work and most likely a change in how you write and use your forms.

First, add the custom element to the plugin manager, in your `Module.php` class:

```
namespace Application;

use Zend\ModuleManager\Feature\FormElementProviderInterface;

class Module implements FormElementProviderInterface
{
    public function getFormElementConfig()
    {
        return array(
            'invokables' => array(
                'phone' => 'Application\Form\Element\Phone'
```

```
            )
        );
    }
}
```

Or, you can do the same in your `module.config.php` file:

```
return array(
    'form_elements' => array(
        'invokables' => array(
            'phone' => 'Application\Form\Element\Phone'
        )
    )
);
```

You can use a factory instead of an invokable in order to handle dependencies in your elements/fieldsets/forms.

And now comes the first catch.

If you are creating your form class by extending `Zend\Form\Form`, you *must not* add the custom element in the `__construct`-or (as we have done in the previous example where we used the custom element's FCQN), but rather in the `init()` method:

```
namespace Application\Form;

use Zend\Form\Form;

class MyForm extends Form
{
    public function init()
    {
        $this->add(array(
            'name' => 'phone',
            'type' => 'Phone',
        ))
    }
}
```

The second catch is that you *must not* directly instantiate your form class, but rather get an instance of it through the `Zend\Form\FormElementManager`:

```
namespace Application\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        $sl = $this->getServiceLocator();
        $form = $sl->get('FormElementManager')->get('\Application\Form\MyForm');
        return array('form' => $form);
    }
}
```

The biggest gain of this is that you can easily override any built-in Zend Framework form elements if they do not fit your needs. For instance, if you want to create your own Email element instead of the standard one, you can simply create your element and add it to the form element config with the same key as the element you want to replace:

```
namespace Application;

use Zend\ModuleManager\Feature\FormElementProviderInterface;

class Module implements FormElementProviderInterface
{
    public function getFormElementConfig()
    {
        return array(
            'invokables' => array(
                'Email' => 'Application\Form\Element\MyEmail'
            )
        );
    }
}
```

Now, whenever you'll create an element whose `type` is 'Email', it will create the custom Email element instead of the built-in one.

Note: if you want to be able to use both the built-in one and your own one, you can still provide the FQCN of the element, i.e. `Zend\Form\Element\Email`.

As you can see here, we first get the form manager (that we modified in our Module.php class), and create the form by specifying the fully qualified class name of the form. Please note that you don't need to add *ApplicationFormMyForm* to the *invokables* array. If it is not specified, the form manager will just instantiate it directly.

In short, to create your own form elements (or even reusable fieldsets !) and be able to use them in your form using the short-name notation, you need to:

1. Create your element (like you did before).
2. Add it to the form element manager by defining the `getFormElementConfig`, exactly like using `getServiceConfig()` and `getControllerConfig`.
3. Make sure the custom form element is not added in the form's `__construct`-or, but rather in it's `init()` method, or after getting an instance of the form.
4. Create your form through the form element manager instead of directly instantiating it.

111.3 Handling dependencies

One of the most complex issues in `Zend\\Form 2.0` was dependency management. For instance, a very frequent use case is a form that creates a fieldset, that itself need access to the database to populate a `Select` element. Previously in such a situation, you would either rely on the Registry using the Singleton pattern, or either you would "transfer" the dependency from controller to form, and from form to fieldset (and even from fieldset to another fieldset if you have a complex form). This was ugly and not easy to use. Hopefully, `Zend\\ServiceManager` solves this use case in an elegant manner.

For instance, let's say that a form create a fieldset called `AlbumFieldset`:

```
namespace Application\Form;

use Zend\Form\Form;

class CreateAlbum extends Form
{
```

```
    public function __construct()
    {
        $this->add(array(
            'name' => 'album',
            'type' => 'AlbumFieldset'
        ));
    }
}
```

Let's now create the `AlbumFieldset` that depends on an `AlbumTable` object that allows you to fetch albums from the database.

```
namespace Application\Form;

use Album\Model;
use Zend\Form\Fieldset;

class AlbumFieldset extends Fieldset
{
    public function __construct(AlbumTable $albumTable)
    {
        // Add any elements that need to fetch data from database
        // using the album table !
    }
}
```

For this to work, you need to add a line to the form element manager by adding an element to your Module.php class:

```
namespace Application;

use Application\Form\AlbumFieldset;
use Zend\ModuleManager\Feature\FormElementProviderInterface;

class Module implements FormElementProviderInterface
{
    public function getFormElementConfig()
    {
        return array(
            'factories' => array(
                'AlbumFieldset' => function($sm) {
                    // I assume here that the Album\Model\AlbumTable
                    // dependency have been defined too

                    $serviceLocator = $sm->getServiceLocator();
                    $albumTable = $serviceLocator->get('Album\Model\AlbumTable');
                    $fieldset = new AlbumFieldset($albumTable);
                }
            )
        );
    }
}
```

Finally, create your form using the form element manager instead of directly instantiating it:

```
public function testAction()
{
    $formManager = $this->serviceLocator->get('FormElementManager');
    $form = $formManager->get('Application\Form\CreateAlbum');
}
```

Et voilà! The dependency will be automatically handled by the form element manager, and you don't need to create the `AlbumTable` in your controller, transfer it to the form, which itself passes it over to the fieldset.

111.4 The specific case of initializers

In the previous example, we explicitly defined the dependency in the constructor of the `AlbumFieldset` class. However, in some cases, you may want to use an initializer (like `Zend\\ServiceManager\\ServiceLocatorAwareInterface`) to inject a specific object to all your forms/fieldsets/elements.

The problem with initializers is that they are injected AFTER the construction of the object, which means that if you need this dependency when you create elements, it won't be available yet. For instance, this example **won't work**:

```
namespace Application\Form;

use Album\Model;
use Zend\Form\Fieldset;
use Zend\ServiceManager\ServiceLocatorAwareInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class AlbumFieldset extends Fieldset implements ServiceLocatorAwareInterface
{
    protected $serviceLocator;

    public function __construct()
    {
        // Here, $this->serviceLocator is null because it has not been
        // injected yet, as initializers are run after __construct
    }

    public function setServiceLocator(ServiceLocatorInterface $sl)
    {
        $this->serviceLocator = $sl;
    }

    public function getServiceLocator()
    {
        return $this->serviceLocator;
    }
}
```

Thankfully, there is an easy workaround: every form element now implements the new interface `Zend\\Stdlib\\InitializableInterface`, that defines a single `init()` function. In the context of form elements, this `init()` function is automatically called once all the dependencies (including all initializers) are resolved. Therefore, the previous example can be rewritten as such:

```
namespace Application\Form;

use Album\Model;
use Zend\Form\Fieldset;
use Zend\ServiceManager\ServiceLocatorAwareInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class AlbumFieldset extends Fieldset implements ServiceLocatorAwareInterface
{
    protected $serviceLocator;

```

```
12      public function init()
13      {
14          // Here, we have $this->serviceLocator !!
15      }
16
17      public function setServiceLocator(ServiceLocatorInterface $sl)
18      {
19          $this->serviceLocator = $sl;
20      }
21
22      public function getServiceLocator()
23      {
24          return $this->serviceLocator;
25      }
26  }
```

CHAPTER

TWO

FORM ELEMENTS

112.1 Introduction

A set of specialized elements are provided for accomplishing application-centric tasks. These include several HTML5 input elements with matching server-side validators, the `Csrf` element (to prevent Cross Site Request Forgery attacks), and the `Captcha` element (to display and validate *CAPTCHAs*).

A `Factory` is provided to facilitate creation of elements, fieldsets, forms, and the related input filter. See the *Zend\Form Quick Start* for more information.

112.2 Element Base Class

`Zend\Form\Element` is a base class for all specialized elements and `Zend\Form\Fieldset`.

Basic Usage

At the bare minimum, each element or fieldset requires a name. You will also typically provide some attributes to hint to the view layer how it might render the item.

```
use Zend\Form\Element;
use Zend\Form\Form;

$username = new Element\Text('username');
$username
    ->setLabel('Username')
    ->setAttributes(array(
        'class' => 'username',
        'size'  => '30',
    ));

$password = new Element\Password('password');
$password
    ->setLabel('Password')
    ->setAttributes(array(
        'size'  => '30',
    ));

$form = new Form('my-form');
$form
```

```
21      ->add($username)
22      ->add($password);
```

Public Methods

setName (*string $name*)
Set the name for this element.

getName ()
Return the name for this element.

Return type string

setValue (*string $value*)
Set the value for this element.

getValue ()
Return the value for this element.

Return type string

setLabel (*string $label*)
Set the label content for this element.

getLabel ()
Return the label content for this element.

Return type string

setLabelAttributes (*array $labelAttributes*)
Set the attributes to use with the label.

getLabelAttributes ()
Return the attributes to use with the label.

Return type array

setOptions (*array $options*)
Set options for an element. Accepted options are: `"label"` and `"label_attributes"`, which call `setLabel` and `setLabelAttributes`, respectively.

getOptions ()
Get defined options for an element

Return type array

getOption (*string $option*)
Return the specified option, if defined. If it's not defined, returns null.

Return type null|mixed

setAttribute (*string $key*, *mixed $value*)
Set a single element attribute.

getAttribute (*string $key*)
Retrieve a single element attribute.

Return type mixed

removeAttribute (*string $key*)
Remove a single attribute

hasAttribute(*string $key*)
Check if a specific attribute exists for this element.

Return type boolean

setAttributes(*array|Traversable $arrayOrTraversable*)
Set many attributes at once. Implementation will decide if this will overwrite or merge.

getAttributes()
Retrieve all attributes at once.

Return type array|Traversable

removeAttributes(*array $keys*)
Remove many attributes at once

clearAttributes()
Clear all attributes for this element.

setMessages(*array|Traversable $messages*)
Set a list of messages to report when validation fails.

getMessages()
Returns a list of validation failure messages, if any.

Return type array|Traversable

112.3 Standard Elements

112.3.1 Button

`Zend\Form\Element\Button` represents a button form input. It can be used with the `Zend\Form\View\Helper\FormButton` view helper.

`Zend\Form\Element\Button` extends from *ZendFormElement*.

Basic Usage

This element automatically adds a `type` attribute of value `button`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$button = new Element\Button('my-button');
$button->setLabel('My Button')
       ->setValue('foo');

$form = new Form('my-form');
$form->add($button);
```

112.3.2 Captcha

`Zend\Form\Element\Captcha` can be used with forms where authenticated users are not necessary, but you want to prevent spam submissions. It is paired with one of the `Zend\Form\View\Helper\Captcha\*` view helpers that matches the type of *CAPTCHA* adapter in use.

Basic Usage

A *CAPTCHA* adapter must be attached in order for validation to be included in the element's input filter specification. See the section on *Zend CAPTCHA Adapters* for more information on what adapters are available.

```
use Zend\Captcha;
use Zend\Form\Element;
use Zend\Form\Form;

$captcha = new Element\Captcha('captcha');
$captcha
    ->setCaptcha(new Captcha\Dumb())
    ->setLabel('Please verify you are human');

$form = new Form('my-form');
$form->add($captcha);
```

Here is with the array notation:

```
 use Zend\Captcha;
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Captcha',
     'name' => 'captcha',
     'options' => array(
         'label' => 'Please verify you are human',
         'captcha' => new Captcha\Dumb(),
     ),
 ));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

setCaptcha(*array|Zend\Captcha\AdapterInterface $captcha*)

Set the *CAPTCHA* adapter for this element. If `$captcha` is an array, `Zend\Captcha\Factory::factory()` will be run to create the adapter from the array configuration.

getCaptcha()

Return the *CAPTCHA* adapter for this element.

Return type `Zend\Captcha\AdapterInterface`

getInputSpecification()

Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter, and a *CAPTCHA* validator.

Return type array

112.3.3 Checkbox

`Zend\Form\Element\Checkbox` is meant to be paired with the `Zend\Form\View\Helper\FormCheckbox` for HTML inputs with type checkbox. This element adds an `InArray` validator to its input filter specification in order to validate on the server if the checkbox contains either the checked value or the unchecked value.

Basic Usage

This element automatically adds a "type" attribute of value "checkbox".

```
use Zend\Form\Element;
use Zend\Form\Form;

$checkbox = new Element\Checkbox('checkbox');
$checkbox->setLabel('A checkbox');
$checkbox->setUseHiddenElement(true);
$checkbox->setCheckedValue("good");
$checkbox->setUncheckedValue("bad");

$form = new Form('my-form');
$form->add($checkbox);
```

Using the array notation:

```
use Zend\Form\Form;

$form = new Form('my-form');
$form->add(array(
    'type' => 'Zend\Form\Element\Checkbox',
    'name' => 'checkbox',
    'options' => array(
        'label' => 'A checkbox',
        'use_hidden_element' => true,
        'checked_value' => 'good',
        'unchecked_value' => 'bad'
    )
));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element* .

setOptions (*array $options*)
Set options for an element of type Checkbox. Accepted options, in addition to the inherited *options of Zend\Form\Element* , are: "use_hidden_element", "checked_value" and "unchecked_value" , which call setUseHiddenElement, setCheckedValue and setUncheckedValue , respectively.

setUseHiddenElement (*boolean $useHiddenElement*)
If set to true (which is default), the view helper will generate a hidden element that contains the unchecked value. Therefore, when using custom unchecked value, this option have to be set to true.

useHiddenElement ()
Return if a hidden element is generated.

Return type boolean

setCheckedValue (*string $checkedValue*)
Set the value to use when the checkbox is checked.

getCheckedValue ()
Return the value used when the checkbox is checked.

Return type string

setUncheckedValue(*string $uncheckedValue*)
Set the value to use when the checkbox is unchecked. For this to work, you must make sure that use_hidden_element is set to true.

getUncheckedValue()
Return the value used when the checkbox is unchecked.

Return type string

getInputSpecification()
Returns a input filter specification, which includes a `Zend\Validator\InArray` to validate if the value is either checked value or unchecked value.

Return type array

isChecked()
Checks if the checkbox is checked.

Return type boolean

setChecked(*bool $value*)
Checks or unchecks the checkbox.

112.3.4 Collection

Sometimes, you may want to add input (or a set of inputs) multiple times, either because you don't want to duplicate code, or because you do not know in advance how many elements you will need (in the case of elements dynamically added to a form using JavaScript, for instance). For more information about Collection, please refer to the *Form Collections tutorial*.

`Zend\Form\Element\Collection` is meant to be paired with the `Zend\Form\View\Helper\FormCollection`.

Basic Usage

```
use Zend\Form\Element;
use Zend\Form\Form;

$colors = new Element\Collection('collection');
$colors->setLabel('Colors');
$colors->setCount(2);
$colors->setTargetElement(new Element\Color());
$colors->setShouldCreateTemplate(true);

$form = new Form('my-form');
$form->add($colors);
```

Using the array notation:

```
use Zend\Form\Element;
use Zend\Form\Form;

$form = new Form('my-form');
$form->add(array(
    'type' => 'Zend\Form\Element\Collection',
    'options' => array(
        'label' => 'Colors',
        'count' => 2,
```

```
        'should_create_template' => true,
        'target_element' => new Element\Color()
    )
));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element* .

setOptions (*array $options*)
Set options for an element of type Collection. Accepted options, in addition to the inherited options of *Zend\Form\Element* , are: `"target_element"`, `"count"`, `"allow_add"`, `"allow_remove"`, `"should_create_template"` and `"template_placeholder"`. Those option keys respectively call call `setTargetElement`, `setCount`, `setAllowAdd`, `setAllowRemove`, `setShouldCreateTemplate` and `setTemplatePlaceholder`.

allowObjectBinding (*object $object*)
Checks if the object can be set in this fieldset.

Return type bool

setObject (*array|Traversable $object*)
Set the object used by the hydrator. In this case the "object" is a collection of objects.

populateValues (*array|Traversable $data*)
Populate values

allowValueBinding ()
Checks if this fieldset can bind data

Return type bool

setCount (*$count*)
Defines how many times the target element will be initially rendered by the `Zend\Form\View\Helper\FormCollection` view helper.

getCount ()
Return the number of times the target element will be initially rendered by the `Zend\Form\View\Helper\FormCollection` view helper.

Return type integer

setTargetElement (*$elementOrFieldset*)
This function either takes an `Zend\Form\ElementInterface`, `Zend\Form\FieldsetInterface` instance or an array to pass to the form factory. When the Collection element will be validated, the input filter will be retrieved from this target element and be used to validate each element in the collection.

getTargetElement ()
Return the target element used by the collection.

Return type ElementInterface | null

setAllowAdd (*$allowAdd*)
If allowAdd is set to true (which is the default), new elements added dynamically in the form (using JavaScript, for instance) will also be validated and retrieved.

allowAdd ()
Return if new elements can be dynamically added in the collection.

Return type boolean

setAllowRemove(*$allowRemove*)
If allowRemove is set to true (which is the default), new elements added dynamically in the form (using JavaScript, for instance) will be allowed to be removed.

allowRemove()
Return if new elements can be dynamically removed from the collection.

Return type boolean

setShouldCreateTemplate(*$shouldCreateTemplate*)
If shouldCreateTemplate is set to true (defaults to false), a element will be generated by the `Zend\Form\View\Helper\FormCollection` view helper. This non-semantic span element contains a single data-template HTML5 attribute whose value is the whole HTML to copy to create a new element in the form. The template is indexed using the `templatePlaceholder` value.

shouldCreateTemplate()
Return if a template should be created.

Return type boolean

setTemplatePlaceholder(*$templatePlaceholder*)
Set the template placeholder (defaults to __index__) used to index element in the template.

getTemplatePlaceholder()
Returns the template placeholder used to index element in the template.

Return type string

getTemplateElement()
Get a template element used for rendering purposes only

Return type null|ElementInterface|FieldsetInterface

prepareElement()
Prepare the collection by adding a dummy template element if the user want one

prepareFieldset()
If both count and targetElement are set, add them to the fieldset

112.3.5 Csrf

`Zend\Form\Element\Csrf` pairs with the `Zend\Form\View\Helper\FormHidden` to provide protection from *CSRF* attacks on forms, ensuring the data is submitted by the user session that generated the form and not by a rogue script. Protection is achieved by adding a hash element to a form and verifying it when the form is submitted.

Basic Usage

This element automatically adds a `"type"` attribute of value `"hidden"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$csrf = new Element\Csrf('csrf');

$form = new Form('my-form');
$form->add($csrf);
```

You can change the options of the CSRF validator using the `setCsrfValidatorOptions` function, or by using the `"csrf_options"` key. Here is an example using the array notation:

```
use Zend\Form\Form;

$form = new Form('my-form');
$form->add(array(
    'type' => 'Zend\Form\Element\Csrf',
    'name' => 'csrf',
    'options' => array(
            'csrf_options' => array(
                    'timeout' => 600
            )
    )
));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()
Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter and a `Zend\Validator\Csrf` to validate the *CSRF* value.

Return type array

setCsrfValidatorOptions(*array $options*)
Set the options that are used by the CSRF validator.

getCsrfValidatorOptions()
Get the options that are used by the CSRF validator.

Return type array

setCsrfValidator(*ZendValidatorCsrf $validator*)
Override the default CSRF validator by setting another one.

getCsrfValidator()
Get the CSRF validator.

Return type ZendValidatorCsrf

112.3.6 File

`Zend\Form\Element\File` represents a form file input and provides a default input specification with a type of *FileInput* (important for handling validators and filters correctly). It can be used with the `Zend\Form\View\Helper\FormFile` view helper.

`Zend\Form\Element\File` extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a `"type"` attribute of value `"file"`. It will also set the form's enctype to `multipart/form-data` during `$form->prepare()`.

```
use Zend\Form\Element;
use Zend\Form\Form;

// Single file upload
$file = new Element\File('file');
```

```
$file->setLabel('Single file input');

// HTML5 multiple file upload
$multiFile = new Element\File('multi-file');
$multiFile->setLabel('Multi file input')
          ->setAttribute('multiple', true);

$form = new Form('my-file');
$form->add($file)
     ->add($multiFile);
```

112.3.7 Hidden

`Zend\Form\Element\Hidden` represents a hidden form input. It can be used with the `Zend\Form\View\Helper\FormHidden` view helper.

`Zend\Form\Element\Hidden` extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a `"type"` attribute of value `"hidden"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$hidden = new Element\Hidden('my-hidden');
$hidden->setValue('foo');

$form = new Form('my-form');
$form->add($hidden);
```

Here is with the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Hidden',
     'name' => 'my-hidden',
     'attributes' => array(
             'value' => 'foo'
     )
 ));
```

112.3.8 Image

`Zend\Form\Element\Image` represents a image button form input. It can be used with the `Zend\Form\View\Helper\FormImage` view helper.

`Zend\Form\Element\Image` extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a `"type"` attribute of value `"image"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$image = new Element\Image('my-image');
$image->setAttribute('src', 'http://my.image.url'); // Src attribute is required

$form = new Form('my-form');
$form->add($image);
```

112.3.9 Month Select

`Zend\Form\Element\MonthSelect` is meant to be paired with the `Zend\Form\View\Helper\FormMonthSelect`. This element creates two select elements, where the first one is populated with months and the second is populated with years. By default, it sets 100 years in the past for the year element, starting with the current year.

Basic Usage

```
use Zend\Form\Element;
use Zend\Form\Form;

$monthYear = new Element\MonthSelect('monthyear');
$monthYear->setLabel('Select a month and a year');
$monthYear->setMinYear(1986);

$form = new Form('dateselect');
$form->add($monthYear);
```

Using the array notation:

```
use Zend\Form\Form;

$form = new Form('dateselect');
$form->add(array(
    'type' => 'Zend\Form\Element\MonthSelect',
    'name' => 'monthyear',
    'options' => array(
        'label' => 'Select a month and a year',
        'min_year' => 1986,
    )
));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getMonthElement()
Returns the Select element that is used for the months part.

Return type `Zend\Form\Element\Select`

getYearElement()
Returns the Select element that is used for the years part.

Return type `Zend\Form\Element\Select`

setMonthAttributes (*array $monthAttributes*)
Set attributes on the Select element that is used for the months part.

getMonthAttributes ()
Get attributes on the Select element that is used for the months part.

Return type array

setYearAttributes (*array $yearAttributes*)
Set attributes on the Select element that is used for the years part.

getYearAttributes ()
Get attributes on the Select element that is used for the years part.

Return type array

setMinYear (*int $minYear*)
Set the minimum year.

getMinYear ()
Get the minimum year.

setMaxYear (*int $maxYear*)
Set the maximum year.

getMaxYear ()
Get the maximum year.

setValue (*mixed $value*)
Set the value for the MonthSelect element.

If the value is an instance of `\DateTime`, it will use the month and year values from that date. Otherwise, the value should be an associative array with the `month` key for the month value, and with the `year` key for the year value.

112.3.10 MultiCheckbox

`Zend\Form\Element\MultiCheckbox` is meant to be paired with the `Zend\Form\View\Helper\FormMultiCheckbox` for HTML inputs with type checkbox. This element adds an `InArray` validator to its input filter specification in order to validate on the server if the checkbox contains values from the multiple checkboxes.

Basic Usage

This element automatically adds a `"type"` attribute of value `"checkbox"` for every checkboxes.

```
use Zend\Form\Element;
use Zend\Form\Form;

$multiCheckbox = new Element\MultiCheckbox('multi-checkbox');
$multiCheckbox->setLabel('What do you like ?');
$multiCheckbox->setValueOptions(array(
                '0' => 'Apple',
                '1' => 'Orange',
                '2' => 'Lemon'
));

$form = new Form('my-form');
$form->add($multiCheckbox);
```

Using the array notation:

```
use Zend\Form\Form;

    $form = new Form('my-form');
    $form->add(array(
            'type' => 'Zend\Form\Element\MultiCheckbox',
            'name' => 'multi-checkbox',
            'options' => array(
                    'label' => 'What do you like ?',
                    'value_options' => array(
                            '0' => 'Apple',
                            '1' => 'Orange',
                            '2' => 'Lemon',
                    ),
            )
    ));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\Checkbox* .

setOptions(*array $options*)
Set options for an element of type Checkbox. Accepted options, in addition to the inherited *options of Zend\Form\Element\Checkbox* , are: `"value_options"`, which call `setValueOptions`.

setValueOptions(*array $options*)
Set the value options for every checkbox of the multi-checkbox. The array must contain a key => value for every checkbox.

getValueOptions()
Return the value options.

Return type array

112.3.11 Password

`Zend\Form\Element\Password` represents a password form input. It can be used with the `Zend\Form\View\Helper\FormPassword` view helper.

`Zend\Form\Element\Password` extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a `"type"` attribute of value `"password"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$password = new Element\Password('my-password');
$password->setLabel('Enter your password');

$form = new Form('my-form');
$form->add($password);
```

112.3.12 Radio

`Zend\Form\Element\Radio` is meant to be paired with the `Zend\Form\View\Helper\FormRadio` for HTML inputs with type radio. This element adds an `InArray` validator to its input filter specification in order to validate on the server if the value is contains within the radio value elements.

Basic Usage

This element automatically adds a `"type"` attribute of value `"radio"` for every radio.

```
use Zend\Form\Element;
use Zend\Form\Form;

$radio = new Element\Radio('gender');
$radio->setLabel('What is your gender ?');
$radio->setValueOptions(array(
                '0' => 'Female',
                '1' => 'Male',
));

$form = new Form('my-form');
$form->add($radio);
```

Using the array notation:

```
use Zend\Form\Form;

$form = new Form('my-form');
$form->add(array(
        'type' => 'Zend\Form\Element\Radio',
        'name' => 'gender',
        'options' => array(
                'label' => 'What is your gender ?',
                'value_options' => array(
                        '0' => 'Female',
                        '1' => 'Male',
                ),
        )
));
```

Public Methods

All the methods from the inherited *methods of Zend\Form\Element\MultiCheckbox* are also available for this element.

112.3.13 Select

`Zend\Form\Element\Select` is meant to be paired with the `Zend\Form\View\Helper\FormSelect` for HTML inputs with type select. This element adds an `InArray` validator to its input filter specification in order to validate on the server if the selected value belongs to the values. This element can be used as a multi-select element by adding the "multiple" HTML attribute to the element.

Basic Usage

This element automatically adds a "type" attribute of value "select".

```
use Zend\Form\Element;
use Zend\Form\Form;

$select = new Element\Select('language');
$select->setLabel('Which is your mother tongue?');
$select->setValueOptions(array(
        '0' => 'French',
        '1' => 'English',
        '2' => 'Japanese',
        '3' => 'Chinese',
));

$form = new Form('language');
$form->add($select);
```

Using the array notation:

```
use Zend\Form\Form;

$form = new Form('my-form');
$form->add(array(
        'type' => 'Zend\Form\Element\Select',
        'name' => 'language',
        'options' => array(
                'label' => 'Which is your mother tongue?',
                'value_options' => array(
                        '0' => 'French',
                        '1' => 'English',
                        '2' => 'Japanese',
                        '3' => 'Chinese',
                ),
        )
));
```

You can add an empty option (option with no value) using the "empty_option" option:

```
use Zend\Form\Form;

$form = new Form('my-form');
$form->add(array(
        'type' => 'Zend\Form\Element\Select',
        'name' => 'language',
        'options' => array(
                'label' => 'Which is your mother tongue?',
                'empty_option' => 'Please choose your language',
                'value_options' => array(
                        '0' => 'French',
                        '1' => 'English',
                        '2' => 'Japanese',
                        '3' => 'Chinese',
                ),
        )
));
```

Option groups are also supported. You just need to add an 'options' key to the value options.

```
use Zend\Form\Element;
use Zend\Form\Form;

$select = new Element\Select('language');
$select->setLabel('Which is your mother tongue?');
$select->setValueOptions(array(
 'european' => array(
    'label' => 'European languages',
    'options' => array(
       '0' => 'French',
       '1' => 'Italian',
    ),
 ),
 'asian' => array(
    'label' => 'Asian languages',
    'options' => array(
       '2' => 'Japanese',
       '3' => 'Chinese',
    ),
 ),
));

$form = new Form('language');
$form->add($select);
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element* .

setOptions(*array $options*)
Set options for an element of type Checkbox. Accepted options, in addition to the inherited *options of Zend\Form\Element\Checkbox* , are: `"value_options"` and `"empty_option"`, which call `setValueOptions` and `setEmptyOption`, respectively.

setValueOptions(*array $options*)
Set the value options for the select element. The array must contain key => value pairs.

getValueOptions()
Return the value options.

Return type array

setEmptyOption(*$emptyOption*)
Optionally set a label for an empty option (option with no value). It is set to "null" by default, which means that no empty option will be rendered.

getEmptyOption()
Get the label for the empty option (null if none).

Return type string|null

112.3.14 Submit

`Zend\Form\Element\Submit` represents a submit button form input. It can be used with the `Zend\Form\View\Helper\FormSubmit` view helper.

`Zend\Form\Element\Submit` extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a "type" attribute of value "submit".

```
use Zend\Form\Element;
use Zend\Form\Form;

$submit = new Element\Submit('my-submit');
$submit->setValue('Submit Form');

$form = new Form('my-form');
$form->add($submit);
```

112.3.15 Text

Zend\Form\Element\Text represents a text form input. It can be used with the Zend\Form\View\Helper\FormText view helper.

Zend\Form\Element\Text extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a "type" attribute of value "text".

```
use Zend\Form\Element;
use Zend\Form\Form;

$text = new Element\Text('my-text');
$text->setLabel('Enter your name');

$form = new Form('my-form');
$form->add($text);
```

112.3.16 Textarea

Zend\Form\Element\Textarea represents a textarea form input. It can be used with the Zend\Form\View\Helper\FormTextarea view helper.

Zend\Form\Element\Textarea extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a "type" attribute of value "textarea".

```
use Zend\Form\Element;
use Zend\Form\Form;

$textarea = new Element\Textarea('my-textarea');
$textarea->setLabel('Enter a description');

$form = new Form('my-form');
$form->add($textarea);
```

112.4 HTML5 Elements

112.4.1 Color

`Zend\Form\Element\Color` is meant to be paired with the `Zend\Form\View\Helper\FormColor` for HTML5 inputs with type color. This element adds filters and a `Regex` validator to it's input filter specification in order to validate a HTML5 valid simple color value on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"color"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$color = new Element\Color('color');
$color->setLabel('Background color');

$form = new Form('my-form');
$form->add($color);
```

Here is the same example using the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Color',
     'name' => 'color',
     'options' => array(
             'label' => 'Background color'
     )
 ));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and `Zend\Filter\StringToLower` filters, and a `Zend\Validator\Regex` to validate the RGB hex format.

Return type array

112.4.2 Date

`Zend\Form\Element\Date` is meant to be paired with the `Zend\Form\View\Helper\FormDate` for HTML5 inputs with type date. This element adds filters and validators to it's input filter specification in order to validate HTML5 date input values on the server.

Basic Usage

This element automatically adds a "type" attribute of value "date".

```
use Zend\Form\Element;
use Zend\Form\Form;

$date = new Element\Date('appointment-date');
$date
    ->setLabel('Appointment Date')
    ->setAttributes(array(
        'min'  => '2012-01-01',
        'max'  => '2020-01-01',
        'step' => '1', // days; default step interval is 1 day
    ))
    ->setOptions(array(
        'format' => 'Y-m-d'
    ));

$form = new Form('my-form');
$form->add($date);
```

Here is with the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Date',
     'name' => 'appointment-date',
     'options' => array(
             'label' => 'Appointment Date',
             'format' => 'Y-m-d'
     ),
     'attributes' => array(
             'min' => '2012-01-01',
             'max' => '2020-01-01',
             'step' => '1', // days; default step interval is 1 day
     )
 ));
```

Note: Note: the min, max, and step attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

getInputSpecification()

Returns a input filter specification, which includes Zend\Filter\StringTrim and will add the appropriate validators based on the values from the min, max, and step attributes and format option. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

One difference from Zend\Form\Element\DateTime is that the Zend\Validator\DateStep validator will expect the step attribute to use an interval of days (default is 1 day).

Return type array

112.4.3 DateTime

`Zend\Form\Element\DateTime` is meant to be paired with the `Zend\Form\View\Helper\FormDateTime` for HTML5 inputs with type datetime. This element adds filters and validators to it's input filter specification in order to validate HTML5 datetime input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"datetime"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$dateTime = new Element\DateTime('appointment-date-time');
$dateTime
    ->setLabel('Appointment Date/Time')
    ->setAttributes(array(
        'min'  => '2010-01-01T00:00:00Z',
        'max'  => '2020-01-01T00:00:00Z',
        'step' => '1', // minutes; default step interval is 1 min
    ))
    ->setOptions(array(
        'format' => 'Y-m-d\TH:iP'
    ));

$form = new Form('my-form');
$form->add($dateTime);
```

Here is with the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\DateTime',
     'name' => 'appointment-date-time',
     'options' => array(
             'label' => 'Appointment Date/Time',
             'format' => 'Y-m-d\TH:iP'
     ),
     'attributes' => array(
             'min' => '2010-01-01T00:00:00Z',
             'max' => '2020-01-01T00:00:00Z',
             'step' => '1', // minutes; default step interval is 1 min
     )
 ));
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes and `format` option.

If the `min` attribute is set, a `Zend\Validator\GreaterThan` validator will be added to ensure the date value is greater than the minimum value.

If the `max` attribute is set, a `Zend\Validator\LessThanValidator` validator will be added to ensure the date value is less than the maximum value.

If the `step` attribute is set to "any", step validations will be skipped. Otherwise, a `Zend\Validator\DateStep` validator will be added to ensure the date value is within a certain interval of minutes (default is 1 minute).

The input filter specification also includes a `Zend\Validator\Date` validator to ensure the format of the value. If the `format` option is set, that format will be used. Otherwise the default format will be used.

Return type array

setOptions(*array $options*)

Set options for an element of type DateTime. The accepted option, in addition to the inherited options of *Zend\Form\Element* , is: `"format"`, which calls `setFormat`.

setFormat(*string $format*)

Sets the format used to validate the value. Accepts a `\DateTime` compatible string.

getFormat()

Return the DateTime format used to validate the value.

Return type String

112.4.4 DateTimeLocal

`Zend\Form\Element\DateTimeLocal` is meant to be paired with the `Zend\Form\View\Helper\FormDateTimeLocal` for HTML5 inputs with type datetime-local. This element adds filters and validators to it's input filter specification in order to validate HTML5 a local datetime input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"datetime-local"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$dateTimeLocal = new Element\DateTimeLocal('appointment-date-time');
$dateTimeLocal
    ->setLabel('Appointment Date')
    ->setAttributes(array(
        'min'  => '2010-01-01T00:00:00',
        'max'  => '2020-01-01T00:00:00',
        'step' => '1', // minutes; default step interval is 1 min
    ))
    ->setOptions(array(
```

```
        'format' => 'Y-m-d\TH:i'
    ));

$form = new Form('my-form');
$form->add($dateTimeLocal);
```

Here is with the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\DateTimeLocal',
     'name' => 'appointment-date-time',
     'options' => array(
             'label'  => 'Appointment Date',
             'format' => 'Y-m-d\TH:i'
     ),
     'attributes' => array(
             'min' => '2010-01-01T00:00:00',
             'max' => '2020-01-01T00:00:00',
             'step' => '1', // minutes; default step interval is 1 min
     )
 ));
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

`getInputSpecification``()`

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes and `format` option. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

Return type array

112.4.5 Email

`Zend\Form\Element\Email` is meant to be paired with the `Zend\Form\View\Helper\FormEmail` for HTML5 inputs with type email. This element adds filters and validators to it's input filter specification in order to validate HTML5 valid email address on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"email"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$form = new Form('my-form');
```

```

// Single email address
$email = new Element\Email('email');
$email->setLabel('Email Address')
$form->add($email);

// Comma separated list of emails
$emails = new Element\Email('emails');
$emails
    ->setLabel('Email Addresses')
    ->setAttribute('multiple', true);
$form->add($emails);
```

Here is with the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Email',
     'name' => 'email',
     'options' => array(
         'label' => 'Email Address'
     ),
 ));

 $form->add(array(
     'type' => 'Zend\Form\Element\Email',
     'name' => 'emails',
     'options' => array(
             'label' => 'Email Addresses'
     ),
     'attributes' => array(
         'multiple' => true
     )
 ));
```

Note: Note: the `multiple` attribute should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

`getInputSpecification``()`

Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter, and a validator based on the `multiple` attribute.

If the `multiple` attribute is unset or false, a `Zend\Validator\Regex` validator will be added to validate a single email address.

If the `multiple` attribute is true, a `Zend\Validator\Explode` validator will be added to ensure the input string value is split by commas before validating each email address with `Zend\Validator\Regex`.

Return type array

setValidator(*ValidatorInterface $validator*)
Sets the primary validator to use for this element

getValidator()
Get the primary validator

Return type ValidatorInterface

setEmailValidator(*ValidatorInterface $validator*)
Sets the email validator to use for multiple or single email addresses.

getEmailValidator()
Get the email validator to use for multiple or single email addresses.

The default Regex validator in use is to match that of the browser validation, but you are free to set a different (more strict) email validator such as `Zend\Validator\Email` if you wish.

112.4.6 Month

`Zend\Form\Element\Month` is meant to be paired with the `Zend\Form\View\Helper\FormMonth` for HTML5 inputs with type month. This element adds filters and validators to it's input filter specification in order to validate HTML5 month input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"month"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$month = new Element\Month('month');
$month
    ->setLabel('Month')
    ->setAttributes(array(
        'min'  => '2012-01',
        'max'  => '2020-01',
        'step' => '1', // months; default step interval is 1 month
    ));

$form = new Form('my-form');
$form->add($month);
```

Here is with the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Month',
     'name' => 'month',
     'options' => array(
             'label' => 'Month'
     ),
     'attributes' => array(
             'min' => '2012-12',
             'max' => '2020-01',
             'step' => '1', // months; default step interval is 1 month
```

```
        )
    ));
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

getInputSpecification()

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

One difference from `Zend\Form\Element\DateTime` is that the `Zend\Validator\DateStep` validator will expect the `step` attribute to use an interval of months (default is 1 month).

Return type array

112.4.7 Number

`Zend\Form\Element\Number` is meant to be paired with the `Zend\Form\View\Helper\FormNumber` for HTML5 inputs with type number. This element adds filters and validators to it's input filter specification in order to validate HTML5 number input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"number"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$number = new Element\Number('quantity');
$number
    ->setLabel('Quantity')
    ->setAttributes(array(
        'min'  => '0',
        'max'  => '10',
        'step' => '1', // default step interval is 1
    ));

$form = new Form('my-form');
$form->add($number);
```

Here is with the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Number',
     'name' => 'quantity',
```

```
    'options' => array(
            'label' => 'Quantity'
    ),
    'attributes' => array(
            'min' => '0',
            'max' => '10',
            'step' => '1', // default step interval is 1
    )
));
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

`getInputSpecification`()

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes.

If the `min` attribute is set, a `Zend\Validator\GreaterThan` validator will be added to ensure the number value is greater than the minimum value. The `min` value should be a valid floating point number.

If the `max` attribute is set, a `Zend\Validator\LessThan` validator will be added to ensure the number value is less than the maximum value. The `max` value should be a valid floating point number.

If the `step` attribute is set to "any", step validations will be skipped. Otherwise, a `Zend\Validator\Step` validator will be added to ensure the number value is within a certain interval (default is 1). The `step` value should be either "any" or a valid floating point number.

Return type array

112.4.8 Range

`Zend\Form\Element\Range` is meant to be paired with the `Zend\Form\View\Helper\FormRange` for HTML5 inputs with type range. This element adds filters and validators to it's input filter specification in order to validate HTML5 range values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"range"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$range = new Element\Range('range');
$range
    ->setLabel('Minimum and Maximum Amount')
    ->setAttributes(array(
        'min'  => '0',   // default minimum is 0
        'max'  => '100', // default maximum is 100
        'step' => '1',   // default interval is 1
    ));
```

```

$form = new Form('my-form');
$form->add($range);
```

Here is with the array notation:

```
use Zend\Form\Form;

$form = new Form('my-form');
$form->add(array(
    'type' => 'Zend\Form\Element\Range',
    'name' => 'range',
    'options' => array(
            'label' => 'Minimum and Maximum Amount'
    ),
    'attributes' => array(
            'min' => 0, // default minimum is 0
            'max' => 100, // default maximum is 100
            'step' => 1 // default interval is 1
    )
));
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\Number*.

`getInputSpecification`()

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\Number* for more information.

The `Range` element differs from `Zend\Form\Element\Number` in that the `Zend\Validator\GreaterThan` and `Zend\Validator\LessThan` validators will always be present. The default minimum is 1, and the default maximum is 100.

Return type array

112.4.9 Time

`Zend\Form\Element\Time` is meant to be paired with the `Zend\Form\View\Helper\FormTime` for HTML5 inputs with type time. This element adds filters and validators to it's input filter specification in order to validate HTML5 time input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"time"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$time = new Element\Time('time');
```

```
$time
    ->setLabel('Time')
    ->setAttributes(array(
        'min'  => '00:00:00',
        'max'  => '23:59:59',
        'step' => '60', // seconds; default step interval is 60 seconds
    ))
    ->setOptions(array(
        'format' => 'H:i:s'
    ));

$form = new Form('my-form');
$form->add($time);
```

Here is the same example using the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Time',
     'name' => 'time',
     'options'=> array(
             'label'  => 'Time',
             'format' => 'H:i:s'
     ),
     'attributes' => array(
             'min' => '00:00:00',
             'max' => '23:59:59',
             'step' => '60', // seconds; default step interval is 60 seconds
     )
 ));
```

Note: The `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Note: The default date format for the validator is `H:i:s`. A valid time string is however not required to have a seconds part. In fact some user agent UIs such as Google Chrome and Opera submits a value on the `H:i` format (i.e. without a second part). You might therefore want to set the date format accordingly.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

`getInputSpecification``()`

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes and `format` option. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

One difference from `Zend\Form\Element\DateTime` is that the `Zend\Validator\DateStep` validator will expect the `step` attribute to use an interval of seconds (default is 60 seconds).

Return type array

112.4.10 Url

`Zend\Form\Element\Url` is meant to be paired with the `Zend\Form\View\Helper\FormUrl` for HTML5 inputs with type url. This element adds filters and a `Zend\Validator\Uri` validator to it's input filter specification for validating HTML5 URL input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"url"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$url = new Element\Url('webpage-url');
$url->setLabel('Webpage URL');

$form = new Form('my-form');
$form->add($url);
```

Here is the same example using the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Url',
     'name' => 'webpage-url',
     'options => array(
             'label' => 'Webpage URL'
     )
 ));
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()

Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter, and a `Zend\Validator\Uri` to validate the URI string.

Return type array

112.4.11 Week

`Zend\Form\Element\Week` is meant to be paired with the `Zend\Form\View\Helper\FormWeek` for HTML5 inputs with type week. This element adds filters and validators to it's input filter specification in order to validate HTML5 week input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"week"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$week = new Element\Week('week');
$week
    ->setLabel('Week')
    ->setAttributes(array(
        'min'  => '2012-W01',
        'max'  => '2020-W01',
        'step' => '1', // weeks; default step interval is 1 week
    ));

$form = new Form('my-form');
$form->add($week);
```

Here is the same example using the array notation:

```
 use Zend\Form\Form;

 $form = new Form('my-form');
 $form->add(array(
     'type' => 'Zend\Form\Element\Week',
     'name' => 'week',
     'options' => array(
             'label' => 'Week'
     ),
     'attributes' => array(
             'min' => '2012-W01',
             'max' => '2020-W01',
             'step' => '1', // weeks; default step interval is 1 week
     )
 ));
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

`getInputSpecification`()

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

One difference from `Zend\Form\Element\DateTime` is that the `Zend\Validator\DateStep` validator will expect the `step` attribute to use an interval of weeks (default is 1 week).

Return type array

CHAPTER

THREE

FORM VIEW HELPERS

113.1 Introduction

Zend Framework comes with an initial set of helper classes related to Forms: e.g., rendering a text input, selection box, or form labels. You can use helper, or plugin, classes to perform these behaviors for you.

See the section on *view helpers* for more information.

113.2 Standard Helpers

113.2.1 Form

The `Form` view helper is used to render a `<form>` HTML element and its attributes.

Basic usage:

```
use Zend\Form\Form;
use Zend\Form\Element;

// Within your view...

$form = new Form();
// ...add elements and input filter to form...

// Set attributes
$form->setAttribute('action', $this->url('contact/process'));
$form->setAttribute('method', 'post');

// Prepare the form elements
$form->prepare();

// Render the opening tag
echo $this->form()->openTag($form);
// <form action="/contact/process" method="post">

// ...render the form elements...

// Render the closing tag
echo $this->form()->closeTag();
// </form>
```

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

openTag(*FormInterface $form = null*)
Renders the `<form>` open tag for the `$form` instance.

Return type string

closeTag()
Renders a `</form>` closing tag.

Return type string

113.2.2 FormButton

The `FormButton` view helper is used to render a `<button>` HTML element and its attributes.

Basic usage:

```
use Zend\Form\Element;

$element = new Element\Button('my-button');
$element->setLabel("Reset");

// Within your view...

/**
 * Example #1: Render entire button in one shot...
 */
echo $this->formButton($element);
// <button name="my-button" type="button">Reset</button>

/**
 * Example #2: Render button in 3 steps
 */
// Render the opening tag
echo $this->formButton()->openTag($element);
// <button name="my-button" type="button">

echo '<span class="inner">' . $element->getLabel() . '</span>';

// Render the closing tag
echo $this->formButton()->closeTag();
// </button>

/**
 * Example #3: Override the element label
 */
echo $this->formButton()->render($element, 'My Content');
// <button name="my-button" type="button">My Content</button>
```

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\FormInput*.

openTag(*$element = null*)
Renders the `<button>` open tag for the `$element` instance.

Return type string

closeTag()
Renders a `</button>` closing tag.

Return type string

render(*ElementInterface $element*[, *$buttonContent = null*])

Renders a button's opening tag, inner content, and closing tag.

Parameters

- **$element** – The button element.
- **$buttonContent** – (optional) The inner content to render. If `null`, will default to the `$element`'s label.

Return type string

113.2.3 FormCaptcha

TODO Basic usage:

```
use Zend\Captcha;
use Zend\Form\Element;

$captcha = new Element\Captcha('captcha');
$captcha
    ->setCaptcha(new Captcha\Dumb())
    ->setLabel('Please verify you are human');

// Within your view...

echo $this->formCaptcha($captcha);

// TODO
```

113.2.4 FormCheckbox

The `FormCheckbox` view helper can be used to render a `<input type="checkbox">` HTML form input. It is meant to work with the *Zend\Form\Element\Checkbox* element, which provides a default input specification for validating the checkbox values.

`FormCheckbox` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Checkbox('my-checkbox');

// Within your view...

/**
 * Example #1: Default options
 */
echo $this->formCheckbox($element);
// <input type="hidden" name="my-checkbox" value="0">
// <input type="checkbox" name="my-checkbox" value="1">

/**
 * Example #2: Disable hidden element
 */
$element->setUseHiddenElement(false);
echo $this->formCheckbox($element);
// <input type="checkbox" name="my-checkbox" value="1">

```

```
/**
 * Example #3: Change checked/unchecked values
 */
$element->setUseHiddenElement(true)
        ->setUncheckedValue('no')
        ->setCheckedValue('yes');
echo $this->formCheckbox($element);
// <input type="hidden" name="my-checkbox" value="no">
// <input type="checkbox" name="my-checkbox" value="yes">
```

113.2.5 FormCollection

TODO Basic usage:

TODO

113.2.6 FormElement

The `FormElement` view helper proxies the rendering to specific form view helpers depending on the type of the `Zend\Form\Element` that is passed in. For instance, if the passed in element had a type of "text", the `FormElement` helper will retrieve and use the `FormText` helper to render the element.

Basic usage:

```
use Zend\Form\Form;
use Zend\Form\Element;

// Within your view...

/**
 * Example #1: Render different types of form elements
 */
$textElement     = new Element\Text('my-text');
$checkboxElement = new Element\Checkbox('my-checkbox');

echo $this->formElement($textElement);
// <input type="text" name="my-text" value="">

echo $this->formElement($checkboxElement);
// <input type="hidden" name="my-checkbox" value="0">
// <input type="checkbox" name="my-checkbox" value="1">

/**
 * Example #2: Loop through form elements and render them
 */
$form = new Form();
// ...add elements and input filter to form...
$form->prepare();

// Render the opening tag
echo $this->form()->openTag($form);

// ...loop through and render the form elements...
foreach ($form as $element) {
    echo $this->formElement($element);       // <-- Magic!
    echo $this->formElementErrors($element);
```

```
}

// Render the closing tag
echo $this->form()->closeTag();
```

113.2.7 FormElementErrors

The `FormElementErrors` view helper is used to render the validation error messages of an element.

Basic usage:

```
use Zend\Form\Form;
use Zend\Form\Element;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Input;

// Create a form
$form    = new Form();
$element = new Element\Text('my-text');
$form->add($element);

// Create a input
$input = new Input('my-text');
$input->setRequired(true);

$inputFilter = new InputFilter();
$inputFilter->add($input);
$form->setInputFilter($inputFilter);

// Force a failure
$form->setData(array()); // Empty data
$form->isValid();        // Not valid

// Within your view...

/**
 * Example #1: Default options
 */
echo $this->formElementErrors($element);
// <ul><li>Value is required and can&#039;t be empty</li></ul>

/**
 * Example #2: Add attributes to open format
 */
echo $this->formElementErrors($element, array('class' => 'help-inline'));
// <ul class="help-inline"><li>Value is required and can&#039;t be empty</li></ul>

/**
 * Example #3: Custom format
 */
echo $this->formElementErrors()
                ->setMessageOpenFormat('<div class="help-inline">')
                ->setMessageSeparatorString('</div><div class="help-inline">')
                ->setMessageCloseString('</div>')
                ->render($element);
// <div class="help-inline">Value is required and can&#039;t be empty</div>
```

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

setMessageOpenFormat(*string $messageOpenFormat*)
Set the formatted string used to open message representation.

Parameters **$messageOpenFormat** – The formatted string to use to open the messages. Uses `'<ul%s><li>'` by default. Attributes are inserted here.

getMessageOpenFormat()
Returns the formatted string used to open message representation.

Return type string

setMessageSeparatorString(*string $messageSeparatorString*)
Sets the string used to separate messages.

Parameters **$messageSeparatorString** – The string to use to separate the messages. Uses `'</li><li>'` by default.

getMessageSeparatorString()
Returns the string used to separate messages.

Return type string

setMessageCloseString(*string $messageCloseString*)
Sets the string used to close message representation.

Parameters **$messageCloseString** – The string to use to close the messages. Uses `'</li></ul>'` by default.

getMessageCloseString()
Returns the string used to close message representation.

Return type string

setAttributes(*array $attributes*)
Set the attributes that will go on the message open format.

Parameters **$attributes** – Key value pairs of attributes.

getAttributes()
Returns the attributes that will go on the message open format.

Return type array

render(*ElementInterface $element*[, *array $attributes = array()*])
Renders validation errors for the provided `$element`.

Parameters

- **$element** – The element.
- **$attributes** – Additional attributes that will go on the message open format. These are merged with those set via `setAttributes()`.

Return type string

113.2.8 FormFile

The `FormFile` view helper can be used to render a `<input type="file">` form input. It is meant to work with the *Zend\Form\Element\File* element.

`FormFile` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\File('my-file');

// Within your view...

echo $this->formFile($element);
// <input type="file" name="my-file">
```

For HTML5 multiple file uploads, the `multiple` attribute can be used. Browsers that do not support HTML5 will default to a single upload input.

```
use Zend\Form\Element;

$element = new Element\File('my-file');
$element->setAttribute('multiple', true);

// Within your view...

echo $this->formFile($element);
// <input type="file" name="my-file" multiple="multiple">
```

113.2.9 FormHidden

The `FormHidden` view helper can be used to render a `<input type="hidden">` HTML form input. It is meant to work with the *Zend\Form\Element\Hidden* element.

`FormHidden` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Hidden('my-hidden');
$element->setValue('foo');

// Within your view...

echo $this->formHidden($element);
// <input type="hidden" name="my-hidden" value="foo">
```

113.2.10 FormImage

The `FormImage` view helper can be used to render a `<input type="image">` HTML form input. It is meant to work with the *Zend\Form\Element\Image* element.

`FormImage` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Image('my-image');
$element->setAttribute('src', '/img/my-pic.png');

// Within your view...

echo $this->formImage($element);
// <input type="image" name="my-image" src="/img/my-pic.png">
```

113.2.11 FormInput

The `FormInput` view helper is used to render a `<input>` HTML form input tag. It acts as a base class for all of the specifically typed form input helpers (FormText, FormCheckbox, FormSubmit, etc.), and is not suggested for direct use.

It contains a general map of valid tag attributes and types for attribute filtering. Each subclass of `FormInput` implements it's own specific map of valid tag attributes. The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

render(*ElementInterface $element*)

Renders the `<input>` tag for the `$element`.

Return type string

113.2.12 FormLabel

The `FormLabel` view helper is used to render a `<label>` HTML element and its attributes. If you have a `Zend\I18n\Translator\Translator` attached, `FormLabel` will translate the label contents during it's rendering.

Basic usage:

```
use Zend\Form\Element;

$element = new Element\Text('my-text');
$element->setLabel('Label')
        ->setAttribute('id', 'text-id')
        ->setLabelAttributes(array('class' => 'control-label'));

// Within your view...

/**
 * Example #1: Render label in one shot
 */
echo $this->formLabel($element);
// <label class="control-label" for="text-id">Label</label>

echo $this->formLabel($element, $this->formText($element));
// <label class="control-label" for="text-id">Label<input type="text" name="my-text"></label>

echo $this->formLabel($element, $this->formText($element), 'append');
// <label class="control-label" for="text-id"><input type="text" name="my-text">Label</label>

/**
 * Example #2: Render label in separate steps
 */
// Render the opening tag
echo $this->formLabel()->openTag($element);
// <label class="control-label" for="text-id">

// Render the closing tag
echo $this->formLabel()->closeTag();
// </label>
```

Attaching a translator and setting a text domain:

```
// Setting a translator
$this->formLabel()->setTranslator($translator);

// Setting a text domain
$this->formLabel()->setTranslatorTextDomain('my-text-domain');

// Setting both
$this->formLabel()->setTranslator($translator, 'my-text-domain');
```

Note: Note: If you have a translator in the Service Manager under the key, 'translator', the view helper plugin manager will automatically attach the translator to the FormLabel view helper. See `Zend\\View\\HelperPluginManager::injectTranslator()` for more information.

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

__invoke(*ElementInterface $element = null*, *string $labelContent = null*, *string $position = null*)

Render a form label, optionally with content.

Always generates a "for" statement, as we cannot assume the form input will be provided in the `$labelContent`.

Parameters

- **$element** – A form element.
- **$labelContent** – If null, will attempt to use the element's label value.
- **$position** – Append or prepend the element's label value to the `$labelContent`. One of `FormLabel::APPEND` or `FormLabel::PREPEND` (default)

Return type string

openTag(*array|ElementInterface $attributesOrElement = null*)

Renders the `<label>` open tag and attributes.

Parameters **$attributesOrElement** – An array of key value attributes or a `ElementInterface` instance.

Return type string

closeTag()

Renders a `</label>` closing tag.

Return type string

113.2.13 FormMultiCheckbox

The `FormMultiCheckbox` view helper can be used to render a group `<input type="checkbox">` HTML form inputs. It is meant to work with the *Zend\Form\Element\MultiCheckbox* element, which provides a default input specification for validating a multi checkbox.

`FormMultiCheckbox` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\MultiCheckbox('my-multicheckbox');
$element->setValueOptions(array(
   '0' => 'Apple',
   '1' => 'Orange',
   '2' => 'Lemon',
```

```
));

// Within your view...

/**
 * Example #1: using the default label placement
 */
echo $this->formMultiCheckbox($element);
// <label><input type="checkbox" name="my-multicheckbox[]" value="0">Apple</label>
// <label><input type="checkbox" name="my-multicheckbox[]" value="1">Orange</label>
// <label><input type="checkbox" name="my-multicheckbox[]" value="2">Lemon</label>

/**
 * Example #2: using the prepend label placement
 */
echo $this->formMultiCheckbox($element, 'prepend');
// <label>Apple<input type="checkbox" name="my-multicheckbox[]" value="0"></label>
// <label>Orange<input type="checkbox" name="my-multicheckbox[]" value="1"></label>
// <label>Lemon<input type="checkbox" name="my-multicheckbox[]" value="2"></label>
```

113.2.14 FormPassword

TODO Basic usage:

TODO

113.2.15 FormRadio

The `FormRadio` view helper can be used to render a group `<input type="radio">` HTML form inputs. It is meant to work with the *Zend\Form\Element\Radio* element, which provides a default input specification for validating a radio.

`FormRadio` extends from *Zend\Form\View\Helper\FormMultiCheckbox*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Radio('gender');
$element->setValueOptions(array(
   '0' => 'Male',
   '1' => 'Female',
));

// Within your view...

/**
 * Example #1: using the default label placement
 */
echo $this->formRadio($element);
// <label><input type="radio" name="gender[]" value="0">Male</label>
// <label><input type="radio" name="gender[]" value="1">Female</label>

/**
 * Example #2: using the prepend label placement
 */
echo $this->formRadio($element, 'prepend');
```

```
// <label>Male<input type="checkbox" name="gender[]" value="0"></label>
// <label>Female<input type="checkbox" name="gender[]" value="1"></label>
```

113.2.16 FormReset

TODO Basic usage:

TODO

113.2.17 FormRow

TODO Basic usage:

TODO

113.2.18 FormSelect

The `FormSelect` view helper can be used to render a group `<input type="select">` HTML form input. It is meant to work with the *Zend\Form\Element\Select* element, which provides a default input specification for validating a select.

`FormSelect` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Select('language');
$element->setValueOptions(array(
   '0' => 'French',
   '1' => 'English',
   '2' => 'Japanese',
   '3' => 'Chinese'
));

// Within your view...

/**
 * Example
 */
echo $this->formSelect($element);
```

113.2.19 FormSubmit

TODO Basic usage:

TODO

113.2.20 FormText

TODO Basic usage:

TODO

113.2.21 FormTextarea

TODO Basic usage:

TODO

113.2.22 AbstractHelper

The `AbstractHelper` is used as a base abstract class for Form view helpers, providing methods for validating form HTML attributes, as well as controlling the doctype and character encoding. `AbstractHelper` also extends from `Zend\I18n\View\Helper\AbstractTranslatorHelper` which provides an implementation for the `Zend\I18n\Translator\TranslatorAwareInterface` that allows setting a translator and text domain. The following public methods are in addition to the inherited *methods of Zend\I18n\View\Helper\AbstractTranslatorHelper*.

setDoctype(*string $doctype*)
Sets a doctype to use in the helper.

getDoctype()
Returns the doctype used in the helper.

Return type string

setEncoding(*string $encoding*)
Set the translation text domain to use in helper when translating.

getEncoding()
Returns the character encoding used in the helper.

Return type string

getId()
Returns the element id. If no ID attribute present, attempts to use the name attribute. If name attribute is also not present, returns null.

Return type string or null

113.3 HTML5 Helpers

113.3.1 FormColor

The `FormColor` view helper can be used to render a `<input type="color">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Color* element, which provides a default input specification for validating HTML5 color values.

`FormColor` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Color('my-color');

// Within your view...

echo $this->formColor($element);
// <input type="color" name="my-color" value="">
```

113.3.2 FormDate

The `FormDate` view helper can be used to render a `<input type="date">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Date* element, which provides a default input specification for validating HTML5 date values.

`FormDate` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Date('my-date');

// Within your view...

echo $this->formDate($element);
// <input type="date" name="my-date" value="">
```

113.3.3 FormDateTime

The `FormDateTime` view helper can be used to render a `<input type="datetime">` HTML5 form input. It is meant to work with the *Zend\Form\Element\DateTime* element, which provides a default input specification for validating HTML5 datetime values.

`FormDateTime` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\DateTime('my-datetime');

// Within your view...

echo $this->formDateTime($element);
// <input type="datetime" name="my-datetime" value="">
```

113.3.4 FormDateTimeLocal

The `FormDateTimeLocal` view helper can be used to render a `<input type="datetime-local">` HTML5 form input. It is meant to work with the *Zend\Form\Element\DateTimeLocal* element, which provides a default input specification for validating HTML5 datetime values.

`FormDateTimeLocal` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\DateTimeLocal('my-datetime');

// Within your view...

echo $this->formDateTimeLocal($element);
// <input type="datetime-local" name="my-datetime" value="">
```

113.3.5 FormEmail

The `FormEmail` view helper can be used to render a `<input type="email">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Email* element, which provides a default input specification with an email validator.

`FormEmail` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Email('my-email');

// Within your view...

echo $this->formEmail($element);
// <input type="email" name="my-email" value="">
```

113.3.6 FormMonth

The `FormMonth` view helper can be used to render a `<input type="month">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Month* element, which provides a default input specification for validating HTML5 date values.

`FormMonth` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Month('my-month');

// Within your view...

echo $this->formMonth($element);
// <input type="month" name="my-month" value="">
```

113.3.7 FormNumber

TODO Basic usage:

TODO

113.3.8 FormRange

TODO Basic usage:

TODO

113.3.9 FormSearch

TODO Basic usage:

TODO

113.3.10 FormTel

TODO Basic usage:

TODO

113.3.11 FormTime

The `FormTime` view helper can be used to render a `<input type="time">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Time* element, which provides a default input specification for validating HTML5 time values.

`FormTime` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Time('my-time');

// Within your view...

echo $this->formTime($element);
// <input type="time" name="my-time" value="">
```

113.3.12 FormUrl

TODO Basic usage:

TODO

113.3.13 FormWeek

The `FormWeek` view helper can be used to render a `<input type="week">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Week* element, which provides a default input specification for validating HTML5 week values.

`FormWeek` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Week('my-week');

// Within your view...

echo $this->formWeek($element);
// <input type="week" name="my-week" value="">
```

113.4 File Upload Progress Helpers

113.4.1 FormFileApcProgress

The `FormFileApcProgress` view helper can be used to render a `<input type="hidden" ...>` with a progress ID value used by the APC File Upload Progress feature. The APC php module is required for this view helper to work. Unlike other Form view helpers, the `FormFileSessionProgress` helper does not accept a Form Element as a parameter.

An `id` attribute with a value of `"progress_key"` will automatically be added.

> **Warning:** The view helper **must** be rendered *before* the file input in the form, or upload progress will not work correctly.

Best used with the *Zend\ProgressBar\Upload\ApcProgress* handler.

See the `apc.rfc1867` ini setting in the APC Configuration documentation for more information.

`FormFileApcProgress` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
// Within your view...

echo $this->formFileApcProgress();
// <input type="hidden" id="progress_key" name="APC_UPLOAD_PROGRESS" value="12345abcde">
```

113.4.2 FormFileSessionProgress

The `FormFileSessionProgress` view helper can be used to render a `<input type="hidden" ...>` which can be used by the PHP 5.4 File Upload Session Progress feature. PHP 5.4 is required for this view helper to work. Unlike other Form view helpers, the `FormFileSessionProgress` helper does not accept a Form Element as a parameter.

An `id` attribute with a value of `"progress_key"` will automatically be added.

> **Warning:** The view helper **must** be rendered *before* the file input in the form, or upload progress will not work correctly.

Best used with the *Zend\ProgressBar\Upload\SessionProgress* handler.

See the Session Upload Progress in the PHP documentation for more information.

`FormFileSessionProgress` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
// Within your view...

echo $this->formFileSessionProgress();
// <input type="hidden" id="progress_key" name="PHP_SESSION_UPLOAD_PROGRESS" value="12345abcde">
```

113.4.3 FormFileUploadProgress

The `FormFileUploadProgress` view helper can be used to render a `<input type="hidden" ...>` which can be used by the PECL uploadprogress extension. Unlike other Form view helpers, the `FormFileUploadProgress` helper does not accept a Form Element as a parameter.

An `id` attribute with a value of `"progress_key"` will automatically be added.

> **Warning:** The view helper **must** be rendered *before* the file input in the form, or upload progress will not work correctly.

Best used with the *Zend\ProgressBar\Upload\UploadProgress* handler.

See the PECL uploadprogress extension for more information.

`FormFileUploadProgress` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
// Within your view...

echo $this->formFileSessionProgress();
// <input type="hidden" id="progress_key" name="UPLOAD_IDENTIFIER" value="12345abcde">
```

CHAPTER

FOUR

OVERVIEW OF ZEND\HTTP

114.1 Overview

`Zend\Http` is a primary foundational component of Zend Framework. Since much of what PHP does is web-based, specifically HTTP, it makes sense to have a performant, extensible, concise and consistent API to do all things HTTP. In nutshell, there are several parts of `Zend\Http`:

- Context-less `Request` and `Response` classes that expose a fluent API for introspecting several aspects of HTTP messages:
  - Request line information and response status information
  - Parameters, such as those found in *POST* and *GET*
  - Message Body
  - Headers
- A Client implementation with various adapters that allow for sending requests and introspecting responses.

114.2 Zend\Http Request, Response and Headers

The Request, Response and Headers portion of the `Zend\Http` component provides a fluent, object-oriented interface for introspecting information from all the various parts of an HTTP request or HTTP response. The two main objects are `Zend\Http\Request` and `Zend\Http\Response`. These two classes are "context-less", meaning that they model a request or response in the same way whether it is presented by a client (to **send** a request and **receive** a response) or by a server (to **receive** a request and **send** a response). In other words, regardless of the context, the API remains the same for introspecting their various respective parts. Each attempts to fully model a request or response so that a developer can create these objects from a factory, or create and populate them manually.

CHAPTER

FIVE

THE REQUEST CLASS

115.1 Overview

The `Zend\Http\Request` object is responsible for providing a fluent API that allows a developer to interact with all the various parts of an HTTP request.

A typical HTTP request looks like this:

```
--------------------------
| METHOD | URI | VERSION |
--------------------------
|        HEADERS         |
--------------------------
|         BODY           |
--------------------------
```

In simplified terms, the request consists of a method, *URI* and HTTP version number which together make up the "Request Line." Next come the HTTP headers, of which there can be 0 or more. After that is the request body, which is typically used when a client wishes to send data to the server in the form of an encoded file, or include a set of POST parameters, for example. More information on the structure and specification of a HTTP request can be found in RFC-2616 on the W3.org site.

115.2 Quick Start

Request objects can either be created from the provided `fromString()` factory, or, if you wish to have a completely empty object to start with, by simply instantiating the `Zend\Http\Request` class.

```
use Zend\Http\Request;

$request = Request::fromString(<<<EOS
POST /foo HTTP/1.1
\r\n
HeaderField1: header-field-value1
HeaderField2: header-field-value2
\r\n\r\n
foo=bar&
EOS
);

// OR, the completely equivalent

$request = new Request();
```

```
$request->setMethod(Request::METHOD_POST);
$request->setUri('/foo');
$request->getHeaders()->addHeaders(array(
    'HeaderField1' => 'header-field-value1',
    'HeaderField2' => 'header-field-value2',
));
$request->getPost()->set('foo', 'bar');
```

115.3 Configuration Options

No configuration options are available.

115.4 Available Methods

Request::fromString `Request::fromString(string $string)`

A factory that produces a Request object from a well-formed HTTP Request string.

Returns `Zend\Http\Request`

setMethod `setMethod(string $method)`

Set the method for this request.

Returns `Zend\Http\Request`

getMethod `getMethod()`

Return the method for this request.

Returns string

setUri `setUri(string|Zend\Uri\Http $uri)`

Set the URI/URL for this request; this can be a string or an instance of `Zend\Uri\Http`.

Returns `Zend\Http\Request`

getUri `getUri()`

Return the URI for this request object.

Returns `Zend\Uri\Http`

getUriString `getUriString()`

Return the URI for this request object as a string.

Returns string

setVersion `setVersion(string $version)`

Set the HTTP version for this object, one of 1.0 or 1.1 (`Request::VERSION_10`, `Request::VERSION_11`).

Returns `Zend\Http\Request`

getVersion `getVersion()`

Return the HTTP version for this request.

Returns string

setQuery `setQuery(Zend\Stdlib\ParametersInterface $query)`

Provide an alternate Parameter Container implementation for query parameters in this object. (This is NOT the primary API for value setting; for that, see `getQuery()`).

Returns `Zend\Http\Request`

getQuery `getQuery(string|null $name, mixed|null $default)`

Return the parameter container responsible for query parameters or a single query parameter.

Returns `string`, `Zend\Stdlib\ParametersInterface`, or `null` depending on value of `$name` argument.

setPost `setPost(Zend\Stdlib\ParametersInterface $post)`

Provide an alternate Parameter Container implementation for POST parameters in this object. (This is NOT the primary API for value setting; for that, see `getPost()`).

Returns `Zend\Http\Request`

getPost `getPost(string|null $name, mixed|null $default)`

Return the parameter container responsible for POST parameters or a single POST parameter.

Returns `string`, `Zend\Stdlib\ParametersInterface`, or `null` depending on value of `$name` argument.

getCookie `getCookie()`

Return the Cookie header, this is the same as calling *$request->getHeaders()->get('Cookie');*.

Returns `Zend\Http\Header\Cookie`

setFiles `setFiles(Zend\Stdlib\ParametersInterface $files)`

Provide an alternate Parameter Container implementation for file parameters in this object, (This is NOT the primary API for value setting; for that, see `getFiles()`).

Returns `Zend\Http\Request`

getFiles `getFiles(string|null $name, mixed|null $default)`

Return the parameter container responsible for file parameters or a single file parameter.

Returns `string`, `Zend\Stdlib\ParametersInterface`, or `null` depending on value of `$name` argument.

setHeaders `setHeaders(Zend\Http\Headers $headers)`

Provide an alternate Parameter Container implementation for headers in this object, (this is NOT the primary API for value setting, for that see `getHeaders()`).

Returns `Zend\Http\Request`

getHeaders `getHeaders(string|null $name, mixed|null $default)`

Return the container responsible for storing HTTP headers. This container exposes the primary API for manipulating headers set in the HTTP request. See *the section on Zend\Http\Headers* for more information.

Returns `Zend\Http\Headers` if `$name` is `null`. Returns `Zend\Http\Header\HeaderInterface` or `ArrayIterator` if `$name` matches one or more stored headers, respectively.

setMetadata `setMetadata(string|int|array|Traversable $spec, mixed $value)`

Set message metadata.

Non-destructive setting of message metadata; always adds to the metadata, never overwrites the entire metadata container.

Returns `Zend\Http\Request`

getMetadata `getMetadata(null|string|int $key, null|mixed $default)`

Retrieve all metadata or a single metadatum as specified by key.

Returns mixed

setContent `setContent(mixed $value)`

Set request body (content).

Returns `Zend\Http\Request`

getContent `getContent()`

Get request body (content).

Returns mixed

isOptions `isOptions()`

Is this an OPTIONS method request?

Returns bool

isGet `isGet()`

Is this a GET method request?

Returns bool

isHead `isHead()`

Is this a HEAD method request?

Returns bool

isPost `isPost()`

Is this a POST method request?

Returns bool

isPut `isPut()`

Is this a PUT method request?

Returns bool

isDelete `isDelete()`

Is this a DELETE method request?

Returns bool

isTrace `isTrace()`

Is this a TRACE method request?

Returns bool

isConnect `isConnect()`

Is this a CONNECT method request?

Returns bool

isPatch `isPatch()`

Is this a PATCH method request?

Returns bool

isXmlHttpRequest `isXmlHttpRequest()`

Is this a Javascript XMLHttpRequest?

Returns bool

isFlashRequest `isFlashRequest()`

Is this a Flash request?

Returns bool

renderRequestLine `renderRequestLine()`

Return the formatted request line (first line) for this HTTP request.

Returns string

toString `toString()`

Returns string

__toString `__toString()`

Allow PHP casting of this object.

Returns string

115.5 Examples

Generating a Request object from a string

```
use Zend\Http\Request;

$string = "GET /foo HTTP/1.1\r\n\r\nSome Content";
$request = Request::fromString($string);

$request->getMethod();     // returns Request::METHOD_GET
$request->getUri();        // returns Zend\Uri\Http object
$request->getUriString(); // returns '/foo'
$request->getVersion();    // returns Request::VERSION_11 or '1.1'
$request->getContent();    // returns 'Some Content'
```

Retrieving and setting headers

```
use Zend\Http\Request;
use Zend\Http\Header\Cookie;

$request = new Request();
$request->getHeaders()->get('Content-Type'); // return content type
$request->getHeaders()->addHeader(new Cookie(array('foo' => 'bar')));
foreach ($request->getHeaders() as $header) {
    echo $header->getFieldName() . ' with value ' . $header->getFieldValue();
}
```

Retrieving and setting GET and POST values

```
use Zend\Http\Request;

$request = new Request();

// getPost() and getQuery() both return, by default, a Parameters object, which extends ArrayObject
$request->getPost()->foo = 'Foo value';
$request->getQuery()->bar = 'Bar value';
$request->getPost('foo'); // returns 'Foo value'
$request->getQuery()->offsetGet('bar'); // returns 'Bar value'
```

Generating a formatted HTTP Request from a Request object

```
use Zend\Http\Request;

$request = new Request();
$request->setMethod(Request::METHOD_POST);
$request->setUri('/foo');
$request->getHeaders()->addHeaders(array(
    'HeaderField1' => 'header-field-value1',
    'HeaderField2' => 'header-field-value2',
));
$request->getPost()->set('foo', 'bar');
echo $request->toString();

/** Will produce:
POST /foo HTTP/1.1
HeaderField1: header-field-value1
HeaderField2: header-field-value2

foo=bar
*/
```

CHAPTER

SIX

THE RESPONSE CLASS

116.1 Overview

The `Zend\Http\Response` class is responsible for providing a fluent API that allows a developer to interact with all the various parts of an HTTP response.

A typical HTTP Response looks like this:

```
---------------------------
| VERSION | CODE | REASON |
---------------------------
|        HEADERS          |
---------------------------
|         BODY            |
---------------------------
```

The first line of the response consists of the HTTP version, status code, and the reason string for the provided status code; this is called the Response Line. Next is a set of headers; there can be 0 or an unlimited number of headers. The remainder of the response is the response body, which is typically a string of HTML that will render on the client's browser, but which can also be a place for request/response payload data typical of an AJAX request. More information on the structure and specification of an HTTP response can be found in RFC-2616 on the W3.org site.

116.2 Quick Start

Response objects can either be created from the provided `fromString()` factory, or, if you wish to have a completely empty object to start with, by simply instantiating the `Zend\Http\Response` class.

```
use Zend\Http\Response;
$response = Response::fromString(<<<EOS
HTTP/1.0 200 OK
HeaderField1: header-field-value
HeaderField2: header-field-value2

<html>
<body>
    Hello World
</body>
</html>
EOS);

// OR

```

```
$response = new Response();
$response->setStatusCode(Response::STATUS_CODE_200);
$response->getHeaders()->addHeaders(array(
    'HeaderField1' => 'header-field-value',
    'HeaderField2' => 'header-field-value2',
);
$response->setContent(<<<EOS
<html>
<body>
    Hello World
</body>
</html>
EOS);
```

116.3 Configuration Options

No configuration options are available.

116.4 Available Methods

Response::fromString `Response::fromString(string $string)`

Populate object from string

Returns `Zend\Http\Response`

renderStatusLine `renderStatusLine()`

Render the status line header

Returns string

setHeaders `setHeaders(Zend\Http\Headers $headers)`

Provide an alternate Parameter Container implementation for headers in this object. (This is NOT the primary API for value setting; for that, see `getHeaders()`.)

Returns `Zend\Http\Request`

getHeaders `getHeaders()`

Return the container responsible for storing HTTP headers. This container exposes the primary API for manipulating headers set in the HTTP response. See *the section on Zend\Http\Headers* for more information.

Returns `Zend\Http\Headers`

setVersion `setVersion(string $version)`

Set the HTTP version for this object, one of 1.0 or 1.1 (`Request::VERSION_10`, `Request::VERSION_11`).

Returns `Zend\Http\Request`.

getVersion `getVersion()`

Return the HTTP version for this request

Returns string

setStatusCode `setStatusCode(numeric $code)`

Set HTTP status code

Returns `Zend\Http\Response`

getStatusCode `getStatusCode()`

Retrieve HTTP status code

Returns int

setReasonPhrase `setReasonPhrase(string $reasonPhrase)`

Set custom HTTP status message

Returns `Zend\Http\Response`

getReasonPhrase `getReasonPhrase()`

Get HTTP status message

Returns string

isClientError `isClientError()`

Does the status code indicate a client error?

Returns bool

isForbidden `isForbidden()`

Is the request forbidden due to ACLs?

Returns bool

isInformational `isInformational()`

Is the current status "informational"?

Returns bool

isNotFound `isNotFound()`

Does the status code indicate the resource is not found?

Returns bool

isOk `isOk()`

Do we have a normal, OK response?

Returns bool

isServerError `isServerError()`

Does the status code reflect a server error?

Returns bool

isRedirect `isRedirect()`

Do we have a redirect?

Returns bool

isSuccess `isSuccess()`

Was the response successful?

Returns bool

decodeChunkedBody `decodeChunkedBody(string $body)`

Decode a "chunked" transfer-encoded body and return the decoded text

Returns string

decodeGzip `decodeGzip(string $body)`

Decode a gzip encoded message (when Content-encoding = gzip)

Currently requires PHP with zlib support

Returns string

decodeDeflate `decodeDeflate(string $body)`

Decode a zlib deflated message (when Content-encoding = deflate)

Currently requires PHP with zlib support

Returns string

setMetadata `setMetadata(string|int|array|Traversable $spec, mixed $value)`

Set message metadata

Non-destructive setting of message metadata; always adds to the metadata, never overwrites the entire metadata container.

Returns `Zend\Stdlib\Message`

getMetadata `getMetadata(null|string|int $key, null|mixed $default)`

Retrieve all metadata or a single metadatum as specified by key

Returns mixed

setContent `setContent(mixed $value)`

Set message content

Returns `Zend\Stdlib\Message`

getContent `getContent()`

Get message content

Returns mixed

toString `toString()`

Returns string

116.5 Examples

Generating a Response object from a string

```
use Zend\Http\Response;
$request = Response::fromString(<<<EOS
HTTP/1.0 200 OK
HeaderField1: header-field-value
HeaderField2: header-field-value2

<html>
<body>
```

```
    Hello World
</body>
</html>
EOS);
```

Generating a formatted HTTP Response from a Response object

```
use Zend\Http\Response;
$response = new Response();
$response->setStatusCode(Response::STATUS_CODE_200);
$response->getHeaders()->addHeaders(array(
    'HeaderField1' => 'header-field-value',
    'HeaderField2' => 'header-field-value2',
);
$response->setContent(<<<EOS
<html>
<body>
    Hello World
</body>
</html>
EOS);
```

CHAPTER

SEVEN

THE HEADERS CLASS

117.1 Overview

The `Zend\Http\Headers` class is a container for HTTP headers. It is typically accessed as part of a `Zend\Http\Request` or `Zend\Http\Response` `getHeaders()` call. The Headers container will lazily load actual Header objects as to reduce the overhead of header specific parsing.

The `Zend\Http\Header\*` classes are the domain specific implementations for the various types of Headers that one might encounter during the typical HTTP request. If a header of unknown type is encountered, it will be implemented as a `Zend\Http\Header\GenericHeader` instance. See the below table for a list of the various HTTP headers and the API that is specific to each header type.

117.2 Quick Start

The quickest way to get started interacting with header objects is by getting an already populated Headers container from a request or response object.

```
// $client is an instance of Zend\Http\Client

// You can retrieve the request headers by first retrieving
// the Request object and then calling getHeaders on it
$requestHeaders  = $client->getRequest()->getHeaders();

// The same method also works for retrieving Response headers
$responseHeaders = $client->getResponse()->getHeaders();
```

`Zend\Http\Headers` can also extract headers from a string:

```
$headerString = <<<EOB
Host: www.example.com
Content-Type: text/html
Content-Length: 1337
EOB;

$headers = Zend\Http\Headers::fromString($headerString);
// $headers is now populated with three objects
//   (1) Zend\Http\Header\Host
//   (2) Zend\Http\Header\ContentType
//   (3) Zend\Http\Header\ContentLength
```

Now that you have an instance of `Zend\Http\Headers` you can manipulate the individual headers it contains using the provided public API methods outlined in the "*Available Methods*" section.

117.3 Configuration Options

No configuration options are available.

117.4 Available Methods

Headers::fromString `Headers::fromString(string $string)`

Populates headers from string representation

Parses a string for headers, and aggregates them, in order, in the current instance, primarily as strings until they are needed (they will be lazy loaded).

Returns `Zend\Http\Headers`

setPluginClassLoader `setPluginClassLoader(Zend\Loader\PluginClassLocator $pluginClassLoader)`

Set an alternate implementation for the plugin class loader

Returns `Zend\Http\Headers`

getPluginClassLoader `getPluginClassLoader()`

Return an instance of a `Zend\Loader\PluginClassLocator`, lazyload and inject map if necessary.

Returns `Zend\Loader\PluginClassLocator`

addHeaders `addHeaders(array|Traversable $headers)`

Add many headers at once

Expects an array (or `Traversable` object) of type/value pairs.

Returns `Zend\Http\Headers`

addHeaderLine `addHeaderLine(string $headerFieldNameOrLine, string $fieldValue)`

Add a raw header line, either in name => value, or as a single string 'name: value'

This method allows for lazy-loading in that the parsing and instantiation of Header object will be delayed until they are retrieved by either `get()` or `current()`.

Returns `Zend\Http\Headers`

addHeader `addHeader(Zend\Http\Header\HeaderInterface $header)`

Add a Header to this container, for raw values see `addHeaderLine()` and `addHeaders()`.

Returns `Zend\Http\Headers`

removeHeader `removeHeader(Zend\Http\Header\HeaderInterface $header)`

Remove a Header from the container

Returns bool

clearHeaders `clearHeaders()`

Clear all headers

Removes all headers from queue

Returns `Zend\Http\Headers`

get `get(string $name)`

Get all headers of a certain name/type

Returns false| `Zend\Http\Header\HeaderInterface`| `ArrayIterator`

has `has(string $name)`

Test for existence of a type of header

Returns bool

next `next()`

Advance the pointer for this object as an iterator

Returns void

key `key()`

Return the current key for this object as an iterator

Returns mixed

valid `valid()`

Is this iterator still valid?

Returns bool

rewind `rewind()`

Reset the internal pointer for this object as an iterator

Returns void

current `current()`

Return the current value for this iterator, lazy loading it if need be

Returns `Zend\Http\Header\HeaderInterface`

count `count()`

Return the number of headers in this container. If all headers have not been parsed, actual count could increase if MultipleHeader objects exist in the Request/Response. If you need an exact count, iterate.

Returns int

toString `toString()`

Render all headers at once

This method handles the normal iteration of headers; it is up to the concrete classes to prepend with the appropriate status/request line.

Returns string

toArray `toArray()`

Return the headers container as an array

Returns array

forceLoading `forceLoading()`

By calling this, it will force parsing and loading of all headers, after this `count()` will be accurate

Returns bool

117.5 Zend\Http\Header* Base Methods

fromString `fromString(string $headerLine)`

Factory to generate a header object from a string

Returns `Zend\Http\Header\GenericHeader`

getFieldName `getFieldName()`

Retrieve header field name

Returns string

getFieldValue `getFieldValue()`

Retrieve header field value

Returns string

toString `toString()`

Cast to string as a well formed HTTP header line

Returns in form of "NAME: VALUE\r\n"

Returns string

117.6 List of HTTP Header Types

Class Name	Additional Methods
Accept	N/A
AcceptCharset	N/A
AcceptEncoding	N/A
AcceptLanguage	N/A
AcceptRanges	getRangeUnit() / setRangeUnit() - The range unit of the accept ranges header
Age	getDeltaSeconds() / setDeltaSeconds() - The age in delta seconds
Allow	getAllowedMethods() / setAllowedMethods() - An array of allowed methods
AuthenticationInfo	N/A
Authorization	N/A
CacheControl	N/A
Connection	N/A
ContentDisposition	N/A
ContentEncoding	N/A
ContentLanguage	N/A
ContentLength	N/A
ContentLocation	N/A
ContentMD5	N/A
ContentRange	N/A
ContentType	N/A
Cookie	Extends \ArrayObject setEncodeValue() / getEncodeValue() - Whether or not to encode values
Date	N/A
Etag	N/A

Class Name	Additional Methods
Expect	N/A
Expires	N/A
From	N/A
Host	N/A
IfMatch	N/A
IfModifiedSince	N/A
IfNoneMatch	N/A
IfRange	N/A
IfUnmodifiedSince	N/A
KeepAlive	N/A
LastModified	N/A
Location	N/A
MaxForwards	N/A
Pragma	N/A
ProxyAuthenticate	N/A
ProxyAuthorization	N/A
Range	N/A
Referer	N/A
Refresh	N/A
RetryAfter	N/A
Server	N/A
SetCookie	getName() / setName() - The cookie name getValue() / setValue() - The cookie value getExpires() / setExpires(
TE	N/A
Trailer	N/A
TransferEncoding	N/A
Upgrade	N/A
UserAgent	N/A
Vary	N/A
Via	N/A
Warning	N/A
WWWAuthenticate	N/A

117.7 Examples

Retrieving headers from a Zend\Http\Headers object

```
// $client is an instance of Zend\Http\Client
$response = $client->send();
$headers = $response->getHeaders();

// We can check if the Request contains a specific header by
// using the ``has`` method. Returns boolean ``TRUE`` if at least
// one matching header found, and ``FALSE`` otherwise
$headers->has('Content-Type');

// We can retrieve all instances of a specific header by using
// the ``get`` method:
$contentTypeHeaders = $headers->get('Content-Type');
```

There are three possibilities for the return value of the above call to the `get` method:

- If no Content-Type header was set in the Request, `get` will return false.
- If only one Content-Type header was set in the Request, `get` will return an instance of `Zend\Http\Header\ContentType`.
- If more than one Content-Type header was set in the Request, `get` will return an ArrayIterator containing one `Zend\Http\Header\ContentType` instance per header.

Adding headers to a Zend\Http\Headers object

```
$headers = new Zend\Http\Headers();

// We can directly add any object that implements Zend\Http\Header\HeaderInterface
$typeHeader = Zend\Http\Header\ContentType::fromString('Content-Type: text/html');
$headers->addHeader($typeHeader);

// We can add headers using the raw string representation, either
// passing the header name and value as separate arguments...
$headers->addHeaderLine('Content-Type', 'text/html');

// .. or we can pass the entire header as the only argument
$headers->addHeaderLine('Content-Type: text/html');

// We can also add headers in bulk using addHeaders, which accepts
// an array of individual header definitions that can be in any of
// the accepted formats outlined below:
$headers->addHeaders(array(

    // An object implementing Zend\Http\Header\HeaderInterface
    Zend\Http\Header\ContentType::fromString('Content-Type: text/html'),

    // A raw header string
    'Content-Type: text/html',

    // We can also pass the header name as the array key and the
    // header content as that array key's value
    'Content-Type' => 'text/html');

));
```

Removing headers from a Zend\Http\Headers object

We can remove all headers of a specific type using the `removeHeader` method, which accepts a single object implementing `Zend\Http\Header\HeaderInterface`

```
// $headers is a pre-configured instance of Zend\Http\Headers

// We can also delete individual headers or groups of headers
$matches = $headers->get('Content-Type');

// If more than one header was found, iterate over the collection
// and remove each one individually
if ($matches instanceof ArrayIterator) {
    foreach ($headers as $header) {
        $headers->removeHeader($header);
    }
```

```
// If only a single header was found, remove it directly
} elseif ($matches instanceof Zend\Http\Header\HeaderInterface) {
    $headers->removeHeader($header);
}

// In addition to this, we can clear all the headers currently stored in
// the container by calling the clearHeaders() method
$matches->clearHeaders();
```

CHAPTER

EIGHT

HTTP CLIENT - OVERVIEW

118.1 Overview

`Zend\Http\Client` provides an easy interface for performing Hyper-Text Transfer Protocol (HTTP) requests. `Zend\Http\Client` supports the most simple features expected from an *HTTP* client, as well as some more complex features such as *HTTP* authentication and file uploads. Successful requests (and most unsuccessful ones too) return a `Zend\Http\Response` object, which provides access to the response's headers and body (see *this section*).

118.2 Quick Start

The class constructor optionally accepts a URL as its first parameter (can be either a string or a `Zend\Uri\Http` object), and an array or `Zend\Config\Config` object containing configuration options. Both can be left out, and set later using the setUri() and setConfig() methods.

```
use Zend\Http\Client;
$client = new Client('http://example.org', array(
    'maxredirects' => 0,
    'timeout'      => 30
));

// This is actually exactly the same:
$client = new Client();
$client->setUri('http://example.org');
$client->setOptions(array(
    'maxredirects' => 0,
    'timeout'      => 30
));

// You can also pass a Zend\Config\Config object to set the client's configuration
$config = Zend\Config\Factory::fromFile('httpclient.ini');
$client->setOptions($config);
```

Note: `Zend\Http\Client` uses `Zend\Uri\Http` to validate URLs. See the *Zend\Uri manual page* for more information on the validation process.

118.3 Configuration Options

The constructor and setOptions() method accept an associative array of configuration parameters, or a `Zend\Config\Config` object. Setting these parameters is optional, as they all have default values.

Table 118.1: Zend\Http\Client configuration parameters

Parameter	Description	Expected Values	Default Value
maxredirects	Maximum number of redirections to follow (0 = none)	integer	5
strictredirects	Whether to strictly follow the RFC when redirecting (see this section)	boolean	FALSE
useragent	User agent identifier string (sent in request headers)	string	'Zend\Http\Client'
timeout	Connection timeout (seconds)	integer	10
httpversion	HTTP protocol version (usually '1.1' or '1.0')	string	'1.1'
adapter	Connection adapter class to use (see this section)	mixed	'Zend\Http\Client\Adapter\Socket'
keepalive	Whether to enable keep-alive connections with the server. Useful and might improve performance if several consecutive requests to the same server are performed.	boolean	FALSE
storeresponse	Whether to store last response for later retrieval with getLastResponse(). If set to FALSEgetLastResponse() will return NULL.	boolean	TRUE
encodecookies	Whether to pass the cookie value through urlencode/urldecode. Enabling this breaks support with some web servers. Disabling this limits the range of values the cookies can contain.	boolean	TRUE
outputstream	Destination for streaming of received data (options: string (filename), true for temp file, false/null to disable streaming)	boolean	FALSE
rfc3986strict	Whether to strictly adhere to RFC 3986 (in practice, this means replacing '+' with '%20')	boolean	FALSE

118.4 Available Methods

__construct `__construct(string $uri, array|Traversable $config)`

Constructor

Returns void

setOptions `setOptions(array|Traversable $config = array ())`

Set configuration parameters for this HTTP client

Returns Zend\Http\Client

setAdapter `setAdapter(Zend\Http\Client\Adapter|string $adapter)`

Load the connection adapter

While this method is not called more than once for a client, it is separated from ->send() to preserve logic and readability

Returns Zend\Http\Client

getAdapter `getAdapter()`

Retrieve the connection adapter

Returns Zend\Http\Client\Adapter\AdapterInterface

setRequest `setRequest(Zend\Http\Request $request)`

Set request object

Returns void

getRequest `getRequest()`

Get Request object

Returns Zend\Http\Request

getLastRawRequest `getLastRawRequest()`

Get the last request (as a string)

Returns string

setResponse `setResponse(Zend\Http\Response $response)`

Set response

Returns Zend\Http\Client

getResponse `getResponse()`

Get Response object

Returns Zend\Http\Response

getLastRawResponse `getLastRawResponse()`

Get the last response (as a string)

Returns string

getRedirectionsCount `getRedirectionsCount()`

Get the redirections count

Returns integer

setUri `setUri(string|Zend\Http\Zend\Uri\Http $uri)`

Set Uri (to the request)

Returns Zend\Http\Client

getUri `getUri()`

Get uri (from the request)

Returns Zend\Uri\Http

setMethod `setMethod(string $method)`

Set the HTTP method (to the request)

Returns Zend\Http\Client

getMethod `getMethod()`

Get the HTTP method

Returns string

setEncType `setEncType(string $encType, string $boundary)`

Set the encoding type and the boundary (if any)

Returns void

getEncType `getEncType()`

Get the encoding type

Returns type

setRawBody `setRawBody(string $body)`

Set raw body (for advanced use cases)

Returns Zend\Http\Client

setParameterPost `setParameterPost(array $post)`

Set the POST parameters

Returns Zend\Http\Client

setParameterGet `setParameterGet(array $query)`

Set the GET parameters

Returns Zend\Http\Client

getCookies `getCookies()`

Return the current cookies

Returns array

addCookie `addCookie(ArrayIterator|SetCookie|string $cookie, string $value = null, string $expire = null, string $path = null, string $domain = null, boolean $secure = false, boolean $httponly = true, string $maxAge = null, string $version = null)`

Add a cookie

Returns Zend\Http\Client

setCookies `setCookies(array $cookies)`

Set an array of cookies

Returns Zend\Http\Client

clearCookies `clearCookies()`

Clear all the cookies

Returns void

setHeaders `setHeaders(Zend\Http\Headers|array $headers)`

Set the headers (for the request)

Returns Zend\Http\Client

hasHeader `hasHeader(string $name)`

Check if exists the header type specified

Returns boolean

getHeader `getHeader(string $name)`

Get the header value of the request

Returns string|boolean

setStream `setStream(string|boolean $streamfile = true)`

Set streaming for received data

Returns Zend\Http\Client

getStream `getStream()`

Get status of streaming for received data

Returns boolean|string

setAuth `setAuth(string $user, string $password, string $type = 'basic')`

Create a HTTP authentication "Authorization:" header according to the specified user, password and authentication method.

Returns Zend\Http\Client

resetParameters `resetParameters()`

Reset all the HTTP parameters (auth,cookies,request, response, etc)

Returns void

dispatch `dispatch(Zend\Stdlib\RequestInterface $request, Zend\Stdlib\ResponseInterface $response= null)`

Dispatch HTTP request

Returns Response

send `send(Zend\Http\Request $request)`

Send HTTP request

Returns Response

setFileUpload `setFileUpload(string $filename, string $formname, string $data = null, string $ctype = null)`

Set a file to upload (using a POST request)

Can be used in two ways: 1. $data is null (default): $filename is treated as the name if a local file which will be read and sent. Will try to guess the content type using mime_content_type(). 2. $data is set - $filename is sent as the file name, but $data is sent as the file contents and no file is read from the file system. In this case, you need to manually set the Content-Type ($ctype) or it will default to application/octet-stream.

Returns Zend\Http\Client

removeFileUpload `removeFileUpload(string $filename)`

Remove a file to upload

Returns boolean

encodeFormData `encodeFormData(string $boundary, string $name, mixed $value, string $filename = null, array $headers = array ( ))`

Encode data to a multipart/form-data part suitable for a POST request.

Returns string

118.5 Examples

118.5.1 Performing a Simple GET Request

Performing simple *HTTP* requests is very easily done using the setRequest() and dispatch() methods:

```
use Zend\Http\Client;
use Zend\Http\Request;

$request = new Request();
$client = new Client('http://example.org');
$client->setRequest($request);
$response = $client->dispatch();
```

The `request` object can be configured using his methods as shown in the *Zend\Http\Request manual page*. One of these methods is `setMethod` which refers to the HTTP Method. This can be either `GET`, `POST`, `PUT`, `HEAD`, `DELETE`, `TRACE`, `OPTIONS` or `CONNECT` as defined by the *HTTP* protocol [1].

118.5.2 Using Request Methods Other Than GET

For convenience, these are all defined as class constants: Zend\Http\Request::METHOD_GET, Zend\Http\Request::METHOD_POST and so on.

If no method is specified, the method set by the last `setMethod()` call is used. If `setMethod()` was never called, the default request method is `GET` (see the above example).

```
use Zend\Http\Client;
use Zend\Http\Request;

$request = new Request();
$request->setUri('http://example.org');
$client = new Client();

// Performing a POST request
$request->setMethod(Request::METHOD_POST);
$client->setRequest($request);
$response = $client->dispatch();
```

118.5.3 Setting GET parameters

Adding `GET` parameters to an *HTTP* request is quite simple, and can be done either by specifying them as part of the URL, or by using the `setParameterGet()` method. This method takes the `GET` parameters as an associative array of name => value `GET` variables.

[1] See RFC 2616 -http://www.w3.org/Protocols/rfc2616/rfc2616.html.

```
use Zend\Http\Client;
$client = new Client();

// This is equivalent to setting a URL in the Client's constructor:
$client->setUri('http://example.com/index.php?knight=lancelot');

// Adding several parameters with one call
$client->setParameterGet(array(
   'first_name'  => 'Bender',
   'middle_name' => 'Bending',
   'last_name'   => 'Rodríguez',
   'made_in'     => 'Mexico',
));
```

118.5.4 Setting POST Parameters

While `GET` parameters can be sent with every request method, `POST` parameters are only sent in the body of `POST` requests. Adding `POST` parameters to a request is very similar to adding `GET` parameters, and can be done with the `setParameterPost()` method, which is identical to the `setParameterGet()` method in structure.

```
use Zend\Http\Client;
$client = new Client();

// Setting several POST parameters, one of them with several values
$client->setParameterPost(array(
    'language'  => 'es',
    'country'   => 'ar',
    'selection' => array(45, 32, 80)
));
```

Note that when sending `POST` requests, you can set both `GET` and `POST` parameters. On the other hand, setting POST parameters on a non-`POST` request will not trigger an error, rendering it useless. Unless the request is a `POST` request, `POST` parameters are simply ignored.

118.5.5 A Complete Example

```
use Zend\Http\Request;
use Zend\Http\Client;
$request = new Request();
$request->setUri('http://www.test.com');
$request->setMethod('POST');
$request->getPost()->set('foo', 'bar');

$client = new Client();
$response = $client->dispatch($request);

if ($response->isSuccess()) {
    //  the POST was successful
}
```

CHAPTER

NINE

HTTP CLIENT - CONNECTION ADAPTERS

119.1 Overview

`Zend\Http\Client` is based on a connection adapter design. The connection adapter is the object in charge of performing the actual connection to the server, as well as writing requests and reading responses. This connection adapter can be replaced, and you can create and extend the default connection adapters to suite your special needs, without the need to extend or replace the entire *HTTP* client class, and with the same interface.

Currently, the `Zend\Http\Client` class provides four built-in connection adapters:

- `Zend\Http\Client\Adapter\Socket` (default)
- `Zend\Http\Client\Adapter\Proxy`
- `Zend\Http\Client\Adapter\Curl`
- `Zend\Http\Client\Adapter\Test`

The `Zend\Http\Client` object's adapter connection adapter is set using the 'adapter' configuration option. When instantiating the client object, you can set the 'adapter' configuration option to a string containing the adapter's name (eg. 'Zend\Http\Client\Adapter\Socket') or to a variable holding an adapter object (eg. `new Zend\Http\Client\Adapter\Socket`). You can also set the adapter later, using the `Zend\Http\Client->setAdapter()` method.

119.2 The Socket Adapter

The default connection adapter is the `Zend\Http\Client\Adapter\Socket` adapter - this adapter will be used unless you explicitly set the connection adapter. The Socket adapter is based on *PHP*'s built-in fsockopen() function, and does not require any special extensions or compilation flags.

The Socket adapter allows several extra configuration options that can be set using `Zend\Http\Client->setOptions()` or passed to the client constructor.

Table 119.1: Zend\Http\Client\Adapter\Socket configuration parameters

Parameter	Description	Expected Type	Default Value
persistent	Whether to use persistent TCP connections	boolean	FALSE
ssltransport	SSL transport layer (eg. 'sslv2', 'tls')	string	ssl
sslcert	Path to a PEM encoded SSL certificate	string	NULL
sslpassphrase	Passphrase for the SSL certificate file	string	NULL
sslverifypeer	Whether to verify the SSL peer	string	NULL
sslcapath	Path to SSL certificate directory	string	NULL
sslallowselfsigned	Whether to allow self-signed certificates	string	NULL
sslusecontext	Enables proxied connections to use SSL even if the proxy connection itself does not.	boolean	FALSE

Note: Persistent TCP Connections

Using persistent *TCP* connections can potentially speed up *HTTP* requests - but in most use cases, will have little positive effect and might overload the *HTTP* server you are connecting to.

It is recommended to use persistent *TCP* connections only if you connect to the same server very frequently, and are sure that the server is capable of handling a large number of concurrent connections. In any case you are encouraged to benchmark the effect of persistent connections on both the client speed and server load before using this option.

Additionally, when using persistent connections it is recommended to enable Keep-Alive *HTTP* requests as described in *the configuration section*- otherwise persistent connections might have little or no effect.

Note: HTTPS SSL Stream Parameters

`ssltransport`, `sslcert` and `sslpassphrase` are only relevant when connecting using *HTTPS*.

While the default *SSL* settings should work for most applications, you might need to change them if the server you are connecting to requires special client setup. If so, you should read the sections about *SSL* transport layers and options here.

Changing the HTTPS transport layer

```
// Set the configuration parameters
$config = array(
    'adapter'      => 'Zend\Http\Client\Adapter\Socket',
    'ssltransport' => 'tls'
);

// Instantiate a client object
$client = new Zend\Http\Client('https://www.example.com', $config);

// The following request will be sent over a TLS secure connection.
$response = $client->send();
```

The result of the example above will be similar to opening a *TCP* connection using the following *PHP* command:

```
fsockopen('tls://www.example.com', 443)
```

119.2.1 Customizing and accessing the Socket adapter stream context

`Zend\Http\Client\Adapter\Socket` provides direct access to the underlying stream context used to connect to the remote server. This allows the user to pass specific options and parameters to the *TCP* stream, and to the *SSL* wrapper in case of *HTTPS* connections.

You can access the stream context using the following methods of `Zend\Http\Client\Adapter\Socket`:

- **setStreamContext($context)** Sets the stream context to be used by the adapter. Can accept either a stream context resource created using the stream_context_create() *PHP* function, or an array of stream context options, in the same format provided to this function. Providing an array will create a new stream context using these options, and set it.
- **getStreamContext()** Get the stream context of the adapter. If no stream context was set, will create a default stream context and return it. You can then set or get the value of different context options using regular *PHP* stream context functions.

Setting stream context options for the Socket adapter

```
// Array of options
$options = array(
    'socket' => array(
        // Bind local socket side to a specific interface
        'bindto' => '10.1.2.3:50505'
    ),
    'ssl' => array(
        // Verify server side certificate,
        // do not accept invalid or self-signed SSL certificates
        'verify_peer' => true,
        'allow_self_signed' => false,

        // Capture the peer's certificate
        'capture_peer_cert' => true
    )
);

// Create an adapter object and attach it to the HTTP client
$adapter = new Zend\Http\Client\Adapter\Socket();
$client = new Zend\Http\Client();
$client->setAdapter($adapter);

// Method 1: pass the options array to setStreamContext()
$adapter->setStreamContext($options);

// Method 2: create a stream context and pass it to setStreamContext()
$context = stream_context_create($options);
$adapter->setStreamContext($context);

// Method 3: get the default stream context and set the options on it
$context = $adapter->getStreamContext();
stream_context_set_option($context, $options);

// Now, perform the request
$response = $client->send();
```

```

// If everything went well, you can now access the context again
$opts = stream_context_get_options($adapter->getStreamContext());
echo $opts['ssl']['peer_certificate'];
```

Note: Note that you must set any stream context options before using the adapter to perform actual requests. If no context is set before performing *HTTP* requests with the Socket adapter, a default stream context will be created. This context resource could be accessed after performing any requests using the `getStreamContext()` method.

119.3 The Proxy Adapter

The `Zend\Http\Client\Adapter\Proxy` adapter is similar to the default Socket adapter - only the connection is made through an *HTTP* proxy server instead of a direct connection to the target server. This allows usage of `Zend\Http\Client` behind proxy servers - which is sometimes needed for security or performance reasons.

Using the Proxy adapter requires several additional configuration parameters to be set, in addition to the default 'adapter' option:

`proxy_host` should always be set - if it is not set, the client will fall back to a direct connection using `Zend\Http\Client\Adapter\Socket`. `proxy_port` defaults to '8080' - if your proxy listens on a different port you must set this one as well.

`proxy_user` and `proxy_pass` are only required if your proxy server requires you to authenticate. Providing these will add a 'Proxy-Authentication' header to the request. If your proxy does not require authentication, you can leave these two options out.

`proxy_auth` sets the proxy authentication type, if your proxy server requires authentication. Possibly values are similar to the ones accepted by the `Zend\Http\Client::setAuth()` method. Currently, only basic authentication (`Zend\Http\Client::AUTH_BASIC`) is supported.

Using Zend\Http\Client behind a proxy server

```
// Set the configuration parameters
$config = array(
    'adapter'    => 'Zend\Http\Client\Adapter\Proxy',
    'proxy_host' => 'proxy.int.zend.com',
    'proxy_port' => 8000,
    'proxy_user' => 'shahar.e',
    'proxy_pass' => 'bananashaped'
);

// Instantiate a client object
$client = new Zend\Http\Client('http://www.example.com', $config);

// Continue working...
```

As mentioned, if `proxy_host` is not set or is set to a blank string, the connection will fall back to a regular direct connection. This allows you to easily write your application in a way that allows a proxy to be used optionally, according to a configuration parameter.

Note: Since the proxy adapter inherits from `Zend\Http\Client\Adapter\Socket`, you can use the stream context access method (see *this section*) to set stream context options on Proxy connections as demonstrated above.

119.4 The cURL Adapter

cURL is a standard *HTTP* client library that is distributed with many operating systems and can be used in *PHP* via the cURL extension. It offers functionality for many special cases which can occur for a *HTTP* client and make it a perfect choice for a *HTTP* adapter. It supports secure connections, proxy, all sorts of authentication mechanisms and shines in applications that move large files around between servers.

Setting cURL options

```
$config = array(
    'adapter'   => 'Zend\Http\Client\Adapter\Curl',
    'curloptions' => array(CURLOPT_FOLLOWLOCATION => true),
);
$client = new Zend\Http\Client($uri, $config);
```

By default the cURL adapter is configured to behave exactly like the Socket Adapter and it also accepts the same configuration parameters as the Socket and Proxy adapters. You can also change the cURL options by either specifying the 'curloptions' key in the constructor of the adapter or by calling `setCurlOption($name, $value)`. The `$name` key corresponds to the CURL_* constants of the cURL extension. You can get access to the Curl handle by calling `$adapter->getHandle();`

Transfering Files by Handle

You can use cURL to transfer very large files over *HTTP* by filehandle.

```
$putFileSize   = filesize("filepath");
$putFileHandle = fopen("filepath", "r");

$adapter = new Zend\Http\Client\Adapter\Curl();
$client = new Zend\Http\Client();
$client->setAdapter($adapter);
$client->setMethod('PUT');
$adapter->setOptions(array(
    'curloptions' => array(
        CURLOPT_INFILE => $putFileHandle,
        CURLOPT_INFILESIZE => $putFileSize
    )
));
$client->send();
```

119.5 The Test Adapter

Sometimes, it is very hard to test code that relies on *HTTP* connections. For example, testing an application that pulls an *RSS* feed from a remote server will require a network connection, which is not always available.

For this reason, the `Zend\Http\Client\Adapter\Test` adapter is provided. You can write your application to use `Zend\Http\Client`, and just for testing purposes, for example in your unit testing suite, you can replace

the default adapter with a Test adapter (a mock object), allowing you to run tests without actually performing server connections.

The `Zend\Http\Client\Adapter\Test` adapter provides an additional method, `setResponse()`. This method takes one parameter, which represents an *HTTP* response as either text or a `Zend\Http\Response` object. Once set, your Test adapter will always return this response, without even performing an actual *HTTP* request.

Testing Against a Single HTTP Response Stub

```
// Instantiate a new adapter and client
$adapter = new Zend\Http\Client\Adapter\Test();
$client = new Zend\Http\Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Set the expected response
$adapter->setResponse(
    "HTTP/1.1 200 OK"        . "\r\n" .
    "Content-type: text/xml" . "\r\n" .
                               "\r\n" .
    '<?xml version="1.0" encoding="UTF-8"?>' .
    '<rss version="2.0" ' .
    '     xmlns:content="http://purl.org/rss/1.0/modules/content/"' .
    '     xmlns:wfw="http://wellformedweb.org/CommentAPI/"' .
    '     xmlns:dc="http://purl.org/dc/elements/1.1/">' .
    '  <channel>' .
    '    <title>Premature Optimization</title>' .
    // and so on...
    '</rss>');

$response = $client->send();
// .. continue parsing $response..
```

The above example shows how you can preset your *HTTP* client to return the response you need. Then, you can continue testing your own code, without being dependent on a network connection, the server's response, etc. In this case, the test would continue to check how the application parses the *XML* in the response body.

Sometimes, a single method call to an object can result in that object performing multiple *HTTP* transactions. In this case, it's not possible to use setResponse() alone because there's no opportunity to set the next response(s) your program might need before returning to the caller.

Testing Against Multiple HTTP Response Stubs

```
// Instantiate a new adapter and client
$adapter = new Zend\Http\Client\Adapter\Test();
$client = new Zend\Http\Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Set the first expected response
$adapter->setResponse(
    "HTTP/1.1 302 Found"      . "\r\n" .
    "Location: /"             . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
```

```
    '  <head><title>Moved</title></head>' .
    '  <body><p>This page has moved.</p></body>' .
    '</html>');

// Set the next successive response
$adapter->addResponse(
    "HTTP/1.1 200 OK"          . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
    '  <head><title>My Pet Store Home Page</title></head>' .
    '  <body><p>...</p></body>' .
    '</html>');

// inject the http client object ($client) into your object
// being tested and then test your object's behavior below
```

The `setResponse()` method clears any responses in the `Zend\Http\Client\Adapter\Test`'s buffer and sets the first response that will be returned. The `addResponse()` method will add successive responses.

The responses will be replayed in the order that they were added. If more requests are made than the number of responses stored, the responses will cycle again in order.

In the example above, the adapter is configured to test your object's behavior when it encounters a 302 redirect. Depending on your application, following a redirect may or may not be desired behavior. In our example, we expect that the redirect will be followed and we configure the test adapter to help us test this. The initial 302 response is set up with the `setResponse()` method and the 200 response to be returned next is added with the `addResponse()` method. After configuring the test adapter, inject the *HTTP* client containing the adapter into your object under test and test its behavior.

If you need the adapter to fail on demand you can use `setNextRequestWillFail($flag)`. The method will cause the next call to `connect()` to throw an `Zend\Http\Client\Adapter\Exception\RuntimeException` exception. This can be useful when our application caches content from an external site (in case the site goes down) and you want to test this feature.

Forcing the adapter to fail

```
// Instantiate a new adapter and client
$adapter = new Zend\Http\Client\Adapter\Test();
$client = new Zend\Http\Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Force the next request to fail with an exception
$adapter->setNextRequestWillFail(true);

try {
    // This call will result in a Zend\Http\Client\Adapter\Exception\RuntimeException
    $client->request();
} catch (Zend\Http\Client\Adapter\Exception\RuntimeException $e) {
    // ...
}

// Further requests will work as expected until
// you call setNextRequestWillFail(true) again
```

119.6 Creating your own connection adapters

`Zend\Http\Client` has been designed so that you can create and use your own connection adapters. You could, for example, create a connection adapter that uses persistent sockets, or a connection adapter with caching abilities, and use them as needed in your application.

In order to do so, you must create your own adapter class that implements the `Zend\Http\Client\Adapter\AdapterInterface` interface. The following example shows the skeleton of a user-implemented adapter class. All the public functions defined in this example must be defined in your adapter as well:

Creating your own connection adapter

```
class MyApp\Http\Client\Adapter\BananaProtocol
    implements Zend\Http\Client\Adapter\AdapterInterface
{
    /**
     * Set Adapter Options
     *
     * @param array $config
     */
    public function setOptions($config = array())
    {
        // This rarely changes - you should usually copy the
        // implementation in Zend\Http\Client\Adapter\Socket.
    }

    /**
     * Connect to the remote server
     *
     * @param string  $host
     * @param int     $port
     * @param boolean $secure
     */
    public function connect($host, $port = 80, $secure = false)
    {
        // Set up the connection to the remote server
    }

    /**
     * Send request to the remote server
     *
     * @param string        $method
     * @param Zend\Uri\Http $url
     * @param string        $http_ver
     * @param array         $headers
     * @param string        $body
     * @return string Request as text
     */
    public function write($method,
                          $url,
                          $http_ver = '1.1',
                          $headers = array(),
                          $body = '')
    {
        // Send request to the remote server.
```

```
        // This function is expected to return the full request
        // (headers and body) as a string
    }

    /**
     * Read response from server
     *
     * @return string
     */
    public function read()
    {
        // Read response from remote server and return it as a string
    }

    /**
     * Close the connection to the server
     *
     */
    public function close()
    {
        // Close the connection to the remote server - called last.
    }
}

// Then, you could use this adapter:
$client = new Zend\Http\Client(array(
    'adapter' => 'MyApp\Http\Client\Adapter\BananaProtocol'
));
```

CHAPTER

HTTP CLIENT - ADVANCED USAGE

120.1 HTTP Redirections

`Zend\Http\Client` automatically handles *HTTP* redirections, and by default will follow up to 5 redirections. This can be changed by setting the `maxredirects` configuration parameter.

According to the *HTTP*/1.1 RFC, *HTTP* 301 and 302 responses should be treated by the client by resending the same request to the specified location - using the same request method. However, most clients to not implement this and always use a `GET` request when redirecting. By default, `Zend\Http\Client` does the same - when redirecting on a 301 or 302 response, all `GET` and POST parameters are reset, and a `GET` request is sent to the new location. This behavior can be changed by setting the `strictredirects` configuration parameter to boolean `TRUE`:

Forcing RFC 2616 Strict Redirections on 301 and 302 Responses

```
// Strict Redirections
$client->setOptions(array('strictredirects' => true));

// Non-strict Redirections
$client->setOptions(array('strictredirects' => false));
```

You can always get the number of redirections done after sending a request using the `getRedirectionsCount()` method.

120.2 Adding Cookies and Using Cookie Persistence

`Zend\Http\Client` provides an easy interface for adding cookies to your request, so that no direct header modification is required. Cookies can be added using either the *addCookie()* or `setCookies` method. The `addCookie` method has a number of operating modes:

Setting Cookies Using addCookie()

```
// Easy and simple: by providing a cookie name and cookie value
$client->addCookie('flavor', 'chocolate chips');

// By directly providing a raw cookie string (name=value)
// Note that the value must be already URL encoded
$client->addCookie('flavor=chocolate%20chips');

// By providing a Zend\Http\Header\SetCookie object
$cookie = Zend\Http\Header\SetCookie::fromString('flavor=chocolate%20chips');
```

```
$client->addCookie($cookie);

// Multiple cookies can be set at once by providing an
// array of Zend\Http\Header\SetCookie objects
$cookies = array(
    Zend\Http\Header\SetCookie::fromString('flavorOne=chocolate%20chips'),
    Zend\Http\Header\SetCookie::fromString('flavorTwo=vanilla'),
);
$client->addCookie($cookies);
```

The `setCookies()` method works in a similar manner, except that it requires an array of cookie values as its only argument and also clears the cookie container before adding the new cookies:

Setting Cookies Using setCookies()

```
// setCookies accepts an array of cookie values, which
// can be in either of the following formats:
$client->setCookies(array(

    // A raw cookie string (name=value)
    // Note that the value must be already URL encoded
    'flavor=chocolate%20chips',

    // A Zend\Http\Header\SetCookie object
    Zend\Http\Header\SetCookie::fromString('flavor=chocolate%20chips'),

));
```

For more information about `Zend\Http\Header\SetCookie` objects, see *this section*.

`Zend\Http\Client` also provides a means for simplifying cookie stickiness - that is having the client internally store all sent and received cookies, and resend them on subsequent requests: `Zend\Http\Client\Cookies`. This is useful, for example when you need to log in to a remote site first and receive and authentication or session ID cookie before sending further requests.

Enabling Cookie Stickiness

```
$cookies = new Zend\Http\Cookies();

// First request: log in and start a session
$client->setUri('http://example.com/login.php');
$client->setParameterPost(array('user' => 'h4x0r', 'password' => 'l33t'));
$response = $client->request('POST');
$cookies->addCookiesFromResponse($response, $client->getUri());

// Now we can send our next request
$client->setUri('http://example.com/read_member_news.php');
$client->addCookies($cookies->getMatchingCookies($client->getUri());
$client->request('GET');
```

For more information about the `Zend\Http\Client\Cookies` class, see *this section*.

120.3 Setting Custom Request Headers

Setting custom headers is performed by first fetching the header container from the client's `Zend\Http\Request` object. This method is quite diverse and can be used in several ways, as the following example shows:

Setting A Single Custom Request Header

```
// Fetch the container
$headers = $client->getRequest()->getHeaders();

// Setting a single header. Will not overwrite any
// previously-added headers of the same name.
$headers->addHeaderLine('Host', 'www.example.com');

// Another way of doing the exact same thing
$headers->addHeaderLine('Host: www.example.com');

// Another way of doing the exact same thing using
// the provided Zend\Http\Header class
$headers->addHeader(Zend\Http\Header\Host::fromString('Host: www.example.com'));

// You can also add multiple headers at once by passing an
// array to addHeaders using any of the formats below:
$headers->addHeaders(array(
    // Zend\Http\Header\* object
    Zend\Http\Header\Host::fromString('Host: www.example.com'),

    // Header name as array key, header value as array key value
    'Cookie' => 'PHPSESSID=1234567890abcdef1234567890abcdef',

    // Raw header string
    'Cookie: language=he',
));
```

`Zend\Http\Client` also provides a convenience method for setting request headers, `setHeaders`. This method will create a new header container, add the specified headers and then store the new header container in it's `Zend\Http\Request` object. As a consequence, any pre-existing headers will be erased.

Setting Multiple Custom Request Headers

```
// Setting multiple headers.  Will remove all existing
// headers and add new ones to the Request header container
$client->setHeaders(array(
 Zend\Http\Header\Host::fromString('Host: www.example.com'),
 'Accept-encoding' => 'gzip,deflate',
 'X-Powered-By: Zend Framework'
));
```

120.4 File Uploads

You can upload files through *HTTP* using the setFileUpload method. This method takes a file name as the first parameter, a form name as the second parameter, and data as a third optional parameter. If the third data parameter is

NULL, the first file name parameter is considered to be a real file on disk, and Zend\Http\Client will try to read this file and upload it. If the data parameter is not NULL, the first file name parameter will be sent as the file name, but no actual file needs to exist on the disk. The second form name parameter is always required, and is equivalent to the "name" attribute of an <input> tag, if the file was to be uploaded through an *HTML* form. A fourth optional parameter provides the file's content-type. If not specified, and Zend\Http\Client reads the file from the disk, the mime_content_type function will be used to guess the file's content type, if it is available. In any case, the default MIME type will be application/octet-stream.

Using setFileUpload to Upload Files

```
// Uploading arbitrary data as a file
$text = 'this is some plain text';
$client->setFileUpload('some_text.txt', 'upload', $text, 'text/plain');

// Uploading an existing file
$client->setFileUpload('/tmp/Backup.tar.gz', 'bufile');

// Send the files
$client->setMethod('POST');
$client->send();
```

In the first example, the $text variable is uploaded and will be available as $_FILES['upload'] on the server side. In the second example, the existing file /tmp/Backup.tar.gz is uploaded to the server and will be available as $_FILES['bufile']. The content type will be guessed automatically if possible - and if not, the content type will be set to 'application/octet-stream'.

Note: Uploading files

When uploading files, the *HTTP* request content-type is automatically set to multipart/form-data. Keep in mind that you must send a POST or PUT request in order to upload files. Most servers will ignore the request body on other request methods.

120.5 Sending Raw POST Data

You can use a Zend\Http\Client to send raw POST data using the setRawBody() method. This method takes one parameter: the data to send in the request body. When sending raw POST data, it is advisable to also set the encoding type using setEncType().

Sending Raw POST Data

```
$xml = '<book>' .
       '  <title>Islands in the Stream</title>' .
       '  <author>Ernest Hemingway</author>' .
       '  <year>1970</year>' .
       '</book>';
$client->setMethod('POST');
$client->setRawBody($xml);
$client->setEncType('text/xml');
$client->send();
```

The data should be available on the server side through *PHP*'s $HTTP_RAW_POST_DATA variable or through the php://input stream.

Note: Using raw POST data

Setting raw POST data for a request will override any POST parameters or file uploads. You should not try to use both on the same request. Keep in mind that most servers will ignore the request body unless you send a POST request.

120.6 HTTP Authentication

Currently, `Zend\Http\Client` only supports basic *HTTP* authentication. This feature is utilized using the `setAuth()` method, or by specifying a username and a password in the URI. The `setAuth()` method takes 3 parameters: The user name, the password and an optional authentication type parameter. As mentioned, currently only basic authentication is supported (digest authentication support is planned).

Setting HTTP Authentication User and Password

```
// Using basic authentication
$client->setAuth('shahar', 'myPassword!', Zend\Http\Client::AUTH_BASIC);

// Since basic auth is default, you can just do this:
$client->setAuth('shahar', 'myPassword!');

// You can also specify username and password in the URI
$client->setUri('http://christer:secret@example.com');
```

120.7 Sending Multiple Requests With the Same Client

`Zend\Http\Client` was also designed specifically to handle several consecutive requests with the same object. This is useful in cases where a script requires data to be fetched from several places, or when accessing a specific *HTTP* resource requires logging in and obtaining a session cookie, for example.

When performing several requests to the same host, it is highly recommended to enable the 'keepalive' configuration flag. This way, if the server supports keep-alive connections, the connection to the server will only be closed once all requests are done and the Client object is destroyed. This prevents the overhead of opening and closing *TCP* connections to the server.

When you perform several requests with the same client, but want to make sure all the request-specific parameters are cleared, you should use the `resetParameters()` method. This ensures that GET and POST parameters, request body and headers are reset and are not reused in the next request.

Note: Resetting parameters

Note that cookies are not reset by default when the `resetParameters()` method is used. To clean all cookies as well, use `resetParameters(true)`, or call `clearCookies()` after calling `resetParameters()`.

Another feature designed specifically for consecutive requests is the `Zend\Http\Client\Cookies` object. This "Cookie Jar" allow you to save cookies set by the server in a request, and send them back on consecutive requests transparently. This allows, for example, going through an authentication request before sending the actual data-fetching request.

If your application requires one authentication request per user, and consecutive requests might be performed in more than one script in your application, it might be a good idea to store the Cookies object in the user's session. This way, you will only need to authenticate the user once every session.

Performing consecutive requests with one client

```
// First, instantiate the client
$client = new Zend\Http\Client('http://www.example.com/fetchdata.php', array(
    'keepalive' => true
));

// Do we have the cookies stored in our session?
if (isset($_SESSION['cookiejar']) &&
    $_SESSION['cookiejar'] instanceof Zend\Http\Client\Cookies) {

    $cookieJar = $_SESSION['cookiejar'];
} else {
    // If we don't, authenticate and store cookies
    $client->setUri('http://www.example.com/login.php');
    $client->setParameterPost(array(
        'user' => 'shahar',
        'pass' => 'somesecret'
    ));
    $response = $client->setMethod('POST')->send();
    $cookieJar = Zend\Http\Client\Cookies::fromResponse($response);

    // Now, clear parameters and set the URI to the original one
    // (note that the cookies that were set by the server are now
    // stored in the jar)
    $client->resetParameters();
    $client->setUri('http://www.example.com/fetchdata.php');
}

// Add the cookies to the new request
$client->setCookies($cookieJar->getMatchingCookies($client->getUri()));
$response = $client->setMethod('GET')->send();

// Store cookies in session, for next page
$_SESSION['cookiejar'] = $cookieJar;
```

120.8 Data Streaming

By default, `Zend\Http\Client` accepts and returns data as *PHP* strings. However, in many cases there are big files to be received, thus keeping them in memory might be unnecessary or too expensive. For these cases, `Zend\Http\Client` supports writing data to files (streams).

In order to receive data from the server as stream, use `setStream()`. Optional argument specifies the filename where the data will be stored. If the argument is just `TRUE` (default), temporary file will be used and will be deleted once response object is destroyed. Setting argument to `FALSE` disables the streaming functionality.

When using streaming, `send()` method will return object of class `Zend\Http\Response\Stream`, which has two useful methods: `getStreamName()` will return the name of the file where the response is stored, and `getStream()` will return stream from which the response could be read.

You can either write the response to pre-defined file, or use temporary file for storing it and send it out or write it to another file using regular stream functions.

Receiving file from HTTP server with streaming

```
$client->setStream(); // will use temp file
$response = $client->send();
// copy file
copy($response->getStreamName(), "my/downloads/file");
// use stream
$fp = fopen("my/downloads/file2", "w");
stream_copy_to_stream($response->getStream(), $fp);
// Also can write to known file
$client->setStream("my/downloads/myfile")->send();
```

CHAPTER

ONE

HTTP CLIENT - STATIC USAGE

121.1 Overview

The `Zend\Http` component also provides `Zend\Http\ClientStatic`, a static HTTP client which exposes a simplified API for quickly performing GET and POST operations:

121.2 Quick Start

```
use Zend\Http\ClientStatic;

// Simple GET request
$response = ClientStatic::get('http://example.org');

// More complex GET request, specifying query string 'foo=bar' and adding a
// custom header to request JSON data be returned (Accept: application/json)
$response = ClientStatic::get(
    'http://example.org',
    array( 'foo' => 'bar' ),
    array( 'Accept' => 'application/json')
);

// We can also do a POST request using the same format.  Here we POST
// login credentials (username/password) to a login page:
$response = ClientStatic::post('https://example.org/login.php', array(
    'username' => 'foo',
    'password' => 'bar',
));
```

121.3 Configuration Options

It is not possible to set configuration options on the `Zend\Http\Client` instance encapsulated by `Zend\Http\ClientStatic`. To perform a HTTP request which requires non-default configurations, please use `Zend\Http\Client` directly.

121.4 Available Methods

get `get(string $url, array $query = array(), array $headers = array(), mixed $body = null)`

Perform an HTTP `GET` request using the provided URL, query string variables, headers and request body.

Returns Zend\Http\Response

post `post(string $url, array $params, array $headers = array(), mixed $body = null)`

Perform an HTTP `POST` request using the provided URL, parameters, headers and request body.

Returns Zend\Http\Response

CHAPTER

TWO

TRANSLATING

ZendI18n comes with a complete translation suite which supports all major formats and includes popular features like plural translations and text domains. The Translator component is mostly dependency free, except for the fallback to a default locale, where it relies on the Intl PHP extension.

The translator itself is initialized without any parameters, as any configuration to it is optional. A translator without any translations will actually do nothing but just return the given message IDs.

122.1 Adding translations

To add translations to the translator, there are two options. You can either add every translation file individually, which is the best way if you use translation formats which store multiple locales in the same file, or you can add translations via a pattern, which works best for formats which contain one locale per file.

To add a single file to the translator, use the `addTranslationFile()` method:

```
use Zend\I18n\Translator\Translator;

$translator = new Translator();
$translator->addTranslationFile($type, $filename, $textDomain, $locale);
```

The type given there is a name of one of the format loaders listed in the next section. Filename points to the file containing the translations, and the text domain specifies a category name for the translations. If the text domain is omitted, it will default to the "default" value. The locale specifies which language the translated strings are from and is only required for formats which contain translations for a single locale.

Note: For each text domain and locale combination, there can only be one file loaded. Every successive file would override the translations which were loaded prior.

When storing one locale per file, you should specify those files via a pattern. This allows you to add new translations to the file system, without touching your code. Patterns are added with the `addTranslationFilePattern()` method:

```
use Zend\I18n\Translator\Translator;

$translator = new Translator();
$translator->addTranslationFilePattern($type, $pattern, $textDomain);
```

The parameters for adding patterns is pretty similar to adding individual files, except that you don't specify a locale and give the file location as a sprintf pattern. The locale is passed to the sprintf call, so you can either use %s or %1$s where it should be substituted. So when your translation files are located in /var/messages/LOCALE/messages.mo, you would specify your pattern as /var/messages/%s/messages.mo.

122.2 Supported formats

The translator supports the following major translation formats:

- PHP arrays
- Gettext
- Tmx
- Xliff

122.3 Setting a locale

By default, the translator will get the locale to use from the Intl extension's `Locale` class. If you want to set an alternative locale explicitly, you can do so by passing it to the `setLocale()` method.

When there is no translation for a specific message ID in a locale, the message ID itself will be returned by default. Alternatively you can set a fallback locale which is used to retrieve a fallback translation. To do so, pass it to the `setFallbackLocale()` method.

122.4 Translating messages

Translating messages can accomplished by calling the `translate()` method of the translator:

```
$translator->translate($message, $textDomain, $locale);
```

The message is the ID of your message to translate. If it does not exist in the loader translations or is empty, the original message ID will be returned. The text domain parameter is the one you specified when adding translations. If omitted, the default text domain will be used. The locale parameter will usually not be used in this context, as by default the locale is taken from the locale set in the translator.

To translate plural messages, you can use the `translatePlural()` method. It works similar to `translate()`, but instead of a single message it takes a singular and a plural value and an additional integer number on which the returned plural form is based on:

```
$translator->translatePlural($singular, $plural, $number, $textDomain, $locale);
```

Plural translations are only available if the underlying format supports the transport of plural messages and plural rule definitions.

122.5 Caching

In production it makes sense to cache your translations. This not only saves you from loading and parsing the individual formats each time, but also guarantees an optimized loading procedure. To enable caching, simply pass a `Zend\Cache\Storage\Adapter` to the `setCache()` method. To disable the cache, you can just pass a null value to it.

CHAPTER

THREE

I18N VIEW HELPERS

123.1 Introduction

Zend Framework comes with an initial set of helper classes related to Internationalization: e.g., formatting a date, formatting currency, or displaying translated content. You can use helper, or plugin, classes to perform these behaviors for you.

See the section on *view helpers* for more information.

123.2 CurrencyFormat Helper

The `CurrencyFormat` view helper can be used to simplify rendering of localized currency values. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl). **Basic Usage**

```
// Within your view

echo $this->currencyFormat(1234.56, "USD", "en_US");
// This returns: "$1,234.56"

echo $this->currencyFormat(1234.56, "EUR", "de_DE");
// This returns: "1.234,56 €"
```

currencyFormat (*float $number*, *string $currencyCode*[, *string $locale*])

Parameters

- **$number** – The numeric currency value.
- **$currencyCode** – The 3-letter ISO 4217 currency code indicating the currency to use.
- **$locale** – (Optional) Locale in which the currency would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

The `$currencyCode` and `$locale` options can be set prior to formatting and will be applied each time the helper is used:

```
// Within your view
$this->plugin("currencyformat")->setCurrencyCode("USD")->setLocale("en_US");

echo $this->currencyFormat(1234.56);  // "$1,234.56"
echo $this->currencyFormat(5678.90);  // "$5,678.90"
```

123.3 DateFormat Helper

The `DateFormat` view helper can be used to simplify rendering of localized date/time values. It acts as a wrapper for the `IntlDateFormatter` class within the Internationalization extension (Intl). **Basic Usage**

```
// Within your view

// Date and Time
echo $this->dateFormat(
    new DateTime(),
    IntlDateFormatter::MEDIUM, // date
    IntlDateFormatter::MEDIUM, // time
    "en_US"
);
// This returns: "Jul 2, 2012 6:44:03 PM"

// Date Only
echo $this->dateFormat(
    new DateTime(),
    IntlDateFormatter::LONG, // date
    IntlDateFormatter::NONE, // time
    "en_US"
);
// This returns: "July 2, 2012"

// Time Only
echo $this->dateFormat(
    new DateTime(),
    IntlDateFormatter::NONE,  // date
    IntlDateFormatter::SHORT, // time
    "en_US"
);
// This returns: "6:44 PM"
```

dateFormat(*mixed $date*[, *int $dateType*[, *int $timeType*[, *string $locale*]]])

Parameters

- **$date** – The value to format. This may be a `DateTime` object, an integer representing a Unix timestamp value or an array in the format output by `localtime()`.
- **$dateType** – (Optional) Date type to use (none, short, medium, long, full). This is one of the IntlDateFormatter constants. Defaults to `IntlDateFormatter::NONE`.
- **$timeType** – (Optional) Time type to use (none, short, medium, long, full). This is one of the IntlDateFormatter constants. Defaults to `IntlDateFormatter::NONE`.
- **$locale** – (Optional) Locale in which the date would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

The `$locale` option can be set prior to formatting with the `setLocale()` method and will be applied each time the helper is used.

By default, the system's default timezone will be used when formatting. This overrides any timezone that may be set inside a DateTime object. To change the timezone when formatting, use the `setTimezone` method.

```
// Within your view
$this->plugin("dateFormat")->setTimezone("America/New_York")->setLocale("en_US");

```

```
echo $this->dateFormat(new DateTime(), IntlDateFormatter::MEDIUM);  // "Jul 2, 2012"
echo $this->dateFormat(new DateTime(), IntlDateFormatter::SHORT);   // "7/2/12"
```

123.4 NumberFormat Helper

The `NumberFormat` view helper can be used to simplify rendering of locale-specific number and percentage strings. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl). **Basic Usage**

```
// Within your view

// Example of Decimal formatting:
echo $this->numberFormat(
    1234567.891234567890000,
    NumberFormatter::DECIMAL,
    NumberFormatter::TYPE_DEFAULT,
    "de_DE"
);
// This returns: "1.234.567,891"

// Example of Percent formatting:
echo $this->numberFormat(
    0.80,
    NumberFormatter::PERCENT,
    NumberFormatter::TYPE_DEFAULT,
    "en_US"
);
// This returns: "80%"

// Example of Scientific notation formatting:
echo $this->numberFormat(
    0.00123456789,
    NumberFormatter::SCIENTIFIC,
    NumberFormatter::TYPE_DEFAULT,
    "fr_FR"
);
// This returns: "1,23456789E-3"
```

numberFormat (*number $number*[, *int $formatStyle*[, *int $formatType*[, *string $locale*]]])

Parameters

- **$number** – The numeric value.
- **$formatStyle** – (Optional) Style of the formatting, one of the format style constants. If unset, it will use `NumberFormatter::DECIMAL` as the default style.
- **$formatType** – (Optional) The formatting type to use. If unset, it will use `NumberFormatter::TYPE_DEFAULT` as the default type.
- **$locale** – (Optional) Locale in which the number would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

The `$formatStyle`, `$formatType`, and `$locale` options can be set prior to formatting and will be applied each time the helper is used.

```
// Within your view
$this->plugin("numberformat")
            ->setFormatStyle(NumberFormatter::PERCENT)
            ->setFormatType(NumberFormatter::TYPE_DOUBLE)
            ->setLocale("en_US");

echo $this->numberFormat(0.56);  // "56%"
echo $this->numberFormat(0.90);  // "90%"
```

123.5 Plural Helper

Most languages have specific rules for handling plurals. For instance, in English, we say "0 cars" and "2 cars" (plural) while we say "1 car" (singular). On the other hand, French uses the singular form for 0 and 1 ("0 voiture" and "1 voiture") and uses the plural form otherwise ("3 voitures").

Therefore, we often need to handle those plural cases even without using translation (mono-lingual application). The `Plural` helper was created for this. Please remember that, if you need to both handle translation and plural, you must use the `TranslatePlural` helper for that. `Plural` does not deal with translation.

Internally, the `Plural` helper uses the `Zend\I18n\Translator\Plural\Rule` class to handle rules. **Setup**

In Zend Framework 1, there was a similar helper. However, this helper hardcoded rules for mostly every languages. The problem with this approach is that languages are alive and can evolve over time. Therefore, we would need to change the rules and hence break current applications that may (or may not) want those new rules.

That's why defining rules is now up to the developer. To help you with this process, here are some links with up-to-date plural rules for tons of languages:

- http://unicode.org/repos/cldr-tmp/trunk/diff/supplemental/language_plural_rules.html
- https://developer.mozilla.org/en-US/docs/Localization_and_Plurals

Basic Usage

The first thing to do is to defining rule. You may want to add this in your `Module.php` file, for example:

```
// Get the ViewHelperPlugin Manager from Service manager, so we can fetch the ``Plural``
// helper and add the plural rule for the application's language
$viewHelperManager = $serviceManager->get('ViewHelperManager');
$pluralHelper      = $viewHelperManager->get('Plural);

// Here is the rule for French
$pluralHelper->setPluralRule('nplurals=2; plural=(n==0 || n==1 ? 0 : 1)');
```

The string reads like that:

1. First, we specify how many plurals form do we have. For French, only two (singular/plural).
2. Then, we specify the rule. Here, if the count is 0 or 1, this is rule n°0 (singular) while it's rule n°1 otherwise.

As we said earlier, English consider "1" as singular, and "0/other" as plural. Here is such a rule:

```
// Here is the rule for English
$pluralHelper->setPluralRule('nplurals=2; plural=(n==1 ? 0 : 1)');
```

Now that we have defined the rule, we can use it in our views:

```php
<?php
   // If the rule defined in Module.php is the English one:

   echo $this->plural(array('car', 'cars'), 0); // prints "cars"
   echo $this->plural(array('car', 'cars'), 1); // prints "car"

   // If the rule defined in Module.php is the French one:
   echo $this->plural(array('voiture', 'voitures'), 0); // prints "voiture"
   echo $this->plural(array('voiture', 'voitures'), 1); // prints "voiture"
   echo $this->plural(array('voiture', 'voitures'), 2); // prints "voitures"
?>
```

123.6 Translate Helper

The `Translate` view helper can be used to translate content. It acts as a wrapper for the `Zend\I18n\Translator\Translator` class. **Setup**

Before using the `Translate` view helper, you must have first created a `Translator` object and have attached it to the view helper. If you use the `Zend\View\HelperPluginManager` to invoke the view helper, this will be done automatically for you. **Basic Usage**

```php
// Within your view

echo $this->translate("Some translated text.");

echo $this->translate("Translated text from a custom text domain.", "customDomain");

echo sprintf($this->translate("The current time is %s."), $currentTime);

echo $this->translate("Translate in a specific locale", "default", "de_DE");
```

translate(*string $message*[, *string $textDomain*[, *string $locale*]])

Parameters

- **$message** – The message to be translated.
- **$textDomain** – (Optional) The text domain where this translation lives. Defaults to the value "default".
- **$locale** – (Optional) Locale in which the message would be translated (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Gettext

The `xgettext` utility can be used to compile *.po files from PHP source files containing the translate view helper.

```
xgettext --language=php --add-location --keyword=translate my-view-file.phtml
```

See the Gettext Wikipedia page for more information. **Public Methods**

Public methods for setting a `Zend\I18n\Translator\Translator` and a default text domain are inherited from *Zend\I18n\View\Helper\AbstractTranslatorHelper*.

123.7 TranslatePlural Helper

The `TranslatePlural` view helper can be used to translate words which take into account numeric meanings. English, for example, has a singular definition of "car", for one car. And has the plural definition, "cars", meaning zero "cars" or more than one car. Other languages like Russian or Polish have more plurals with different rules.

The viewhelper acts as a wrapper for the `Zend\I18n\Translator\Translator` class. **Setup**

Before using the `TranslatePlural` view helper, you must have first created a `Translator` object and have attached it to the view helper. If you use the `Zend\View\HelperPluginManager` to invoke the view helper, this will be done automatically for you. **Basic Usage**

```
// Within your view
echo $this->translatePlural("car", "cars", $num);

// Use a custom domain
echo $this->translatePlural("monitor", "monitors", $num, "customDomain");

// Change locale
echo $this->translatePlural("locale", "locales", $num, "default", "de_DE");
```

translatePlural(*string $singular*, *string $plural*, *int $number*[, *string $textDomain*[, *string $locale*]])

Parameters

- **$singular** – The singular message to be translated.
- **$plural** – The plural message to be translated.
- **$number** – The number to evaluate and determine which message to use.
- **$textDomain** – (Optional) The text domain where this translation lives. Defaults to the value "default".
- **$locale** – (Optional) Locale in which the message would be translated (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

Public methods for setting a `Zend\I18n\Translator\Translator` and a default text domain are inherited from *Zend\I18n\View\Helper\AbstractTranslatorHelper*.

123.8 Abstract Translator Helper

The `AbstractTranslatorHelper` view helper is used as a base abstract class for any helpers that need to translate content. It provides an implementation for the `Zend\I18n\Translator\TranslatorAwareInterface` which allows injecting a translator and setting a text domain. **Public Methods**

setTranslator(*Translator $translator*[, *string $textDomain = null*])

Sets `Zend\I18n\Translator\Translator` to use in helper. The `$textDomain` argument is optional. It is provided as a convenience for setting both the translator and textDomain at the same time.

getTranslator()

Returns the `Zend\I18n\Translator\Translator` used in the helper.

Return type `Zend\I18n\Translator\Translator`

hasTranslator()

Returns true if a `Zend\I18n\Translator\Translator` is set in the helper, and false if otherwise.

Return type boolean

setTranslatorEnabled(*boolean $enabled*)
Sets whether translations should be enabled or disabled.

isTranslatorEnabled()
Returns true if translations are enabled, and false if disabled.

Return type boolean

setTranslatorTextDomain(*string $textDomain*)
Set the translation text domain to use in helper when translating.

getTranslatorTextDomain()
Returns the translation text domain used in the helper.

Return type string

CHAPTER

FOUR

I18N FILTERS

Zend Framework comes with a set of filters related to Internationalization.

124.1 Alnum

The `Alnum` filter can be used to return only alphabetic characters and digits in the unicode "letter" and "number" categories, respectively. All other characters are suppressed.

Supported Options for Alnum Filter

The following options are supported for `Alnum`:

`Alnum([ boolean $allowWhiteSpace [, string $locale ]])`

- `$allowWhiteSpace`: If set to true then whitespace characters are allowed. Otherwise they are suppressed. Default is "false" (whitespace is not allowed).

  Methods for getting/setting the allowWhiteSpace option are also available: `getAllowWhiteSpace()` and `setAllowWhiteSpace()`

- `$locale`: The locale string used in identifying the characters to filter (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`).

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

Alnum Filter Usage

```
// Default settings, deny whitespace
$filter = new \Zend\I18n\Filter\Alnum();
echo $filter->filter("This is (my) content: 123");
// Returns "Thisismycontent123"

// First param in constructor is $allowWhiteSpace
$filter = new \Zend\I18n\Filter\Alnum(true);
echo $filter->filter("This is (my) content: 123");
// Returns "This is my content 123"
```

Note: `Alnum` works on almost all languages, except: Chinese, Japanese and Korean. Within these languages the english alphabet is used instead of the characters from these languages. The language itself is detected using the `Locale`.

124.2 Alpha

The `Alpha` filter can be used to return only alphabetic characters in the unicode "letter" category. All other characters are suppressed.

Supported Options for Alpha Filter

The following options are supported for `Alpha`:

`Alpha([ boolean $allowWhiteSpace [, string $locale ]])`

- `$allowWhiteSpace`: If set to true then whitespace characters are allowed. Otherwise they are suppressed. Default is "false" (whitespace is not allowed).

  Methods for getting/setting the allowWhiteSpace option are also available: `getAllowWhiteSpace()` and `setAllowWhiteSpace()`

- `$locale`: The locale string used in identifying the characters to filter (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`).

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

Alpha Filter Usage

```
// Default settings, deny whitespace
$filter = new \Zend\I18n\Filter\Alpha();
echo $filter->filter("This is (my) content: 123");
// Returns "Thisismycontent"

// Allow whitespace
$filter = new \Zend\I18n\Filter\Alpha(true);
echo $filter->filter("This is (my) content: 123");
// Returns "This is my content "
```

Note: `Alpha` works on almost all languages, except: Chinese, Japanese and Korean. Within these languages the english alphabet is used instead of the characters from these languages. The language itself is detected using the `Locale`.

124.3 NumberFormat

The `NumberFormat` filter can be used to return locale-specific number and percentage strings. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl).

Supported Options for NumberFormat Filter

The following options are supported for `NumberFormat`:

`NumberFormat([ string $locale [, int $style [, int $type ]]])`

- `$locale`: (Optional) Locale in which the number would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

- `$style`: (Optional) Style of the formatting, one of the format style constants. If unset, it will use `NumberFormatter::DEFAULT_STYLE` as the default style.

  Methods for getting/setting the format style are also available: `getStyle()` and `setStyle()`

- `$type`: (Optional) The formatting type to use. If unset, it will use `NumberFormatter::TYPE_DOUBLE` as the default type.

  Methods for getting/setting the format type are also available: `getType()` and `setType()`

NumberFormat Filter Usage

```
$filter = new \Zend\I18n\Filter\NumberFormat("de_DE");
echo $filter->filter(1234567.8912346);
// Returns "1.234.567,891"

$filter = new \Zend\I18n\Filter\NumberFormat("en_US", NumberFormatter::PERCENT);
echo $filter->filter(0.80);
// Returns "80%"

$filter = new \Zend\I18n\Filter\NumberFormat("fr_FR", NumberFormatter::SCIENTIFIC);
echo $filter->filter(0.00123456789);
// Returns "1,23456789E-3"
```

CHAPTER

FIVE

I18N VALIDATORS

Zend Framework comes with a set of validators related to Internationalization.

CHAPTER

SIX

FLOAT

`Zend\I18n\Validator\Float` allows you to validate if a given value contains a floating-point value. This validator validates also localized input.

126.1 Supported options for Zend\I18n\Validator\Float

The following options are supported for `Zend\I18n\Validator\Float`:

- **locale**: Sets the locale which will be used to validate localized float values.

126.2 Simple float validation

The simplest way to validate a float is by using the system settings. When no option is used, the environment locale is used for validation:

```
$validator = new Zend\I18n\Validator\Float();

$validator->isValid(1234.5);   // returns true
$validator->isValid('10a01'); // returns false
$validator->isValid('1,234.5'); // returns true
```

In the above example we expected that our environment is set to "en" as locale.

126.3 Localized float validation

Often it's useful to be able to validate also localized values. Float values are often written different in other countries. For example using english you will write "1.5". In german you may write "1,5" and in other languages you may use grouping.

`Zend\I18n\Validator\Float` is able to validate such notations. However,it is limited to the locale you set. See the following code:

```
$validator = new Zend\I18n\Validator\Float(array('locale' => 'de'));

$validator->isValid(1234.5); // returns true
$validator->isValid("1 234,5"); // returns false
$validator->isValid("1.234"); // returns true
```

As you can see, by using a locale, your input is validated localized. Using a different notation you get a `FALSE` when the locale forces a different notation.

The locale can also be set afterwards by using `setLocale()` and retrieved by using `getLocale()`.

CHAPTER

SEVEN

INT

`Zend\I18n\Validator\Int` validates if a given value is an integer. Also localized integer values are recognised and can be validated.

127.1 Supported options for Zend\I18n\Validator\Int

The following options are supported for `Zend\I18n\Validator\Int`:

- **locale**: Sets the locale which will be used to validate localized integers.

127.2 Simple integer validation

The simplest way to validate an integer is by using the system settings. When no option is used, the environment locale is used for validation:

```
$validator = new Zend\I18n\Validator\Int();

$validator->isValid(1234);   // returns true
$validator->isValid(1234.5); // returns false
$validator->isValid('1,234'); // returns true
```

In the above example we expected that our environment is set to "en" as locale. As you can see in the third example also grouping is recognised.

127.3 Localized integer validation

Often it's useful to be able to validate also localized values. Integer values are often written different in other countries. For example using english you can write "1234" or "1,234". Both are integer values but the grouping is optional. In german for example you may write "1.234" and in french "1 234".

`Zend\I18n\Validator\Int` is able to validate such notations. But it is limited to the locale you set. This means that it not simply strips off the separator, it validates if the correct separator is used. See the following code:

```
$validator = new Zend\I18n\Validator\Int(array('locale' => 'de'));

$validator->isValid(1234); // returns true
$validator->isValid("1,234"); // returns false
$validator->isValid("1.234"); // returns true
```

As you can see, by using a locale, your input is validated localized. Using the english notation you get a `FALSE` when the locale forces a different notation.

The locale can also be set afterwards by using `setLocale()` and retrieved by using `getLocale()`.

CHAPTER

EIGHT

INTRODUCTION TO ZEND\INPUTFILTER

The `Zend\InputFilter` component can be used to filter and validate generic sets of input data. For instance, you could use it to filter `$_GET` or `$_POST` values, CLI arguments, etc.

To pass input data to the `InputFilter`, you can use the `setData()` method. The data must be specified using an associative array. Below is an example on how to validate the data coming from a form using the *POST* method.

```
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Input;
use Zend\Validator;

$email = new Input('email');
$email->getValidatorChain()
      ->addValidator(new Validator\EmailAddress());

$password = new Input('password');
$password->getValidatorChain()
         ->addValidator(new Validator\StringLength(8));

$inputFilter = new InputFilter();
$inputFilter->add($email)
            ->add($password)
            ->setData($_POST);

if ($inputFilter->isValid()) {
    echo "The form is valid\n";
} else {
    echo "The form is not valid\n";
    foreach ($inputFilter->getInvalidInput() as $error) {
        print_r ($error->getMessages());
    }
}
```

In this example we validated the email and password values. The email must be a valid address and the password must be composed with at least 8 characters. If the input data are not valid, we report the list of invalid input using the `getInvalidInput()` method.

You can add one or more validators to each input using the `addValidator()` method for each validator. It is also possible to specify a "validation group", a subset of the data to be validated; this may be done using the `setValidationGroup()` method. You can specify the list of the input names as an array or as individual parameters.

```
// As individual parameters
$filterInput->setValidationGroup('email', 'password');

// or as an array of names
$filterInput->setValidationGroup(array('email', 'password'));
```

You can validate and/or filter the data using the `InputFilter`. To filter data, use the `getFilterChain()` method of individual `Input` instances, and attach filters to the returned filter chain. Below is an example that uses filtering without validation.

```
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;

$input = new Input('foo');
$input->getFilterChain()
      ->attachByName('stringtrim')
      ->attachByName('alpha');

$inputFilter = new InputFilter();
$inputFilter->add($input)
            ->setData(array(
                'foo' => ' Bar3 ',
            ));

echo "Before:\n";
echo $inputFilter->getRawValue('foo') . "\n"; // the output is ' Bar3 '
echo "After:\n";
echo $inputFilter->getValue('foo') . "\n"; // the output is 'Bar'
```

The `getValue()` method returns the filtered value of the 'foo' input, while `getRawValue()` returns the original value of the input.

We provide also `Zend\InputFilter\Factory`, to allow initialization of the `InputFilter` based on a configuration array (or `Traversable` object). Below is an example where we create a password input value with the same constraints proposed before (a string with at least 8 characters):

```
use Zend\InputFilter\Factory;

$factory = new Factory();
$inputFilter = $factory->createInputFilter(array(
    'password' => array(
        'name'       => 'password',
        'required'   => true,
        'validators' => array(
            array(
                'name' => 'not_empty',
            ),
            array(
                'name' => 'string_length',
                'options' => array(
                    'min' => 8
                ),
            ),
        ),
    ),
));

$inputFilter->setData($_POST);
echo $inputFilter->isValid() ? "Valid form" : "Invalid form";
```

The factory may be used to create not only `Input` instances, but also nested `InputFilters`, allowing you to create validation and filtering rules for hierarchical data sets.

Finally, the default `InputFilter` implementation is backed by a `Factory`. This means that when calling `add()`, you can provide a specification that the `Factory` would understand, and it will create the appropriate object. You may create either `Input` or `InputFilter` objects in this fashion.

```
use Zend\InputFilter\InputFilter;

$filter = new InputFilter();

// Adding a single input
$filter->add(array(
    'name' => 'username',
    'required' => true,
    'validators' => array(
        array(
            'name' => 'not_empty',
        ),
        array(
            'name' => 'string_length',
            'options' => array(
                'min' => 5
            ),
        ),
    ),
));

// Adding another input filter what also contains a single input. Merging both.
$filter->add(array(
    'type' => 'Zend\InputFilter\InputFilter',
    'password' => array(
        'name' => 'password',
        'required' => true,
        'validators' => array(
            array(
                'name' => 'not_empty',
            ),
            array(
                'name' => 'string_length',
                'options' => array(
                    'min' => 8
                ),
            ),
        ),
    ),
));
```

CHAPTER

NINE

FILE UPLOAD INPUT

The `Zend\FileInput` class is a special `Input` type for uploaded files found in the `$_FILES` array.

While `FileInput` uses the same interface as `Input`, it differs in a few ways:

1. It expects the raw value to be in the `$_FILES` array format.
2. The validators are run **before** the filters (which is the opposite behavior of `Input`). This is so that any `is_uploaded_file()` validation can be run prior to any filters that may rename/move/modify the file.
3. Instead of adding a `NotEmpty` validator, it will (by default) automatically add a `Zend\Validator\File\UploadFile` validator.

The biggest thing to be concerned about is that if you are using a `<input type="file">` element in your form, you will need to use the `FileInput` **instead of** `Input` or else you will encounter issues.

Usage of `FileInput` is essentially the same as `Input`:

```
use Zend\Http\PhpEnvironment\Request;
use Zend\Filter;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Input;
use Zend\InputFilter\FileInput;
use Zend\Validator;

// Description text input
$description = new Input('description'); // Standard Input type
$description->getFilterChain()           // Filters are run first w/ Input
            ->attach(new Filter\StringTrim();
$description->getValidatorChain()        // Validators are run second w/ Input
            ->addValidator(new Validator\StringLength(array('max' => 140));

// File upload input
$file = new FileInput('file');           // Special File Input type
$file->getValidatorChain()               // Validators are run first w/ FileInput
     ->addValidator(new Validator\File\UploadFile());
$file->getFilterChain()                  // Filters are run second w/ FileInput
     ->attach(new Filter\File\RenameUpload(array(
         'target'    => './data/tmpuploads/file',
         'randomize' => true,
     )));

// Merge $_POST and $_FILES data together
$request  = new Request();
$postData = array_merge_recursive($request->getPost(), $request->getFiles());

$inputFilter = new InputFilter();
```

```
30 $inputFilter->add($description)
31             ->add($file)
32             ->setData($postData);
33
34 if ($inputFilter->isValid()) {           // FileInput validators are run, but not the filters...
35     echo "The form is valid\n";
36     $data = $inputFilter->getValues();   // This is when the FileInput filters are run.
37 } else {
38     echo "The form is not valid\n";
39     foreach ($inputFilter->getInvalidInput() as $error) {
40         print_r ($error->getMessages());
41     }
42 }
```

CHAPTER

INTRODUCTION TO ZEND\JSON

`Zend\Json` provides convenience methods for serializing native *PHP* to *JSON* and decoding *JSON* to native *PHP*. For more information on *JSON*, visit the JSON project site.

JSON, JavaScript Object Notation, can be used for data interchange between JavaScript and other languages. Since *JSON* can be directly evaluated by JavaScript, it is a more efficient and lightweight format than *XML* for exchanging data with JavaScript clients.

In addition, `Zend\Json` provides a useful way to convert any arbitrary *XML* formatted string into a *JSON* formatted string. This built-in feature will enable *PHP* developers to transform the enterprise data encoded in *XML* format into *JSON* format before sending it to browser-based Ajax client applications. It provides an easy way to do dynamic data conversion on the server-side code thereby avoiding unnecessary *XML* parsing in the browser-side applications. It offers a nice utility function that results in easier application-specific data processing techniques.

CHAPTER

ONE

BASIC USAGE

Usage of `Zend\Json` involves using the two public static methods available: `Zend\Json\Json::encode()` and `Zend\Json\Json::decode()`.

```
// Retrieve a value:
$phpNative = Zend\Json\Json::decode($encodedValue);

// Encode it to return to the client:
$json = Zend\Json\Json::encode($phpNative);
```

131.1 Pretty-printing JSON

Sometimes, it may be hard to explore *JSON* data generated by `Zend\Json\Json::encode()`, since it has no spacing or indentation. In order to make it easier, `Zend\Json\Json` allows you to pretty-print *JSON* data in the human-readable format with `Zend\Json\Json::prettyPrint()`.

```
// Encode it to return to the client:
$json = Zend\Json\Json::encode($phpNative);
if ($debug) {
    echo Zend\Json\Json::prettyPrint($json, array("indent" => " "));
}
```

Second optional argument of `Zend\Json\Json::prettyPrint()` is an option array. Option `indent` allows to set indentation string - by default it's a single tab character.

CHAPTER

TWO

ADVANCED USAGE OF ZEND\JSON

132.1 JSON Objects

When encoding *PHP* objects as *JSON*, all public properties of that object will be encoded in a *JSON* object.

JSON does not allow object references, so care should be taken not to encode objects with recursive references. If you have issues with recursion, `Zend\Json\Json::encode()` and `Zend\Json\Encoder::encode()` allow an optional second parameter to check for recursion; if an object is serialized twice, an exception will be thrown.

Decoding *JSON* objects poses an additional difficulty, however, since JavaScript objects correspond most closely to *PHP*'s associative array. Some suggest that a class identifier should be passed, and an object instance of that class should be created and populated with the key/value pairs of the *JSON* object; others feel this could pose a substantial security risk.

By default, `Zend\Json\Json` will decode *JSON* objects as associative arrays. However, if you desire an object returned, you can specify this:

```
// Decode JSON objects as PHP objects
$phpNative = Zend\Json\Json::decode($encodedValue, Zend\Json\Json::TYPE_OBJECT);
```

Any objects thus decoded are returned as `stdClass` objects with properties corresponding to the key/value pairs in the *JSON* notation.

The recommendation of Zend Framework is that the individual developer should decide how to decode *JSON* objects. If an object of a specified type should be created, it can be created in the developer code and populated with the values decoded using `Zend\Json`.

132.2 Encoding PHP objects

If you are encoding *PHP* objects by default the encoding mechanism can only access public properties of these objects. When a method `toJson()` is implemented on an object to encode, `Zend\Json\Json` calls this method and expects the object to return a *JSON* representation of its internal state.

`Zend\Json\Json` can encode PHP objects recursively but does not do so by default. This can be enabled by passing `true` as a second argument to `Zend\Json\Json::encode()`.

```
// Encode PHP object recursively
$jsonObject = Zend\Json\Json::encode($data, true);
```

When doing recursive encoding of objects, as JSON does not support cycles, an `Zend\Json\Exception\RecursionException` will be thrown. If you wish, you can silence these exceptions by passing the `silenceCyclicalExceptions` option:

```
$jsonObject = Zend\Json\Json::encode(
    $data,
    true,
    array('silenceCyclicalExceptions' => true)
);
```

132.3 Internal Encoder/Decoder

`Zend\Json` has two different modes depending if ext/json is enabled in your *PHP* installation or not. If ext/json is installed by default `json_encode()` and `json_decode()` functions are used for encoding and decoding *JSON*. If ext/json is not installed a Zend Framework implementation in *PHP* code is used for en-/decoding. This is considerably slower than using the *PHP* extension, but behaves exactly the same.

Still sometimes you might want to use the internal encoder/decoder even if you have ext/json installed. You can achieve this by calling:

```
Zend\Json\Json::$useBuiltinEncoderDecoder = true;
```

132.4 JSON Expressions

JavaScript makes heavy use of anonymous function callbacks, which can be saved within *JSON* object variables. Still they only work if not returned inside double quotes, which `Zend\Json` naturally does. With the Expression support for `Zend\Json` support you can encode *JSON* objects with valid JavaScript callbacks. This works for both `json_encode()` or the internal encoder.

A JavaScript callback is represented using the `Zend\Json\Expr` object. It implements the value object pattern and is immutable. You can set the JavaScript expression as the first constructor argument. By default `Zend\Json\Json::encode` does not encode JavaScript callbacks, you have to pass the option `enableJsonExprFinder` and set it to `TRUE` into the `encode` function. If enabled the expression support works for all nested expressions in large object structures. A usage example would look like:

```
$data = array(
    'onClick' => new Zend\Json\Expr('function() {'
               . 'alert("I am a valid JavaScript callback '
               . 'created by Zend\Json"); }'),
    'other' => 'no expression',
);
$jsonObjectWithExpression = Zend\Json\Json::encode(
    $data,
    false,
    array('enableJsonExprFinder' => true)
);
```

CHAPTER

THREE

XML TO JSON CONVERSION

`Zend\Json` provides a convenience method for transforming *XML* formatted data into *JSON* format. This feature was inspired from an IBM developerWorks article.

`Zend\Json` includes a static function called `Zend\Json\Json::fromXml()`. This function will generate *JSON* from a given *XML* input. This function takes any arbitrary *XML* string as an input parameter. It also takes an optional boolean input parameter to instruct the conversion logic to ignore or not ignore the *XML* attributes during the conversion process. If this optional input parameter is not given, then the default behavior is to ignore the *XML* attributes. This function call is made as shown below:

```
// fromXml function simply takes a String containing XML contents
// as input.
$jsonContents = Zend\Json\Json::fromXml($xmlStringContents, true);
```

`Zend\Json\Json::fromXml()` function does the conversion of the *XML* formatted string input parameter and returns the equivalent *JSON* formatted string output. In case of any *XML* input format error or conversion logic error, this function will throw an exception. The conversion logic also uses recursive techniques to traverse the *XML* tree. It supports recursion upto 25 levels deep. Beyond that depth, it will throw a `Zend\Json\Exception`. There are several *XML* files with varying degree of complexity provided in the tests directory of Zend Framework. They can be used to test the functionality of the xml2json feature.

The following is a simple example that shows both the *XML* input string passed to and the *JSON* output string returned as a result from the `Zend\Json\Json::fromXml()` function. This example used the optional function parameter as not to ignore the *XML* attributes during the conversion. Hence, you can notice that the resulting *JSON* string includes a representation of the *XML* attributes present in the *XML* input string.

XML input string passed to `Zend\Json\Json::fromXml()` function:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book id="1">
        <title>Code Generation in Action</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>Manning</publisher>
    </book>

    <book id="2">
        <title>PHP Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>

    <book id="3">
        <title>Podcasting Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
```

```
        <publisher>O'Reilly</publisher>
    </book>
</books>
```

JSON output string returned from `Zend\Json\Json::fromXml()` function:

```
{
   "books" : {
      "book" : [ {
         "@attributes" : {
            "id" : "1"
         },
         "title" : "Code Generation in Action",
         "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "Manning"
      }, {
         "@attributes" : {
            "id" : "2"
         },
         "title" : "PHP Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }, {
         "@attributes" : {
            "id" : "3"
         },
         "title" : "Podcasting Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }
   ]}
}
```

More details about this xml2json feature can be found in the original proposal itself. Take a look at the Zend_xml2json proposal.

CHAPTER

FOUR

ZEND\JSON\SERVER - JSON-RPC SERVER

`Zend\Json\Server` is a JSON-RPC server implementation. It supports both the JSON-RPC version 1 specification as well as the version 2 specification; additionally, it provides a *PHP* implementation of the Service Mapping Description (SMD) specification for providing service metadata to service consumers.

JSON-RPC is a lightweight Remote Procedure Call protocol that utilizes *JSON* for its messaging envelopes. This JSON-RPC implementation follows *PHP*'s SoapServer *API*. This means, in a typical situation, you will simply:

- Instantiate the server object
- Attach one or more functions and/or classes/objects to the server object
- handle() the request

`Zend\Json\Server` utilizes *Zend\Server\Reflection* to perform reflection on any attached classes or functions, and uses that information to build both the SMD and enforce method call signatures. As such, it is imperative that any attached functions and/or class methods have full *PHP* docblocks documenting, minimally:

- All parameters and their expected variable types
- The return value variable type

`Zend\Json\Server` listens for POST requests only at this time; fortunately, most JSON-RPC client implementations in the wild at the time of this writing will only POST requests as it is. This makes it simple to utilize the same server end point to both handle requests as well as to deliver the service SMD, as is shown in the next example.

Zend\Json\Server Usage

First, let's define a class we wish to expose via the JSON-RPC server. We'll call the class 'Calculator', and define methods for 'add', 'subtract', 'multiply', and 'divide':

```
/**
 * Calculator - sample class to expose via JSON-RPC
 */
class Calculator
{
    /**
     * Return sum of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
```

```
    public function add($x, $y)
    {
        return $x + $y;
    }

    /**
     * Return difference of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function subtract($x, $y)
    {
        return $x - $y;
    }

    /**
     * Return product of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function multiply($x, $y)
    {
        return $x * $y;
    }

    /**
     * Return the division of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return float
     */
    public function divide($x, $y)
    {
        return $x / $y;
    }
}
```

Note that each method has a docblock with entries indicating each parameter and its type, as well as an entry for the return value. This is **absolutely critical** when utilizing `Zend\Json\Server` or any other server component in Zend Framework, for that matter.

Now we'll create a script to handle the requests:

```
$server = new Zend\Json\Server\Server();

// Indicate what functionality is available:
$server->setClass('Calculator');

// Handle the request:
$server->handle();
```

However, this will not address the issue of returning an SMD so that the JSON-RPC client can autodiscover methods. That can be accomplished by determining the *HTTP* request method, and then specifying some server metadata:

```
$server = new Zend\Json\Server\Server();
$server->setClass('Calculator');

if ('GET' == $_SERVER['REQUEST_METHOD']) {
    // Indicate the URL endpoint, and the JSON-RPC version used:
    $server->setTarget('/json-rpc.php')
           ->setEnvelope(Zend\Json\Server\Smd::ENV_JSONRPC_2);

    // Grab the SMD
    $smd = $server->getServiceMap();

    // Return the SMD to the client
    header('Content-Type: application/json');
    echo $smd;
    return;
}

$server->handle();
```

If utilizing the JSON-RPC server with Dojo toolkit, you will also need to set a special compatibility flag to ensure that the two interoperate properly:

```
$server = new Zend\Json\Server\Server();
$server->setClass('Calculator');

if ('GET' == $_SERVER['REQUEST_METHOD']) {
    $server->setTarget('/json-rpc.php')
           ->setEnvelope(Zend\Json\Server\Smd::ENV_JSONRPC_2);
    $smd = $server->getServiceMap();

    // Set Dojo compatibility:
    $smd->setDojoCompatible(true);

    header('Content-Type: application/json');
    echo $smd;
    return;
}

$server->handle();
```

134.1 Advanced Details

While most functionality for `Zend\Json\Server` is spelled out in *this section*, more advanced functionality is available.

134.1.1 Zend\Json\Server\Server

`Zend\Json\Server\Server` is the core class in the JSON-RPC offering; it handles all requests and returns the response payload. It has the following methods:

- `addFunction($function)`: Specify a userland function to attach to the server.
- `setClass($class)`: Specify a class or object to attach to the server; all public methods of that item will be exposed as JSON-RPC methods.

- `fault($fault = null, $code = 404, $data = null)`: Create and return a `Zend\Json\Server\Error` object.
- `handle($request = false)`: Handle a JSON-RPC request; optionally, pass a `Zend\Json\Server\Request` object to utilize (creates one by default).
- `getFunctions()`: Return a list of all attached methods.
- `setRequest(Zend\Json\Server\Request $request)`: Specify a request object for the server to utilize.
- `getRequest()`: Retrieve the request object used by the server.
- `setResponse(Zend\Json\Server\Response $response)`: Set the response object for the server to utilize.
- `getResponse()`: Retrieve the response object used by the server.
- `setAutoEmitResponse($flag)`: Indicate whether the server should automatically emit the response and all headers; by default, this is `TRUE`.
- `autoEmitResponse()`: Determine if auto-emission of the response is enabled.
- `getServiceMap()`: Retrieve the service map description in the form of a `Zend\Json\Server\Smd` object

134.1.2 Zend\Json\Server\Request

The JSON-RPC request environment is encapsulated in the `Zend\Json\Server\Request` object. This object allows you to set necessary portions of the JSON-RPC request, including the request ID, parameters, and JSON-RPC specification version. It has the ability to load itself via *JSON* or a set of options, and can render itself as *JSON* via the `toJson()` method.

The request object has the following methods available:

- `setOptions(array $options)`: Specify object configuration. `$options` may contain keys matching any 'set' method: `setParams()`, `setMethod()`, `setId()`, and `setVersion()`.
- `addParam($value, $key = null)`: Add a parameter to use with the method call. Parameters can be just the values, or can optionally include the parameter name.
- `addParams(array $params)`: Add multiple parameters at once; proxies to `addParam()`
- `setParams(array $params)`: Set all parameters at once; overwrites any existing parameters.
- `getParam($index)`: Retrieve a parameter by position or name.
- `getParams()`: Retrieve all parameters at once.
- `setMethod($name)`: Set the method to call.
- `getMethod()`: Retrieve the method that will be called.
- `isMethodError()`: Determine whether or not the request is malformed and would result in an error.
- `setId($name)`: Set the request identifier (used by the client to match requests to responses).
- `getId()`: Retrieve the request identifier.
- `setVersion($version)`: Set the JSON-RPC specification version the request conforms to. May be either '1.0' or '2.0'.
- `getVersion()`: Retrieve the JSON-RPC specification version used by the request.
- `loadJson($json)`: Load the request object from a *JSON* string.

- `toJson()`: Render the request as a *JSON* string.

An *HTTP* specific version is available via `Zend\Json\Server\Request\Http`. This class will retrieve the request via `php://input`, and allows access to the raw *JSON* via the `getRawJson()` method.

134.1.3 Zend\Json\Server\Response

The JSON-RPC response payload is encapsulated in the `Zend\Json\Server\Response` object. This object allows you to set the return value of the request, whether or not the response is an error, the request identifier, the JSON-RPC specification version the response conforms to, and optionally the service map.

The response object has the following methods available:

- `setResult($value)`: Set the response result.
- `getResult()`: Retrieve the response result.
- `setError(Zend\Json\Server\Error $error)`: Set an error object. If set, this will be used as the response when serializing to *JSON*.
- `getError()`: Retrieve the error object, if any.
- `isError()`: Whether or not the response is an error response.
- `setId($name)`: Set the request identifier (so the client may match the response with the original request).
- `getId()`: Retrieve the request identifier.
- `setVersion($version)`: Set the JSON-RPC version the response conforms to.
- `getVersion()`: Retrieve the JSON-RPC version the response conforms to.
- `toJson()`: Serialize the response to *JSON*. If the response is an error response, serializes the error object.
- `setServiceMap($serviceMap)`: Set the service map object for the response.
- `getServiceMap()`: Retrieve the service map object, if any.

An *HTTP* specific version is available via `Zend\Json\Server\Response\Http`. This class will send the appropriate *HTTP* headers as well as serialize the response as *JSON*.

134.1.4 Zend\Json\Server\Error

JSON-RPC has a special format for reporting error conditions. All errors need to provide, minimally, an error message and error code; optionally, they can provide additional data, such as a backtrace.

Error codes are derived from those recommended by the XML-RPC EPI project. `Zend\Json\Server` appropriately assigns the code based on the error condition. For application exceptions, the code '-32000' is used.

`Zend\Json\Server\Error` exposes the following methods:

- `setCode($code)`: Set the error code; if the code is not in the accepted XML-RPC error code range, -32000 will be assigned.
- `getCode()`: Retrieve the current error code.
- `setMessage($message)`: Set the error message.
- `getMessage()`: Retrieve the current error message.
- `setData($data)`: Set auxiliary data further qualifying the error, such as a backtrace.
- `getData()`: Retrieve any current auxiliary error data.

- `toArray()`: Cast the error to an array. The array will contain the keys 'code', 'message', and 'data'.
- `toJson()`: Cast the error to a JSON-RPC error representation.

134.1.5 Zend\Json\Server\Smd

SMD stands for Service Mapping Description, a *JSON* schema that defines how a client can interact with a particular web service. At the time of this writing, the specification has not yet been formally ratified, but it is in use already within Dojo toolkit as well as other JSON-RPC consumer clients.

At its most basic, a Service Mapping Description indicates the method of transport (POST, `GET`, *TCP*/IP, etc), the request envelope type (usually based on the protocol of the server), the target *URL* of the service provider, and a map of services available. In the case of JSON-RPC, the service map is a list of available methods, which each method documenting the available parameters and their types, as well as the expected return value type.

`Zend\Json\Server\Smd` provides an object-oriented way to build service maps. At its most basic, you pass it metadata describing the service using mutators, and specify services (methods and functions).

The service descriptions themselves are typically instances of `Zend\Json\Server\Smd\Service`; you can also pass all information as an array to the various service mutators in `Zend\Json\Server\Smd`, and it will instantiate a service for you. The service objects contain information such as the name of the service (typically the function or method name), the parameters (names, types, and position), and the return value type. Optionally, each service can have its own target and envelope, though this functionality is rarely used.

`Zend\Json\Server\Server` actually does all of this behind the scenes for you, by using reflection on the attached classes and functions; you should create your own service maps only if you need to provide custom functionality that class and function introspection cannot offer.

Methods available in `Zend\Json\Server\Smd` include:

- `setOptions(array $options)`: Setup an SMD object from an array of options. All mutators (methods beginning with 'set') can be used as keys.
- `setTransport($transport)`: Set the transport used to access the service; only POST is currently supported.
- `getTransport()`: Get the current service transport.
- `setEnvelope($envelopeType)`: Set the request envelope that should be used to access the service. Currently, supports the constants `Zend\Json\Server\Smd::ENV_JSONRPC_1` and `Zend\Json\Server\Smd::ENV_JSONRPC_2`.
- `getEnvelope()`: Get the current request envelope.
- `setContentType($type)`: Set the content type requests should use (by default, this is 'application/json').
- `getContentType()`: Get the current content type for requests to the service.
- `setTarget($target)`: Set the *URL* endpoint for the service.
- `getTarget()`: Get the *URL* endpoint for the service.
- `setId($id)`: Typically, this is the *URL* endpoint of the service (same as the target).
- `getId()`: Retrieve the service ID (typically the *URL* endpoint of the service).
- `setDescription($description)`: Set a service description (typically narrative information describing the purpose of the service).
- `getDescription()`: Get the service description.

- `setDojoCompatible($flag)`: Set a flag indicating whether or not the SMD is compatible with Dojo toolkit. When `TRUE`, the generated *JSON* SMD will be formatted to comply with the format that Dojo's JSON-RPC client expects.
- `isDojoCompatible()`: Returns the value of the Dojo compatibility flag (`FALSE`, by default).
- `addService($service)`: Add a service to the map. May be an array of information to pass to the constructor of `Zend\Json\Server\Smd\Service`, or an instance of that class.
- `addServices(array $services)`: Add multiple services at once.
- `setServices(array $services)`: Add multiple services at once, overwriting any previously set services.
- `getService($name)`: Get a service by its name.
- `getServices()`: Get all attached services.
- `removeService($name)`: Remove a service from the map.
- `toArray()`: Cast the service map to an array.
- `toDojoArray()`: Cast the service map to an array compatible with Dojo Toolkit.
- `toJson()`: Cast the service map to a *JSON* representation.

`Zend\Json\Server\Smd\Service` has the following methods:

- `setOptions(array $options)`: Set object state from an array. Any mutator (methods beginning with 'set') may be used as a key and set via this method.
- `setName($name)`: Set the service name (typically, the function or method name).
- `getName()`: Retrieve the service name.
- `setTransport($transport)`: Set the service transport (currently, only transports supported by `Zend\Json\Server\Smd` are allowed).
- `getTransport()`: Retrieve the current transport.
- `setTarget($target)`: Set the *URL* endpoint of the service (typically, this will be the same as the overall SMD to which the service is attached).
- `getTarget()`: Get the *URL* endpoint of the service.
- `setEnvelope($envelopeType)`: Set the service envelope (currently, only envelopes supported by `Zend\Json\Server\Smd` are allowed).
- `getEnvelope()`: Retrieve the service envelope type.
- `addParam($type, array $options = array(), $order = null)`: Add a parameter to the service. By default, only the parameter type is necessary. However, you may also specify the order, as well as options such as:
  - **name**: the parameter name
  - **optional**: whether or not the parameter is optional
  - **default**: a default value for the parameter
  - **description**: text describing the parameter
- `addParams(array $params)`: Add several parameters at once; each param should be an assoc array containing minimally the key 'type' describing the parameter type, and optionally the key 'order'; any other keys will be passed as `$options` to `addOption()`.
- `setParams(array $params)`: Set many parameters at once, overwriting any existing parameters.

- `getParams()`: Retrieve all currently set parameters.
- `setReturn($type)`: Set the return value type of the service.
- `getReturn()`: Get the return value type of the service.
- `toArray()`: Cast the service to an array.
- `toJson()`: Cast the service to a *JSON* representation.

CHAPTER

FIVE

INTRODUCTION TO ZEND\LDAP

`Zend\Ldap\Ldap` is a class for performing *LDAP* operations including but not limited to binding, searching and modifying entries in an *LDAP* directory.

135.1 Theory of operation

This component currently consists of the main `Zend\Ldap\Ldap` class, that conceptually represents a binding to a single *LDAP* server and allows for executing operations against a *LDAP* server such as OpenLDAP or ActiveDirectory (AD) servers. The parameters for binding may be provided explicitly or in the form of an options array. `Zend\Ldap\Node` provides an object-oriented interface for single *LDAP* nodes and can be used to form a basis for an active-record-like interface for a *LDAP*-based domain model.

The component provides several helper classes to perform operations on *LDAP* entries (`Zend\Ldap\Attribute`) such as setting and retrieving attributes (date values, passwords, boolean values, ...), to create and modify *LDAP* filter strings (`Zend\Ldap\Filter`) and to manipulate *LDAP* distinguished names (DN) (`Zend\Ldap\Dn`).

Additionally the component abstracts *LDAP* schema browsing for OpenLDAP and ActiveDirectory servers `Zend\Ldap\Node\Schema` and server information retrieval for OpenLDAP-, ActiveDirectory- and Novell eDirectory servers (`Zend\Ldap\Node\RootDse`).

Using the `Zend\Ldap\Ldap` class depends on the type of *LDAP* server and is best summarized with some simple examples.

If you are using OpenLDAP, a simple example looks like the following (note that the **bindRequiresDn** option is important if you are **not** using AD):

```
$options = array(
    'host'               => 's0.foo.net',
    'username'           => 'CN=user1,DC=foo,DC=net',
    'password'           => 'pass1',
    'bindRequiresDn'     => true,
    'accountDomainName' => 'foo.net',
    'baseDn'             => 'OU=Sales,DC=foo,DC=net',
);
$ldap = new Zend\Ldap\Ldap($options);
$acctname = $ldap->getCanonicalAccountName('abaker',
                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

If you are using Microsoft AD a simple example is:

```
$options = array(
    'host'                    => 'dc1.w.net',
```

```
    'useStartTls'            => true,
    'username'               => 'user1@w.net',
    'password'               => 'pass1',
    'accountDomainName'      => 'w.net',
    'accountDomainNameShort' => 'W',
    'baseDn'                 => 'CN=Users,DC=w,DC=net',
);
$ldap = new Zend\Ldap\Ldap($options);
$acctname = $ldap->getCanonicalAccountName('bcarter',
                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

Note that we use the `getCanonicalAccountName()` method to retrieve the account DN here only because that is what exercises the most of what little code is currently present in this class.

135.1.1 Automatic Username Canonicalization When Binding

If `bind()` is called with a non-DN username but **bindRequiresDN** is `TRUE` and no username in DN form was supplied as an option, the bind will fail. However, if a username in DN form is supplied in the options array, `Zend\Ldap\Ldap` will first bind with that username, retrieve the account DN for the username supplied to `bind()` and then re-bind with that DN.

This behavior is critical to *Zend\Authentication\Adapter\Ldap*, which passes the username supplied by the user directly to `bind()`.

The following example illustrates how the non-DN username '**abaker**' can be used with `bind()`:

```
$options = array(
        'host'              => 's0.foo.net',
        'username'          => 'CN=user1,DC=foo,DC=net',
        'password'          => 'pass1',
        'bindRequiresDn'    => true,
        'accountDomainName' => 'foo.net',
        'baseDn'            => 'OU=Sales,DC=foo,DC=net',
);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind('abaker', 'moonbike55');
$acctname = $ldap->getCanonicalAccountName('abaker',
                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

The `bind()` call in this example sees that the username '**abaker**' is not in DN form, finds **bindRequiresDn** is `TRUE`, uses '`CN=user1,DC=foo,DC=net`' and '**pass1**' to bind, retrieves the DN for '**abaker**', unbinds and then rebinds with the newly discovered '`CN=Alice Baker,OU=Sales,DC=foo,DC=net`'.

135.1.2 Account Name Canonicalization

The **accountDomainName** and **accountDomainNameShort** options are used for two purposes: (1) they facilitate multi-domain authentication and failover capability, and (2) they are also used to canonicalize usernames. Specifically, names are canonicalized to the form specified by the **accountCanonicalForm** option. This option may one of the following values:

Table 135.1: Options for accountCanonicalForm

Name	Value	Example
ACCTNAME_FORM_DN	1	CN=Alice Baker,CN=Users,DC=example,DC=com
ACCTNAME_FORM_USERNAME	2	abaker
ACCTNAME_FORM_BACKSLASH	3	EXAMPLE\abaker
ACCTNAME_FORM_PRINCIPAL	4	abaker@example.com

The default canonicalization depends on what account domain name options were supplied. If **accountDomainNameShort** was supplied, the default **accountCanonicalForm** value is `ACCTNAME_FORM_BACKSLASH`. Otherwise, if **accountDomainName** was supplied, the default is `ACCTNAME_FORM_PRINCIPAL`.

Account name canonicalization ensures that the string used to identify an account is consistent regardless of what was supplied to `bind()`. For example, if the user supplies an account name of `abaker@example.com` or just **abaker** and the **accountCanonicalForm** is set to 3, the resulting canonicalized name would be **EXAMPLEabaker**.

135.1.3 Multi-domain Authentication and Failover

The `Zend\Ldap\Ldap` component by itself makes no attempt to authenticate with multiple servers. However, `Zend\Ldap\Ldap` is specifically designed to handle this scenario gracefully. The required technique is to simply iterate over an array of arrays of serve options and attempt to bind with each server. As described above `bind()` will automatically canonicalize each name, so it does not matter if the user passes `abaker@foo.net` or **Wbcarter** or **cdavis**- the `bind()` method will only succeed if the credentials were successfully used in the bind.

Consider the following example that illustrates the technique required to implement multi-domain authentication and failover:

```
$acctname = 'W\\user2';
$password = 'pass2';

$multiOptions = array(
    'server1' => array(
        'host'                   => 's0.foo.net',
        'username'               => 'CN=user1,DC=foo,DC=net',
        'password'               => 'pass1',
        'bindRequiresDn'         => true,
        'accountDomainName'      => 'foo.net',
        'accountDomainNameShort' => 'FOO',
        'accountCanonicalForm'   => 4, // ACCT_FORM_PRINCIPAL
        'baseDn'                 => 'OU=Sales,DC=foo,DC=net',
    ),
    'server2' => array(
        'host'                   => 'dc1.w.net',
        'useSsl'                 => true,
        'username'               => 'user1@w.net',
        'password'               => 'pass1',
        'accountDomainName'      => 'w.net',
        'accountDomainNameShort' => 'W',
        'accountCanonicalForm'   => 4, // ACCT_FORM_PRINCIPAL
        'baseDn'                 => 'CN=Users,DC=w,DC=net',
    ),
);

$ldap = new Zend\Ldap\Ldap();

foreach ($multiOptions as $name => $options) {

```

```
    echo "Trying to bind using server options for '$name'\n";

    $ldap->setOptions($options);
    try {
        $ldap->bind($acctname, $password);
        $acctname = $ldap->getCanonicalAccountName($acctname);
        echo "SUCCESS: authenticated $acctname\n";
        return;
    } catch (Zend\Ldap\Exception\LdapException $zle) {
        echo '  ' . $zle->getMessage() . "\n";
        if ($zle->getCode() === Zend\Ldap\Exception\LdapException::LDAP_X_DOMAIN_MISMATCH) {
            continue;
        }
    }
}
```

If the bind fails for any reason, the next set of server options is tried.

The `getCanonicalAccountName()` call gets the canonical account name that the application would presumably use to associate data with such as preferences. The **accountCanonicalForm = 4** in all server options ensures that the canonical form is consistent regardless of which server was ultimately used.

The special `LDAP_X_DOMAIN_MISMATCH` exception occurs when an account name with a domain component was supplied (e.g., `abaker@foo.net` or **FOOabaker** and not just **abaker**) but the domain component did not match either domain in the currently selected server options. This exception indicates that the server is not an authority for the account. In this case, the bind will not be performed, thereby eliminating unnecessary communication with the server. Note that the **continue** instruction has no effect in this example, but in practice for error handling and debugging purposes, you will probably want to check for `LDAP_X_DOMAIN_MISMATCH` as well as `LDAP_NO_SUCH_OBJECT` and `LDAP_INVALID_CREDENTIALS`.

The above code is very similar to code used within *Zend\Authentication\Adapter\Ldap*. In fact, we recommend that you simply use that authentication adapter for multi-domain + failover *LDAP* based authentication (or copy the code).

CHAPTER

SIX

API OVERVIEW

136.1 Configuration / options

The `Zend\Ldap\Ldap` component accepts an array of options either supplied to the constructor or through the `setOptions()` method. The permitted options are as follows:

Table 136.1: Zend\Ldap\Ldap Options

Name	Description
host	The default hostname of LDAP server if not supplied to connect() (also may be used when trying to canonicalize usernames in bind()).
port	Default port of LDAP server if not supplied to connect().
useStartTls	Whether or not the LDAP client should use TLS (aka SSLv2) encrypted transport. A value of TRUE is strongly favored in production environments to prevent passwords from be transmitted in clear text. The default value is FALSE, as servers frequently require that a certificate be installed separately after installation. The useSsl and useStartTls options are mutually exclusive. The useStartTls option should be favored over useSsl but not all servers support this newer mechanism.
useSsl	Whether or not the LDAP client should use SSL encrypted transport. The useSsl and useStartTls options are mutually exclusive.
username	The default credentials username. Some servers require that this be in DN form. This must be given in DN form if the LDAP server requires a DN to bind and binding should be possible with simple usernames.
password	The default credentials password (used only with username above).
bindRequiresDn	If TRUE, this instructs Zend\Ldap\Ldap to retrieve the DN for the account used to bind if the username is not already in DN form. The default value is FALSE.
baseDn	The default base DN used for searching (e.g., for accounts). This option is required for most account related operations and should indicate the DN under which accounts are located.
accountCanonicalForm	A small integer indicating the form to which account names should be canonicalized. See the Account Name Canonicalization section below.
accountDomainName	The FQDN domain for which the target LDAP server is an authority (e.g., example.com).
accountDomainNameShort	The 'short' domain for which the target LDAP server is an authority. This is usually used to specify the NetBIOS domain name for Windows networks but may also be used by non-AD servers.
accountFilterFormat	The LDAP search filter used to search for accounts. This string is a sprintf() style expression that must contain one '%s' to accommodate the username. The default value is '(&(objectClass=user)(sAMAccountName=%s))' unless bindRequiresDn is set to TRUE, in which case the default is '(&(objectClass=posixAccount)(uid=%s))'. Users of custom schemas may need to change this option.
allowEmptyPassword	Some LDAP servers can be configured to accept an empty string password as an anonymous bind. This behavior is almost always undesirable. For this reason, empty passwords are explicitly disallowed. Set this value to TRUE to allow an empty string password to be submitted during the bind.
optReferrals	If set to TRUE, this option indicates to the LDAP client that referrals should be followed. The default value is FALSE.
tryUsernameSplit	If set to FALSE, this option indicates that the given username should not be split at the first @ or \ character to separate the username from the domain during the binding-procedure. This allows the user to use usernames that contain an @ or \ character that do not inherit some domain-information, e.g. using email-addresses for binding. The default value is TRUE.
networkTimeout	Number of seconds to wait for LDAP connection before fail. If not set the default value is the system value.

136.2 API Reference

Note: Method names in **italics** are static methods.

CHAPTER

SEVEN

ZEND\LDAP\LDAP

`Zend\Ldap\Ldap` is the base interface into a *LDAP* server. It provides connection and binding methods as well as methods to operate on the *LDAP* tree.

Method
__construct($options)
resource getResource()
integer getLastErrorCode()
string getLastError(integer &$errorCode, array &$errorMessages)
Zend\Ldap\Ldap setOptions($options)
array getOptions()
string getBaseDn()
string getCanonicalAccountName(string $acctname, integer $form)
Zend\Ldap\Ldap disconnect()
Zend\Ldap\Ldap connect(string $host, integer $port, boolean $useSsl, boolean $useStartTls, integer $networkTimeout)
Zend\Ldap\Ldap bind(string $username, string $password)
Zend\Ldap\Collection search(string|Zend\Ldap\Filter\AbstractFilter $filter, string|Zend\Ldap\Dn $basedn, integer $scope, array $attrib
integer count(string|Zend\Ldap\Filter\AbstractFilter $filter, string|Zend\Ldap\Dn $basedn, integer $scope)
integer countChildren(string|Zend\Ldap\Dn $dn)
boolean exists(string|Zend\Ldap\Dn $dn)
array searchEntries(string|Zend\Ldap\Filter\AbstractFilter $filter, string|Zend\Ldap\Dn $basedn, integer $scope, array $attributes, strin
array getEntry(string|Zend\Ldap\Dn $dn, array $attributes, boolean $throwOnNotFound)
void prepareLdapEntryArray(array &$entry)
Zend\Ldap\Ldap add(string|Zend\Ldap\Dn $dn, array $entry)
Zend\Ldap\Ldap update(string|Zend\Ldap\Dn $dn, array $entry)
Zend\Ldap\Ldap save(string|Zend\Ldap\Dn $dn, array $entry)
Zend\Ldap\Ldap delete(string|Zend\Ldap\Dn $dn, boolean $recursively)
Zend\Ldap\Ldap moveToSubtree(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively, boolean $alwaysEmula
Zend\Ldap\Ldap move(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively, boolean $alwaysEmulate)
Zend\Ldap\Ldap rename(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively, boolean $alwaysEmulate)
Zend\Ldap\Ldap copyToSubtree(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively)
Zend\Ldap\Ldap copy(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively)
Zend\Ldap\Node getNode(string|Zend\Ldap\Dn $dn)
Zend\Ldap\Node getBaseNode()
Zend\Ldap\Node\RootDse getRootDse()
Zend\Ldap\Node\Schema getSchema()

137.1 Zend\Ldap\Collection

`Zend\Ldap\Collection` implements *Iterator* to allow for item traversal using `foreach()` and *Countable* to be able to respond to `count()`. With its protected `createEntry()` method it provides a simple extension point for developers needing custom result objects.

Table 137.2: Zend\Ldap\Collection API

Method	Description
__construct(Zend\Ldap\Collection\Iterator\Interface $iterator)	Constructor. The constructor must be provided by a Zend\Ldap\Collection\Iterator\Interface which does the real result iteration. Zend\Ldap\Collection\Iterator\Default is the default implementation for iterating ext/ldap results.
boolean close()	Closes the internal iterator. This is also called in the destructor.
array toArray()	Returns all entries as an array.
array getFirst()	Returns the first entry in the collection or NULL if the collection is empty.

CHAPTER

EIGHT

ZEND\LDAP\ATTRIBUTE

`Zend\Ldap\Attribute` is a helper class providing only static methods to manipulate arrays suitable to the structure used in `Zend\Ldap\Ldap` data modification methods and to the data format required by the *LDAP* server. *PHP* data types are converted using `Zend\Ldap\Converter\Converter` methods.

Table 138.1: Zend\Ldap\Attribute API

Method	Description
void setAttribute(array &$data, string $attribName, mixed $value, boolean $append)	Sets the attribute $attribName in $data to the value $value. If $append is TRUE (FALSE by default) $value will be appended to the attribute. $value can be a scalar value or an array of scalar values. Conversion will take place.
array|mixed getAttribute(array $data, string $attribName, integer|null $index)	Returns the attribute $attribName from $data. If $index is NULL (default) an array will be returned containing all the values for the given attribute. An empty array will be returned if the attribute does not exist in the given array. If an integer index is specified the corresponding value at the given index will be returned. If the index is out of bounds, NULL will be returned. Conversion will take place.
boolean attributeHasValue(array &$data, string $attribName, mixed|array $value)	Checks if the attribute $attribName in $data has the value(s) given in $value. The method returns TRUE only if all values in $value are present in the attribute. Comparison is done strictly (respecting the data type).
void removeDuplicatesFromAttribute(array &$data, string $attribName)	Removes all duplicates from the attribute $attribName in $data.
void removeFromAttribute(array &$data, string $attribName, mixed|array $value)	Removes the value(s) given in $value from the attribute $attribName in $data.
void setPassword(array &$data, string $password, string $hashType, string $attribName)	Sets a LDAP password for the attribute $attribName in $data. $attribName defaults to 'userPassword' which is the standard password attribute. The password hash can be specified with $hashType. The default value here is Zend\Ldap\Attribute::PASSWORD_HASH_MD5 with Zend\Ldap\Attribute::PASSWORD_HASH_SHA as the other possibility.
string createPassword(string $password, string $hashType)	Creates a LDAP password. The password hash can be specified with $hashType. The default value here is Zend\Ldap\Attribute::PASSWORD_HASH_MD5 with Zend\Ldap\Attribute::PASSWORD_HASH_SHA as the other possibility.
void setDateTimeAttribute(array &$data, string $attribName, integer|array $value, boolean $utc, boolean $append)	Sets the attribute $attribName in $data to the date/time value $value. if $append is TRUE (FALSE by default) $value will be appended to the attribute. $value can be an integer value or an array of integers. Date-time-conversion according to Zend\Ldap\Converter\Converter::toLdapDateTime() will take place.
array|integer getDateTimeAttribute(array $data, string $attribName, integer|null $index)	Returns the date/time attribute $attribName from $data. If $index is NULL (default) an array will be returned containing all the date/time values for the given attribute. An empty array will be returned if the attribute does not exist in the given array. If an integer index is specified the corresponding date/time value at the given index will be returned. If the index is out of bounds, NULL will be returned. Date-time-conversion according to Zend\Ldap\Converter\Converter::fromLdapDateTime() will take place.

CHAPTER

NINE

ZEND\LDAP\CONVERTER\CONVERTER

`Zend\Ldap\Converter\Converter` is a helper class providing only static methods to manipulate arrays suitable to the data format required by the *LDAP* server. *PHP* data types are converted the following way:

string No conversion will be done.

integer and float The value will be converted to a string.

boolean `TRUE` will be converted to **'TRUE'** and `FALSE` to **'FALSE'**

object and array The value will be converted to a string by using `serialize()`.

Date/Time The value will be converted to a string with the following `date()` format *YmdHisO*, UTC timezone (+0000) will be replaced with a *Z*. For example *01-30-2011 01:17:32 PM GMT-6* will be *20113001131732-0600* and *30-01-2012 15:17:32 UTC* will be *20120130151732Z*

resource If a *stream* resource is given, the data will be fetched by calling `stream_get_contents()`.

others All other data types (namely non-stream resources) will be omitted.

On reading values the following conversion will take place:

'TRUE' Converted to `TRUE`.

'FALSE' Converted to `FALSE`.

others All other strings won't be automatically converted and are passed as they are.

Table 139.1: Zend\Ldap\Converter\Converter API

Method	Description
string ascToHex32(string $string)	Convert all Ascii characters with decimal value less than 32 to hexadecimal value.
string hex32ToAsc(string $string)	Convert all hexadecimal characters by his Ascii value.
string|null toLdap(mixed $value, int $type)	Converts a PHP data type into its LDAP representation. $type argument is used to set the conversion method by default Converter::STANDARD where the function will try to guess the conversion method to use, others possibilities are Converter::BOOLEAN and Converter::GENERALIZED_TIME See introduction for details.
mixed fromLdap(string $value, int $type, boolean $dateTimeAsUtc)	Converts an LDAP value into its PHP data type. See introduction and toLdap() and toLdapDateTime() for details.
string|null toLdapDate-Time(integer|string|DateTime $date, boolean $asUtc)	Converts a timestamp, a DateTime Object, a string that is parseable by strtotime() or a DateTime into its LDAP date/time representation. If $asUtc is TRUE (FALSE by default) the resulting LDAP date/time string will be inUTC, otherwise a local date/time string will be returned.
DateTime fromLdapDateTime(string $date, boolean $asUtc)	Converts LDAP date/time representation into a PHP DateTime object.
string toLdap-Boolean(boolean|integer|string $value)	Converts a PHP data type into its LDAP boolean representation. By default always return 'FALSE' except if the value is true , 'true' or 1
boolean fromLdapBoolean(string $value)	Converts LDAP boolean representation into a PHP boolean data type.
string toLdapSerialize(mixed $value)	The value will be converted to a string by using serialize().
mixed fromLdapUnserialize(string $value)	The value will be converted from a string by using unserialize().

CHAPTER

ZEND\LDAP\DN

`Zend\Ldap\Dn` provides an object-oriented interface to manipulating *LDAP* distinguished names (DN). The parameter `$caseFold` that is used in several methods determines the way DN attributes are handled regarding their case. Allowed values for this parameter are:

ZendLdapDn::ATTR_CASEFOLD_NONE No case-folding will be done.

ZendLdapDn::ATTR_CASEFOLD_UPPER All attributes will be converted to upper-case.

ZendLdapDn::ATTR_CASEFOLD_LOWER All attributes will be converted to lower-case.

The default case-folding is `Zend\Ldap\Dn::ATTR_CASEFOLD_NONE` and can be set with `Zend\Ldap\Dn::setDefaultCaseFold()`. Each instance of `Zend\Ldap\Dn` can have its own case-folding-setting. If the `$caseFold` parameter is omitted in method-calls it defaults to the instance's case-folding setting.

The class implements *ArrayAccess* to allow indexer-access to the different parts of the DN. The *ArrayAccess*-methods proxy to `Zend\Ldap\Dn::get($offset, 1, null)` for *offsetGet(integer $offset)*, to `Zend\Ldap\Dn::set($offset, $value)` for `offsetSet()` and to `Zend\Ldap\Dn::remove($offset, 1)` for `offsetUnset()`. `offsetExists()` simply checks if the index is within the bounds.

Table 140.1: Zend\Ldap\Dn API

Method	Description
Zend\Ldap\Dn factory(string|array $dn, string|null $caseFold)	Creates a Zend\Ldap\Dn instance from an array or a string. The array must conform to the array structure detailed under Zend\Ldap\Dn::implodeDn().
Zend\Ldap\Dn fromString(string $dn, string|null $caseFold)	Creates a Zend\Ldap\Dn instance from a string.
Zend\Ldap\Dn fromArray(array $dn, string|null $caseFold)	Creates a Zend\Ldap\Dn instance from an array. The array must conform to the array structure detailed under Zend\Ldap\Dn::implodeDn().
array getRdn(string|null $caseFold)	Gets the RDN of the current DN. The return value is an array with the RDN attribute names its keys and the RDN attribute values.
string getRdnString(string|null $caseFold)	Gets the RDN of the current DN. The return value is a string.
Zend\Ldap\Dn getParentDn(integer $levelUp)	Gets the DN of the current DN's ancestor $levelUp levels up the tree. $levelUp defaults to 1.
array get(integer $index, integer $length, string|null $caseFold)	Returns a slice of the current DN determined by $index and $length. $index starts with 0 on the DN part from the left.
Zend\Ldap\Dn set(integer $index, array $value)	Replaces a DN part in the current DN. This operation manipulates the current instance.
Zend\Ldap\Dn remove(integer $index, integer $length)	Removes a DN part from the current DN. This operation manipulates the current instance. $length defaults to 1
Zend\Ldap\Dn append(array $value)	Appends a DN part to the current DN. This operation manipulates the current instance.
Zend\Ldap\Dn prepend(array $value)	Prepends a DN part to the current DN. This operation manipulates the current instance.
Zend\Ldap\Dn insert(integer $index, array $value)	Inserts a DN part after the index $index to the current DN. This operation manipulates the current instance.
void setCaseFold(string|null $caseFold)	Sets the case-folding option to the current DN instance. If $caseFold is NULL the default case-folding setting (Zend\Ldap\Dn::ATTR_CASEFOLD_NONE by default or set via Zend\Ldap\Dn::setDefaultCaseFold() will be set for the current instance.
string toString(string|null $caseFold)	Returns DN as a string.
array toArray(string|null $caseFold)	Returns DN as an array.
string __toString()	Returns DN as a string - proxies to Zend\Ldap\Dn::toString(null).
void setDefaultCaseFold(string $caseFold)	Sets the default case-folding option used by all instances on creation by default. Already existing instances are not affected by this setting.
array escapeValue(string|array $values)	Escapes a DN value according to RFC 2253.
array unescapeValue(string|array $values)	Undoes the conversion done by Zend\Ldap\Dn::escapeValue().
array explodeDn(string $dn, array &$keys, array &$vals, string|null $caseFold)	Explodes the DN $dn into an array containing all parts of the given DN. $keys optionally receive DN keys (e.g. CN, OU, DC, ...). $vals optionally receive DN values. The resulting array will be of type array(array("cn" => "name1", "uid" => "user"), array("cn" => "name2"), array("dc" => "example"), array("dc" => "org")) for a DN of cn=name1+uid=user,cn=name2,dc=example,dc=org.
boolean checkDn(string $dn, array &$keys, array &$vals, string|null $caseFold)	Checks if a given DN $dn is malformed. If $keys or $keys and $vals are given, these arrays will be filled with the appropriate DN keys and values.
string implodeRdn(array $part, string|null $caseFold)	Returns a DN part in the form $attribute=$value
string implodeDn(array $dnArray, string|null $caseFold, string $separator)	Implodes an array in the form delivered by Zend\Ldap\Dn::explodeDn() to a DN string. $separator defaults to ',' but some LDAP servers also understand ';'. $dnArray must of type array(array("cn" => "name1", "uid" => "user"), array("cn" => "name2"), array("dc" => "example"), array("dc" => "org"))
boolean isChildOf(string|Zend\Ldap\Dn $childDn, string|Zend\Ldap\Dn	Checks if given $childDn is beneath $parentDn subtree.

CHAPTER

ONE

ZEND\LDAP\FILTER

Table 141.1: Zend\Ldap\Filter API

Method	Description
Zend\Ldap\Filter equals(string $attr, string $value)	Creates an 'equals' filter: (attr=value).
Zend\Ldap\Filter begins(string $attr, string $value)	Creates an 'begins with' filter: (attr=value*).
Zend\Ldap\Filter ends(string $attr, string $value)	Creates an 'ends with' filter: (attr=*value).
Zend\Ldap\Filter contains(string $attr, string $value)	Creates an 'contains' filter: (attr=*value*).
Zend\Ldap\Filter greater(string $attr, string $value)	Creates an 'greater' filter: (attr>value).
Zend\Ldap\Filter greaterOrEqual(string $attr, string $value)	Creates an 'greater or equal' filter: (attr>=value).
Zend\Ldap\Filter less(string $attr, string $value)	Creates an 'less' filter: (attr<value).
Zend\Ldap\Filter lessOrEqual(string $attr, string $value)	Creates an 'less or equal' filter: (attr<=value).
Zend\Ldap\Filter approx(string $attr, string $value)	Creates an 'approx' filter: (attr~=value).
Zend\Ldap\Filter any(string $attr)	Creates an 'any' filter: (attr=*).
Zend\Ldap\Filter string(string $filter)	Creates a simple custom string filter. The user is responsible for all value-escaping as the filter is used as is.
Zend\Ldap\Filter mask(string $mask, string $value,...)	Creates a filter from a string mask. All $value parameters will be escaped and substituted into $mask by using sprintf()
Zend\Ldap\Filter andFilter(Zend\Ldap\Filter\AbstractFilter $filter,...)	Creates an 'and' filter from all arguments given.
Zend\Ldap\Filter orFilter(Zend\Ldap\Filter\AbstractFilter $filter,...)	Creates an 'or' filter from all arguments given.
__construct(string $attr, string $value, string $filtertype, string|null $prepend, string|null $append)	Constructor. Creates an arbitrary filter according to the parameters supplied. The resulting filter will be a concatenation $attr . $filtertype . $prepend . $value . $append. Normally this constructor is not needed as all filters can be created by using the appropriate factory methods.
string toString()	Returns a string representation of the filter.
string __toString()	Returns a string representation of the filter. Proxies to Zend\Ldap\Filter::toString().
Zend\Ldap\Filter\AbstractFilter negate()	Negates the current filter.
Zend\Ldap\Filter\AbstractFilter addAnd(Zend\Ldap\Filter\AbstractFilter $filter,...)	Creates an 'and' filter from the current filter and all filters passed in as the arguments.
Zend\Ldap\Filter\AbstractFilter addOr(Zend\Ldap\Filter\AbstractFilter $filter,...)	Creates an 'or' filter from the current filter and all filters passed in as the arguments.

CHAPTER

TWO

ZEND\LDAP\NODE

`Zend\Ldap\Node` includes the magic property accessors `__set()`, `__get()`, `__unset()` and `__isset()` to access the attributes by their name. They proxy to `Zend\Ldap\Node::setAttribute()`, `Zend\Ldap\Node::getAttribute()`, `Zend\Ldap\Node::deleteAttribute()` and `Zend\Ldap\Node::existsAttribute()` respectively. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. `Zend\Ldap\Node` also implements *Iterator* and *RecursiveIterator* to allow for recursive tree-traversal.

Method	Description
Zend\Ldap\Ldap getLdap()	Returns the cu
Zend\Ldap\Node attachLdap(Zend\Ldap\Ldap $ldap)	Attach the cur
Zend\Ldap\Node detachLdap()	Detach node f
boolean isAttached()	Checks if the
Zend\Ldap\Node create(string|array|Zend\Ldap\Dn $dn, array $objectClass)	Factory metho
Zend\Ldap\Node fromLdap(string|array|Zend\Ldap\Dn $dn, Zend\Ldap\Ldap $ldap)	Factory metho
Zend\Ldap\Node fromArray((array $data, boolean $fromDataSource)	Factory metho
boolean isNew()	Tells if the no
boolean willBeDeleted()	Tells if this no
Zend\Ldap\Node delete()	Marks this no
boolean willBeMoved()	Tells if this no
Zend\Ldap\Node update(Zend\Ldap\Ldap $ldap)	Sends all pend
Zend\Ldap\Dn getCurrentDn()	Gets the curre
Zend\Ldap\Dn getDn()	Gets the origi
string getDnString(string $caseFold)	Gets the origi
array getDnArray(string $caseFold)	Gets the origi
string getRdnString(string $caseFold)	Gets the RDN
array getRdnArray(string $caseFold)	Gets the RDN
Zend\Ldap\Node setDn(Zend\Ldap\Dn|string|array $newDn)	Sets the new
Zend\Ldap\Node move(Zend\Ldap\Dn|string|array $newDn)	This is an alia
Zend\Ldap\Node rename(Zend\Ldap\Dn|string|array $newDn)	This is an alia
array getObjectClass()	Returns the ob
Zend\Ldap\Node setObjectClass(array|string $value)	Sets the objec
Zend\Ldap\Node appendObjectClass(array|string $value)	Appends to th
string toLdif(array $options)	Returns a LD
array getChangedData()	Gets changed
array getChanges()	Returns all ch
string toString()	Returns the D

Method	Description
string __toString()	Casts to string
array toArray(boolean $includeSystemAttributes)	Returns an arr
string toJson(boolean $includeSystemAttributes)	Returns a JSO
array getData(boolean $includeSystemAttributes)	Returns the no
boolean existsAttribute(string $name, boolean $emptyExists)	Checks wheth
boolean attributeHasValue(string $name, mixed|array $value)	Checks if the
integer count()	Returns the nu
mixed getAttribute(string $name, integer|null $index)	Gets a LDAP
array getAttributes(boolean $includeSystemAttributes)	Gets all attrib
Zend\Ldap\Node setAttribute(string $name, mixed $value)	Sets a LDAP
Zend\Ldap\Node appendToAttribute(string $name, mixed $value)	Appends to a
array|integer getDateTimeAttribute(string $name, integer|null $index)	Gets a LDAP
Zend\Ldap\Node setDateTimeAttribute(string $name, integer|array $value, boolean $utc)	Sets a LDAP
Zend\Ldap\Node appendToDateTimeAttribute(string $name, integer|array $value, boolean $utc)	Appends to a
Zend\Ldap\Node setPasswordAttribute(string $password, string $hashType, string $attribName)	Sets a LDAP
Zend\Ldap\Node deleteAttribute(string $name)	Deletes a LDA
void removeDuplicatesFromAttribute(string$name)	Removes dup
void removeFromAttribute(string $attribName, mixed|array $value)	Removes the
boolean exists(Zend\Ldap\Ldap $ldap)	Checks if the
Zend\Ldap\Node reload(Zend\Ldap\Ldap $ldap)	Reloads the cu
Zend\Ldap\Node\Collection searchSubtree(string|Zend\Ldap\Filter\AbstractFilter $filter, integer $scope, string $sort)	Searches the n
integer countSubtree(string|Zend\Ldap\Filter\AbstractFilter $filter, integer $scope)	Count the noc
integer countChildren()	Count the noc
Zend\Ldap\Node\Collection searchChildren(string|Zend\Ldap\Filter\AbstractFilter $filter, string $sort)	Searches the n
boolean hasChildren()	Returns whetl
Zend\Ldap\Node\ChildrenIterator getChildren()	Returns all ch
Zend\Ldap\Node getParent(Zend\Ldap\Ldap $ldap)	Returns the pa

CHAPTER

THREE

ZEND\LDAP\NODE\ROOTDSE

The following methods are available on all vendor-specific subclasses.

`Zend\Ldap\Node\RootDse` includes the magic property accessors `__get()` and `__isset()` to access the attributes by their name. They proxy to `Zend\Ldap\Node\RootDse::getAttribute()` and `Zend\Ldap\Node\RootDse::existsAttribute()` respectively. `__set()` and `__unset()` are also implemented but they throw a *BadMethodCallException* as modifications are not allowed on RootDSE nodes. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. `offsetSet()` and `offsetUnset()` also throw a *BadMethodCallException* due ro obvious reasons.

Table 143.1: Zend\Ldap\Node\RootDse API

Method	Description
Zend\Ldap\Dn getDn()	Gets the DN of the current node as a Zend\Ldap\Dn.
string getDnString(string $caseFold)	Gets the DN of the current node as a string.
array getDnArray(string $caseFold)	Gets the DN of the current node as an array.
string getRdnString(string $caseFold)	Gets the RDN of the current node as a string.
array getRdnArray(string $caseFold)	Gets the RDN of the current node as an array.
array getObjectClass()	Returns the objectClass of the node.
string toString()	Returns the DN of the current node - proxies to Zend\Ldap\Dn::getDnString().
string __toString()	Casts to string representation - proxies to Zend\Ldap\Dn::toString().
array toArray(boolean $includeSystemAttributes)	Returns an array representation of the current node. If $includeSystemAttributes is FALSE (defaults to TRUE) the system specific attributes are stripped from the array. Unlike Zend\Ldap\Node\RootDse::getAttributes() the resulting array contains the DN with key 'dn'.
string toJson(boolean $includeSystemAttributes)	Returns a JSON representation of the current node using Zend\Ldap\Node\RootDse::toArray().
array getData(boolean $includeSystemAttributes)	Returns the node's attributes. The array contains all attributes in its internal format (no conversion).
boolean existsAttribute(string $name, boolean $emptyExists)	Checks whether a given attribute exists. If $emptyExists is FALSE, empty attributes (containing only array()) are treated as non-existent returning FALSE. If $emptyExists is TRUE, empty attributes are treated as existent returning TRUE. In this case the method returns FALSE only if the attribute name is missing in the key-collection.
boolean attributeHasValue(string $name, mixed|array $value)	Checks if the given value(s) exist in the attribute. The method returns TRUE only if all values in $value are present in the attribute. Comparison is done strictly (respecting the data type).
integer count()	Returns the number of attributes in the node. Implements Countable.
mixed getAttribute(string $name, integer|null $index)	Gets a LDAP attribute. Data conversion is applied using Zend\Ldap\Attribute::getAttribute().
array getAttributes(boolean $includeSystemAttributes)	Gets all attributes of node. If $includeSystemAttributes is FALSE (defaults to TRUE) the system specific attributes are stripped from the array.
array|integer getDateTimeAttribute(string $name, integer|null $index)	Gets a LDAP date/time attribute. Data conversion is applied using Zend\Ldap\Attribute::getDateTimeAttribute().
Zend\Ldap\Node\RootDse reload(Zend\Ldap\Ldap $ldap)	Reloads the current node's attributes from the given LDAP server.
Zend\Ldap\Node\RootDse create(Zend\Ldap\Ldap $ldap)	Factory method to create the RootDSE.
array getNamingContexts()	Gets the namingContexts.
string|null getSubschemaSubentry()	Gets the subschemaSubentry.
boolean supportsVersion(string|int|array $versions)	Determines if the LDAP version is supported.
boolean supportsSaslMechanism(string|array $mechlist)	Determines if the sasl mechanism is supported.
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_GENERICfor unknown LDAP serversZend\Ldap\Node\RootDse::SERVER_TYPE_OPENLDAPfor OpenLDAP serversZend\Ldap\Node\RootDse::SERVER_TYPE_ACTIVEDIRECTORYfor Microsoft ActiveDirectory

143.1 OpenLDAP

Additionally the common methods above apply to instances of `Zend\Ldap\Node\RootDse\OpenLdap`.

Note: Refer to LDAP Operational Attributes and Objects for information on the attributes of OpenLDAP RootDSE.

Table 143.2: Zend\Ldap\Node\RootDse\OpenLdap API

Method	Description
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_OPENLDAP
string|null getConfigContext()	Gets the configContext.
string|null getMonitorContext()	Gets the monitorContext.
boolean supportsControl(string|array $oids)	Determines if the control is supported.
boolean supportsExtension(string|array $oids)	Determines if the extension is supported.
boolean supportsFeature(string|array $oids)	Determines if the feature is supported.

143.2 ActiveDirectory

Additionally the common methods above apply to instances of `Zend\Ldap\Node\RootDse\ActiveDirectory`.

Note: Refer to RootDSE for information on the attributes of Microsoft ActiveDirectory RootDSE.

Table 143.3: Zend\Ldap\Node\RootDse\ActiveDirectory API

Method	Description
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_ACTIVEDIRECTORY
string|null getConfigurationNamingContext()	Gets the configurationNamingContext.
string|null getCurrentTime()	Gets the currentTime.
string|null getDefaultNamingContext()	Gets the defaultNamingContext.
string|null getDnsHostName()	Gets the dnsHostName.
string|null getDomainController-Functionality()	Gets the domainControllerFunctionality.
string|null getDomainFunctionality()	Gets the domainFunctionality.
string|null getDsServiceName()	Gets the dsServiceName.
string|null getForestFunctionality()	Gets the forestFunctionality.
string|null getHighestCommittedUSN()	Gets the highestCommittedUSN.
string|null getIsGlobalCatalogReady()	Gets the isGlobalCatalogReady.
string|null getIsSynchronized()	Gets the isSynchronized.
string|null getLdapServiceName()	Gets the ldapServiceName.
string|null getRootDomainNamingContext()	Gets the rootDomainNamingContext.
string|null getSchemaNamingContext()	Gets the schemaNamingContext.
string|null getServerName()	Gets the serverName.
boolean supportsCapability(string|array $oids)	Determines if the capability is supported.
boolean supportsControl(string|array $oids)	Determines if the control is supported.
boolean supportsPolicy(string|array $policies)	Determines if the version is supported.

143.3 eDirectory

Additionally the common methods above apply to instances of *ZendLdapNodeRootDseeDirectory*.

Note: Refer to Getting Information about the LDAP Server for information on the attributes of Novell eDirectory RootDSE.

Table 143.4: Zend\Ldap\Node\RootDse\eDirectory API

Method	Description
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_EDIRECTORY
boolean supportsExtension(string|array $oids)	Determines if the extension is supported.
string|null getVendorName()	Gets the vendorName.
string|null getVendorVersion()	Gets the vendorVersion.
string|null getDsaName()	Gets the dsaName.
string|null getStatisticsErrors()	Gets the server statistics "errors".
string|null getStatisticsSecurityErrors()	Gets the server statistics "securityErrors".
string|null getStatisticsChainings()	Gets the server statistics "chainings".
string|null getStatisticsReferralsReturned()	Gets the server statistics "referralsReturned".
string|null getStatisticsExtendedOps()	Gets the server statistics "extendedOps".
string|null getStatisticsAbandonOps()	Gets the server statistics "abandonOps".
string|null getStatisticsWholeSubtreeSearchOps()	Gets the server statistics "wholeSubtreeSearchOps".

CHAPTER

FOUR

ZEND\LDAP\NODE\SCHEMA

The following methods are available on all vendor-specific subclasses.

ZendLdapNodeSchema includes the magic property accessors *__get()* and *__isset()* to access the attributes by their name. They proxy to *ZendLdapNodeSchema::getAttribute()* and *ZendLdapNodeSchema::existsAttribute()* respectively. *__set()* and *__unset()* are also implemented, but they throw a *BadMethodCallException* as modifications are not allowed on RootDSE nodes. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. *offsetSet()* and *offsetUnset()* also throw a *BadMethodCallException* due to obvious reasons.

Table 144.1: Zend\Ldap\Node\Schema API

Method	Description
Zend\Ldap\Dn getDn()	Gets the DN of the current node as a Zend\Ldap\Dn.
string getDnString(string $caseFold)	Gets the DN of the current node as a string.
array getDnArray(string $caseFold)	Gets the DN of the current node as an array.
string getRdnString(string $caseFold)	Gets the RDN of the current node as a string.
array getRdnArray(string $caseFold)	Gets the RDN of the current node as an array.
array getObjectClass()	Returns the objectClass of the node.
string toString()	Returns the DN of the current node - proxies to Zend\Ldap\Dn::getDnString().
string __toString()	Casts to string representation - proxies to Zend\Ldap\Dn::toString().
array toArray(boolean $includeSystemAttributes)	Returns an array representation of the current node. If $includeSystemAttributes is FALSE (defaults to TRUE), the system specific attributes are stripped from the array. Unlike Zend\Ldap\Node\Schema::getAttributes(), the resulting array contains the DN with key 'dn'.
string toJson(boolean $includeSystemAttributes)	Returns a JSON representation of the current node using Zend\Ldap\Node\Schema::toArray().
array getData(boolean $includeSystemAttributes)	Returns the node's attributes. The array contains all attributes in its internal format (no conversion).
boolean existsAttribute(string $name, boolean $emptyExists)	Checks whether a given attribute exists. If $emptyExists is FALSE, empty attributes (containing only array()) are treated as non-existent returning FALSE. If $emptyExists is TRUE, empty attributes are treated as existent returning TRUE. In this case the method returns FALSE only if the attribute name is missing in the key-collection.
boolean attributeHasValue(string $name, mixed|array $value)	Checks if the given value(s) exist in the attribute. The method returns TRUE only if all values in $value are present in the attribute. Comparison is done strictly (respecting the data type).
integer count()	Returns the number of attributes in the node. Implements Countable.
mixed getAttribute(string $name, integer|null $index)	Gets a LDAP attribute. Data conversion is applied using Zend\Ldap\Attribute::getAttribute().
array getAttributes(boolean $includeSystemAttributes)	Gets all attributes of node. If $includeSystemAttributes is FALSE (defaults to TRUE) the system specific attributes are stripped from the array.
array|integer getDateTimeAttribute(string $name, integer|null $index)	Gets a LDAP date/time attribute. Data conversion is applied using Zend\Ldap\Attribute::getDateTimeAttribute().
Zend\Ldap\Node\Schema reload(Zend\Ldap\Ldap $ldap)	Reloads the current node's attributes from the given LDAP server.
Zend\Ldap\Node\Schema create(Zend\Ldap\Ldap $ldap)	Factory method to create the Schema node.
array getAttributeTypes()	Gets the attribute types as an array of .
array getObjectClasses()	Gets the object classes as an array of Zend\Ldap\Node\Schema\ObjectClass\Interface.

Table 144.2: Zend\Ldap\Node\Schema\AttributeType\Interface API

Method	Description
string getName()	Gets the attribute name.
string getOid()	Gets the attribute OID.
string getSyntax()	Gets the attribute syntax.
int|null getMaxLength()	Gets the attribute maximum length.
boolean isSingleValued()	Returns if the attribute is single-valued.
string getDescription()	Gets the attribute description

Table 144.3: Zend\Ldap\Node\Schema\ObjectClass\Interface API

Method	Description
string getName()	Returns the objectClass name.
string getOid()	Returns the objectClass OID.
array getMustContain()	Returns the attributes that this objectClass must contain.
array getMayContain()	Returns the attributes that this objectClass may contain.
string getDescription()	Returns the attribute description
integer getType()	Returns the objectClass type. The method returns one of the following values: Zend\Ldap\Node\Schema::OBJECTCLASS_TYPE_UNKNOWNfor unknown class typesZend\Ldap\Node\Schema::OBJECTCLASS_TYPE_STRUCTURALfor structural classesZend\Ldap\Node\Schema::OBJECTCLASS_TYPE_ABSTRACTfor abstract classesZend\Ldap\Node\Schema::OBJECTCLASS_TYPE_AUXILIARYfor auxiliary classes
array getParentClasses()	Returns the parent objectClasses of this class. This includes structural, abstract and auxiliary objectClasses.

Classes representing attribute types and object classes extend *ZendLdapNodeSchemaAbstractItem* which provides some core methods to access arbitrary attributes on the underlying *LDAP* node. *ZendLdapNodeSchemaAbstractItem* includes the magic property accessors *__get()* and *__isset()* to access the attributes by their name. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. *offsetSet()* and *offsetUnset()* throw a *BadMethodCallException* as modifications are not allowed on schema information nodes.

Table 144.4: Zend\Ldap\Node\Schema\AbstractItem API

Method	Description
array getData()	Gets all the underlying data from the schema information node.
integer count()	Returns the number of attributes in this schema information node. Implements Countable.

144.1 OpenLDAP

Additionally the common methods above apply to instances of *ZendLdapNodeSchemaOpenLDAP*.

Table 144.5: Zend\Ldap\Node\Schema\OpenLDAP API

Method	Description
array getLdapSyntaxes()	Gets the LDAP syntaxes.
array getMatchingRules()	Gets the matching rules.
array getMatchingRuleUse()	Gets the matching rule use.

Table 144.6: Zend\Ldap\Node\Schema\AttributeType\OpenLDAP API

Method	Description
Zend\Ldap\Node\Schema\AttributeType\OpenLdap|null getParent()	Returns the parent attribute type in the inheritance tree if one exists.

Table 144.7: Zend\Ldap\Node\Schema\ObjectClass\OpenLDAP API

Method	Description
array getParents()	Returns the parent object classes in the inheritance tree if one exists. The returned array is an array of Zend\Ldap\Node\Schema\ObjectClass\OpenLdap.

144.2 ActiveDirectory

Note: Schema browsing on ActiveDirectory servers

Due to restrictions on Microsoft ActiveDirectory servers regarding the number of entries returned by generic search routines and due to the structure of the ActiveDirectory schema repository, schema browsing is currently **not** available for Microsoft ActiveDirectory servers.

ZendLdapNodeSchemaActiveDirectory does not provide any additional methods.

Table 144.8: Zend\Ldap\Node\Schema\AttributeType\ActiveDirectory API

Zend\Ldap\Node\Schema\AttributeType\ActiveDirectory does not provide any additional methods.

Table 144.9: Zend\Ldap\Node\Schema\ObjectClass\ActiveDirectory API

Zend\Ldap\Node\Schema\ObjectClass\ActiveDirectory does not provide any additional methods.

CHAPTER

FIVE

ZEND\LDAP\LDIF\ENCODER

Table 145.1: Zend\Ldap\Ldif\Encoder API

Method	Description
array decode(string $string)	Decodes the string $string into an array of LDIF items.
string encode(scalar|array|Zend\Ldap\Node $value, array $options)	Encode $value into a LDIF representation. $options is an array that may contain the following keys: 'sort' Sort the given attributes with dn following objectClass and following all other attributes sorted alphabetically. TRUE by default. 'version' The LDIF format version. 1 by default. 'wrap' The line-length. 78 by default to conform to the LDIF specification.

CHAPTER

SIX

USAGE SCENARIOS

146.1 Authentication scenarios

146.1.1 OpenLDAP

146.1.2 ActiveDirectory

146.2 Basic CRUD operations

146.2.1 Retrieving data from the LDAP

Getting an entry by its DN

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
/*
$hm is an array of the following structure
array(
    'dn'          => 'cn=Hugo Müller,ou=People,dc=my,dc=local',
    'cn'          => array('Hugo Müller'),
    'sn'          => array('Müller'),
    'objectclass' => array('inetOrgPerson', 'top'),
    ...
)
*/
```

Check for the existence of a given DN

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$isThere = $ldap->exists('cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Count children of a given DN

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$childrenCount = $ldap->countChildren(
                            'cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Searching the LDAP tree

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$result = $ldap->search('(objectclass=*)',
                        'ou=People,dc=my,dc=local',
                        Zend\Ldap\Ldap::SEARCH_SCOPE_ONE);
foreach ($result as $item) {
    echo $item["dn"] . ': ' . $item['cn'][0] . PHP_EOL;
}
```

146.2.2 Adding data to the LDAP

Add a new entry to the LDAP

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$entry = array();
Zend\Ldap\Attribute::setAttribute($entry, 'cn', 'Hans Meier');
Zend\Ldap\Attribute::setAttribute($entry, 'sn', 'Meier');
Zend\Ldap\Attribute::setAttribute($entry, 'objectClass', 'inetOrgPerson');
$ldap->add('cn=Hans Meier,ou=People,dc=my,dc=local', $entry);
```

146.2.3 Deleting from the LDAP

Delete an existing entry from the LDAP

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ldap->delete('cn=Hans Meier,ou=People,dc=my,dc=local');
```

146.2.4 Updating the LDAP

Update an existing entry on the LDAP

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
```

```
$hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
Zend\Ldap\Attribute::setAttribute($hm, 'mail', 'mueller@my.local');
Zend\Ldap\Attribute::setPassword($hm,
                                 'newPa$$w0rd',
                                 Zend\Ldap\Attribute::PASSWORD_HASH_SHA1);
$ldap->update('cn=Hugo Müller,ou=People,dc=my,dc=local', $hm);
```

146.3 Extended operations

146.3.1 Copy and move entries in the LDAP

Copy a LDAP entry recursively with all its descendants

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ldap->copy('cn=Hugo Müller,ou=People,dc=my,dc=local',
            'cn=Hans Meier,ou=People,dc=my,dc=local',
            true);
```

Move a LDAP entry recursively with all its descendants to a different subtree

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ldap->moveToSubtree('cn=Hugo Müller,ou=People,dc=my,dc=local',
                     'ou=Dismissed,dc=my,dc=local',
                     true);
```

CHAPTER

SEVEN

TOOLS

147.1 Creation and modification of DN strings

147.2 Using the filter API to create search filters

Create simple LDAP filters

```
$f1  = Zend\Ldap\Filter::equals('name', 'value');         // (name=value)
$f2  = Zend\Ldap\Filter::begins('name', 'value');         // (name=value*)
$f3  = Zend\Ldap\Filter::ends('name', 'value');           // (name=*value)
$f4  = Zend\Ldap\Filter::contains('name', 'value');       // (name=*value*)
$f5  = Zend\Ldap\Filter::greater('name', 'value');        // (name>value)
$f6  = Zend\Ldap\Filter::greaterOrEqual('name', 'value'); // (name>=value)
$f7  = Zend\Ldap\Filter::less('name', 'value');           // (name<value)
$f8  = Zend\Ldap\Filter::lessOrEqual('name', 'value');    // (name<=value)
$f9  = Zend\Ldap\Filter::approx('name', 'value');         // (name~=value)
$f10 = Zend\Ldap\Filter::any('name');                     // (name=*)
```

Create more complex LDAP filters

```
$f1 = Zend\Ldap\Filter::ends('name', 'value')->negate(); // (!(name=*value))

$f2 = Zend\Ldap\Filter::equals('name', 'value');
$f3 = Zend\Ldap\Filter::begins('name', 'value');
$f4 = Zend\Ldap\Filter::ends('name', 'value');

// (&(name=value)(name=value*)(name=*value))
$f5 = Zend\Ldap\Filter::andFilter($f2, $f3, $f4);

// (|(name=value)(name=value*)(name=*value))
$f6 = Zend\Ldap\Filter::orFilter($f2, $f3, $f4);
```

147.3 Modify LDAP entries using the Attribute API

CHAPTER

EIGHT

OBJECT-ORIENTED ACCESS TO THE LDAP TREE USING ZEND\LDAP\NODE

148.1 Basic CRUD operations

148.1.1 Retrieving data from the LDAP

148.1.2 Getting a node by its DN

148.1.3 Searching a node's subtree

148.1.4 Adding a new node to the LDAP

148.1.5 Deleting a node from the LDAP

148.1.6 Updating a node on the LDAP

148.2 Extended operations

148.2.1 Copy and move nodes in the LDAP

148.3 Tree traversal

Traverse LDAP tree recursively

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ri = new RecursiveIteratorIterator($ldap->getBaseNode(),
                                    RecursiveIteratorIterator::SELF_FIRST);
foreach ($ri as $rdn => $n) {
    var_dump($n);
}
```

CHAPTER

NINE

GETTING INFORMATION FROM THE LDAP SERVER

149.1 RootDSE

See the following documents for more information on the attributes contained within the RootDSE for a given *LDAP* server.

- OpenLDAP
- Microsoft ActiveDirectory
- Novell eDirectory

Getting hands on the RootDSE

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$rootdse = $ldap->getRootDse();
$serverType = $rootdse->getServerType();
```

149.2 Schema Browsing

Getting hands on the server schema

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$schema = $ldap->getSchema();
$classes = $schema->getObjectClasses();
```

149.2.1 OpenLDAP

149.2.2 ActiveDirectory

Note: Schema browsing on ActiveDirectory servers

Due to restrictions on Microsoft ActiveDirectory servers regarding the number of entries returned by generic search routines and due to the structure of the ActiveDirectory schema repository, schema browsing is currently **not** available for Microsoft ActiveDirectory servers.

CHAPTER

SERIALIZING LDAP DATA TO AND FROM LDIF

150.1 Serialize a LDAP entry to LDIF

```
$data = array(
    'dn'                          => 'uid=rogasawara,ou=,o=Airius',
    'objectclass'                 => array('top',
                                           'person',
                                           'organizationalPerson',
                                           'inetOrgPerson'),
    'uid'                         => array('rogasawara'),
    'mail'                        => array('rogasawara@airius.co.jp'),
    'givenname;lang-ja'           => array(''),
    'sn;lang-ja'                  => array(''),
    'cn;lang-ja'                  => array(' '),
    'title;lang-ja'               => array(' '),
    'preferredlanguage'           => array('ja'),
    'givenname'                   => array(''),
    'sn'                          => array(''),
    'cn'                          => array(' '),
    'title'                       => array(' '),
    'givenname;lang-ja;phonetic' => array(''),
    'sn;lang-ja;phonetic'         => array(''),
    'cn;lang-ja;phonetic'         => array(' '),
    'title;lang-ja;phonetic'      => array(' '),
    'givenname;lang-en'           => array('Rodney'),
    'sn;lang-en'                  => array('Ogasawara'),
    'cn;lang-en'                  => array('Rodney Ogasawara'),
    'title;lang-en'               => array('Sales, Director'),
);
$ldif = Zend\Ldap\Ldif\Encoder::encode($data, array('sort' => false,
                                                    'version' => null));
/*
$ldif contains:
dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6YOoLG89QWlyaXVz
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: rogasawara
mail: rogasawara@airius.co.jp
givenname;lang-ja:: 44Ot44OJ44OL44O8
sn;lang-ja:: 5bCP56yg5Y6f
cn;lang-ja:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
```

```
title;lang-ja:: 5Za25qWt6YOoIOmDqOmVtw==
preferredlanguage: ja
givenname:: 44Ot44OJ44OL44O8
sn:: 5bCP56yg5Y6f
cn:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title:: 5Za25qWt6YOoIOmDqOmVtw==
givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
givenname;lang-en: Rodney
sn;lang-en: Ogasawara
cn;lang-en: Rodney Ogasawara
title;lang-en: Sales, Director
*/
```

150.2 Deserialize a LDIF string into a LDAP entry

```
$ldif = "dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6YOoLG89QWlyaXVz
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: rogasawara
mail: rogasawara@airius.co.jp
givenname;lang-ja:: 44Ot44OJ44OL44O8
sn;lang-ja:: 5bCP56yg5Y6f
cn;lang-ja:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title;lang-ja:: 5Za25qWt6YOoIOmDqOmVtw==
preferredlanguage: ja
givenname:: 44Ot44OJ44OL44O8
sn:: 5bCP56yg5Y6f
cn:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title:: 5Za25qWt6YOoIOmDqOmVtw==
givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
givenname;lang-en: Rodney
sn;lang-en: Ogasawara
cn;lang-en: Rodney Ogasawara
title;lang-en: Sales, Director";
$data = Zend\Ldap\Ldif\Encoder::decode($ldif);
/*
$data = array(
    'dn'                        => 'uid=rogasawara,ou=,o=Airius',
    'objectclass'               => array('top',
                                         'person',
                                         'organizationalPerson',
                                         'inetOrgPerson'),
    'uid'                       => array('rogasawara'),
    'mail'                      => array('rogasawara@airius.co.jp'),
    'givenname;lang-ja'         => array(''),
    'sn;lang-ja'                => array(''),
    'cn;lang-ja'                => array(' '),
    'title;lang-ja'             => array(' '),
```

```
    'preferredlanguage'          => array('ja'),
    'givenname'                  => array(''),
    'sn'                         => array(''),
    'cn'                         => array(' '),
    'title'                      => array(' '),
    'givenname;lang-ja;phonetic' => array(''),
    'sn;lang-ja;phonetic'        => array(''),
    'cn;lang-ja;phonetic'        => array(' '),
    'title;lang-ja;phonetic'     => array(' '),
    'givenname;lang-en'          => array('Rodney'),
    'sn;lang-en'                 => array('Ogasawara'),
    'cn;lang-en'                 => array('Rodney Ogasawara'),
    'title;lang-en'              => array('Sales, Director'),
);
*/
```

CHAPTER

ONE

THE AUTOLOADERFACTORY

151.1 Overview

Starting with version 2.0, Zend Framework now offers multiple autoloader strategies. Often, it will be useful to employ multiple autoloading strategies; as an example, you may have a class map for your most used classes, but want to use a PSR-0 style autoloader for 3rd party libraries.

While you could potentially manually configure these, it may be more useful to define the autoloader configuration somewhere and cache it. For these cases, the `AutoloaderFactory` will be useful.

151.2 Quick Start

Configuration may be stored as a PHP array, or in some form of configuration file. As an example, consider the following PHP array:

```
$config = array(
    'Zend\Loader\ClassMapAutoloader' => array(
        'application' => APPLICATION_PATH . '/.classmap.php',
        'zf'          => APPLICATION_PATH . '/../library/Zend/.classmap.php',
    ),
    'Zend\Loader\StandardAutoloader' => array(
        'namespaces' => array(
            'Phly\Mustache' => APPLICATION_PATH . '/../library/Phly/Mustache',
            'Doctrine'      => APPLICATION_PATH . '/../library/Doctrine',
        ),
    ),
);
```

An equivalent INI-style configuration might look like the following:

```
Zend\Loader\ClassMapAutoloader.application = APPLICATION_PATH "/.classmap.php"
Zend\Loader\ClassMapAutoloader.zf          = APPLICATION_PATH "/../library/Zend/.classmap.php"
Zend\Loader\StandardAutoloader.namespaces.Phly\Mustache = APPLICATION_PATH "/../library/Phly/Mustache
Zend\Loader\StandardAutoloader.namespaces.Doctrine       = APPLICATION_PATH "/../library/Doctrine"
```

Once you have your configuration in a PHP array, you simply pass it to the `AutoloaderFactory`.

```
// This example assumes ZF is on your include_path.
// You could also load the factory class from a path relative to the
// current script, or via an absolute path.
require_once 'Zend/Loader/AutoloaderFactory.php';
Zend\Loader\AutoloaderFactory::factory($config);
```

The `AutoloaderFactory` will instantiate each autoloader with the given options, and also call its `register()` method to register it with the SPL autoloader.

151.3 Configuration Options

AutoloaderFactory Options

$options The `AutoloaderFactory` expects an associative array or `Traversable` object. Keys should be valid autoloader class names, and the values should be the options that should be passed to the class constructor.

Internally, the `AutoloaderFactory` checks to see if the autoloader class referenced exists. If not, it will use the *StandardAutoloader* to attempt to load the class via the `include_path` (or, in the case of "Zend"-namespaced classes, using the Zend Framework library path). If the class is not found, or does not implement the *SplAutoloader* interface, an exception will be raised.

151.4 Available Methods

factory Instantiate and register autoloaders `factory($options)`

factory() This method is **static**, and is used to instantiate autoloaders and register them with the SPL autoloader. It expects either an array or `Traversable` object as denoted in the *Options section*.

getRegisteredAutoloaders Retrieve a list of all autoloaders registered using the factory `getRegisteredAutoloaders()`

getRegisteredAutoloaders() This method is **static**, and may be used to retrieve a list of all autoloaders registered via the `factory()` method. It returns simply an array of autoloader instances.

getRegisteredAutoloader Retrieve an autoloader by class name `getRegisteredAutoloader($class)`

getRegisteredAutoloader() This method is **static**, and is used to retrieve a specific autoloader. It expects a string with the autoloader class name. If the autoloader is not registered, an exception will be thrown.

unregisterAutoloaders Unregister all autoloaders registered via the factory. `unregisterAutoloaders()`

unregisterAutoloaders() This method is **static**, and can be used to unregister all autoloaders that were registered via the factory. Note that this will **not** unregister autoloaders that were registered outside of the factory.

unregisterAutoloader Unregister an autoloader registered via the factory. `unregisterAutoloader($class)`

unregisterAutoloader() This method is **static**, and can be used to unregister an autoloader that was registered via the factory. Note that this will **not** unregister autoloaders that were registered outside of the factory. If the autoloader is registered via the factory, after unregistering it will return `TRUE`, otherwise `FALSE`.

151.5 Examples

Please see the *Quick Start* for a detailed example.

CHAPTER

TWO

THE STANDARDAUTOLOADER

152.1 Overview

`Zend\Loader\StandardAutoloader` is designed as a PSR-0-compliant autoloader. It assumes a 1:1 mapping of the namespace+classname to the filesystem, wherein namespace separators and underscores are translated to directory separators. A simple statement that illustrates how resolution works is as follows:

```
$filename = str_replace(array('_', '\\'), DIRECTORY_SEPARATOR, $classname)
          . '.php';
```

Previous incarnations of PSR-0-compliant autoloaders in Zend Framework have relied upon the `include_path` for file lookups. This has led to a number of issues:

- Due to the use of `include`, if the file is not found, a warning is raised – even if another autoloader is capable of resolving the class later.
- Documenting how to setup the `include_path` has proven to be a difficult concept to convey.
- If multiple Zend Framework installations exist on the `include_path`, the first one on the path wins – even if that was not the one the developer intended.

To solve these problems, the `StandardAutoloader` by default requires that you explicitly register namespace/path pairs (or vendor prefix/path pairs), and will only load a file if it exists within the given path. Multiple pairs may be provided.

As a measure of last resort, you may also use the `StandardAutoloader` as a "fallback" autoloader – one that will look for classes of any namespace or vendor prefix on the `include_path`. This practice is not recommended, however, due to performance implications.

Finally, as with all autoloaders in Zend Framework, the `StandardAutoloader` is capable of registering itself with PHP's SPL autoloader registry.

Note: Vocabulary: Namespaces vs. Vendor Prefixes

In terms of autloading, a "namespace" corresponds to PHP's own definition of namespaces in PHP versions 5.3 and above.

A "vendor prefix" refers to the practice, popularized in PHP versions prior to 5.3, of providing a pseudo-namespace in the form of underscore-separated words in class names. As an example, the class `Phly_Couch_Document` uses a vendor prefix of "Phly", and a component prefix of "Phly_Couch" – but it is a class sitting in the global namespace within PHP 5.3.

The `StandardAutoloader` is capable of loading either namespaced or vendor prefixed class names, but treats them separately when attempting to match them to an appropriate path.

152.2 Quick Start

Basic use of the `StandardAutoloader` requires simply registering namespace/path pairs. This can either be done at instantiation, or via explicit method calls after the object has been initialized. Calling `register()` will register the autoloader with the SPL autoloader registry.

If the option key 'autoregister_zf' is set to true then the class will register the "Zend" namespace to the directory above where its own classfile is located on the filesystem.

Manual Configuration

```
// This example assumes ZF is on your include_path.
// You could also load the autoloader class from a path relative to the
// current script, or via an absolute path.
require_once 'Zend/Loader/StandardAutoloader.php';
$loader = new Zend\Loader\StandardAutoloader(array('autoregister_zf' => true));

// Register the "Phly" namespace:
$loader->registerNamespace('Phly', APPLICATION_PATH . '/../library/Phly');

// Register the "Scapi" vendor prefix:
$loader->registerPrefix('Scapi', APPLICATION_PATH . '/../library/Scapi');

// Optionally, specify the autoloader as a "fallback" autoloader;
// this is not recommended.
$loader->setFallbackAutoloader(true);

// Register with spl_autoload:
$loader->register();
```

Configuration at Instantiation

The `StandardAutoloader` may also be configured at instantiation. Please note:

- The argument passed may be either an array or a `Traversable` object.
- The argument passed is also a valid argument for passing to the `setOptions()` method.

The following is equivalent to the previous example.

```
require_once 'Zend/Loader/StandardAutoloader.php';
$loader = new Zend\Loader\StandardAutoloader(array(
    'autoregister_zf' => true,
    'namespaces' => array(
        'Phly' => APPLICATION_PATH . '/../library/Phly',
    ),
    'prefixes' => array(
        'Scapi' => APPLICATION_PATH . '/../library/Scapi',
    ),
    'fallback_autoloader' => true,
));

// Register with spl_autoload:
$loader->register();
```

152.3 Configuration Options

The `StandardAutoloader` defines the following options.

StandardAutoloader Options

namespaces An associative array of namespace/path pairs. The path should be an absolute path or path relative to the calling script, and contain only classes that live in that namespace (or its subnamespaces). By default, the "Zend" namespace is registered, pointing to the parent directory of the file defining the `StandardAutoloader`.

prefixes An associative array of vendor prefix/path pairs. The path should be an absolute path or path relative to the calling script, and contain only classes that begin with the provided vendor prefix.

fallback_autoloader A boolean value indicating whether or not this instance should act as a "fallback" autoloader (i.e., look for classes of any namespace or vendor prefix on the `include_path`). By default, `false`.

autoregister_zf An boolean value indicating that the class should register the "Zend" namespace to the directory above where its own classfile is located on the filesystem.

152.4 Available Methods

__construct Initialize a new instance of the object `__construct($options = null)`

Constructor Takes an optional `$options` argument. This argument may be an associative array or `Traversable` object. If not null, the argument is passed to *setOptions()*.

setOptions Set object state based on provided options. `setOptions($options)`

setOptions() Takes an argument of either an associative array or `Traversable` object. Recognized keys are detailed under *Configuration options*, with the following behaviors:

- The `namespaces` value will be passed to *registerNamespaces()*.
- The `prefixes` value will be passed to *registerPrefixes()*.
- The `fallback_autoloader` value will be passed to *setFallbackAutoloader()*.

setFallbackAutoloader Enable/disable fallback autoloader status `setFallbackAutoloader($flag)`

setFallbackAutoloader() Takes a boolean flag indicating whether or not to act as a fallback autoloader when registered with the SPL autoloader.

isFallbackAutoloader Query fallback autoloader status `isFallbackAutoloader()`

isFallbackAutoloader() Indicates whether or not this instance is flagged as a fallback autoloader.

registerNamespace Register a namespace with the autoloader `registerNamespace($namespace, $directory)`

registerNamespace() Register a namespace with the autoloader, pointing it to a specific directory on the filesystem for class resolution. For classes matching that initial namespace, the autoloader will then perform lookups within that directory.

registerNamespaces Register multiple namespaces with the autoloader `registerNamespaces($namespaces)`

registerNamespaces() Accepts either an array or `Traversable` object. It will then iterate through the argument, and pass each item to *registerNamespace()*.

registerPrefix Register a vendor prefix with the autoloader. `registerPrefix($prefix, $directory)`

registerPrefix() Register a vendor prefix with the autoloader, pointing it to a specific directory on the filesystem for class resolution. For classes matching that initial vendor prefix, the autoloader will then perform lookups within that directory.

registerPrefixes Register many vendor prefixes with the autoloader `registerPrefixes($prefixes)`

registerPrefixes() Accepts either an array or `Traversable` object. It will then iterate through the argument, and pass each item to *registerPrefix()*.

autoload Attempt to load a class. `autoload($class)`

autoload() Attempts to load the class specified. Returns a boolean `false` on failure, or a string indicating the class loaded on success.

register Register with spl_autoload. `register()`

register() Registers the `autoload()` method of the current instance with `spl_autoload_register()`.

152.5 Examples

Please review the *examples in the quick start* for usage.

CHAPTER

THREE

THE CLASSMAPAUTOLOADER

153.1 Overview

The `ClassMapAutoloader` is designed with performance in mind. The idea behind it is simple: when asked to load a class, see if it's in the map, and, if so, load the file associated with the class in the map. This avoids unnecessary filesystem operations, and can also ensure the autoloader "plays nice" with opcode caches and PHP's realpath cache.

Zend Framework provides a tool for generating these class maps; you can find it in `bin/classmap_generator.php` of the distribution. Full documentation of this is provided in the *Class Map generator* section.

153.2 Quick Start

The first step is to generate a class map file. You may run this over any directory containing source code anywhere underneath it.

```
php classmap_generator.php Some/Directory/
```

This will create a file named `Some/Directory/autoload_classmap.php`, which is a PHP file returning an associative array that represents the class map.

Within your code, you will now instantiate the `ClassMapAutoloader`, and provide it the location of the map.

```
// This example assumes ZF is on your include_path.
// You could also load the autoloader class from a path relative to the
// current script, or via an absolute path.
require_once 'Zend/Loader/ClassMapAutoloader.php';
$loader = new Zend\Loader\ClassMapAutoloader();

// Register the class map:
$loader->registerAutoloadMap('Some/Directory/autoload_classmap.php');

// Register with spl_autoload:
$loader->register();
```

At this point, you may now use any classes referenced in your class map.

153.3 Configuration Options

The `ClassMapAutoloader` defines the following options.

ClassMapAutoloader Options

$options The `ClassMapAutoloader` expects an array of options, where each option is either a filename referencing a class map, or an associative array of class name/filename pairs.

As an example:

```
// Configuration defining both a file-based class map, and an array map
$config = array(
    __DIR__ . '/library/autoloader_classmap.php', // file-based class map
    array(                                        // array class map
        'Application\Bootstrap' => __DIR__ . '/application/Bootstrap.php',
        'Test\Bootstrap'        => __DIR__ . '/tests/Bootstrap.php',
    ),
);
```

153.4 Available Methods

__construct Initialize and configure the object `__construct($options = null)`

Constructor Used during instantiation of the object. Optionally, pass options, which may be either an array or `Traversable` object; this argument will be passed to *setOptions()*.

setOptions Configure the autoloader `setOptions($options)`

setOptions() Configures the state of the autoloader, including registering class maps. Expects an array or `Traversable` object; the argument will be passed to *registerAutoloadMaps()*.

registerAutoloadMap Register a class map `registerAutoloadMap($map)`

registerAutoloadMap() Registers a class map with the autoloader. `$map` may be either a string referencing a PHP script that returns a class map, or an array defining a class map.

More than one class map may be registered; each will be merged with the previous, meaning it's possible for a later class map to overwrite entries from a previously registered map.

registerAutoloadMaps Register multiple class maps at once `registerAutoloadMaps($maps)`

registerAutoloadMaps() Register multiple class maps with the autoloader. Expects either an array or `Traversable` object; it then iterates over the argument and passes each value to *registerAutoloadMap()*.

getAutoloadMap Retrieve the current class map `getAutoloadMap()`

getAutoloadMap() Retrieves the state of the current class map; the return value is simply an array.

autoload Attempt to load a class. `autoload($class)`

autoload() Attempts to load the class specified. Returns a boolean `false` on failure, or a string indicating the class loaded on success.

register Register with spl_autoload. `register()`

register() Registers the `autoload()` method of the current instance with `spl_autoload_register()`.

153.5 Examples

Using configuration to seed ClassMapAutoloader

Often, you will want to configure your `ClassMapAutoloader`. These values may come from a configuration file, a cache (such as ShMem or memcached), or a simple PHP array. The following is an example of a PHP array that could be used to configure the autoloader:

```
// Configuration defining both a file-based class map, and an array map
$config = array(
    APPLICATION_PATH . '/../library/autoloader_classmap.php', // file-based class map
    array(                                // array class map
        'Application\Bootstrap' => APPLICATION_PATH . '/Bootstrap.php',
        'Test\Bootstrap'        => APPLICATION_PATH . '/../tests/Bootstrap.php',
    ),
);
```

An equivalent INI style configuration might look like this:

```
classmap.library = APPLICATION_PATH "/../library/autoloader_classmap.php"
classmap.resources.Application\Bootstrap = APPLICATION_PATH "/Bootstrap.php"
classmap.resources.Test\Bootstrap = APPLICATION_PATH "/../tests/Bootstrap.php"
```

Once you have your configuration, you can pass it either to the constructor of the `ClassMapAutoloader`, to its `setOptions()` method, or to `registerAutoloadMaps()`.

```
/* The following are all equivalent */

// To the constructor:
$loader = new Zend\Loader\ClassMapAutoloader($config);

// To setOptions():
$loader = new Zend\Loader\ClassMapAutoloader();
$loader->setOptions($config);

// To registerAutoloadMaps():
$loader = new Zend\Loader\ClassMapAutoloader();
$loader->registerAutoloadMaps($config);
```

CHAPTER

FOUR

THE MODULEAUTOLOADER

154.1 Overview

`Zend\Loader\ModuleAutoloader` is a special implementation of the *Zend\Loader\SplAutoloader* interface, used by *Zend\ModuleManager* to autoload `Module` classes from different sources.

Apart from being able to autoload modules from directories, the `ModuleAutoloader` can also autoload modules packaged as Phar archives, which allows for packaging your modules in a single file for easier distribution. Supported archive formats are: `.phar`, `.phar.gz`, `.phar.bz2`, `.phar.tar`, `.phar.tar.gz`, `.phar.tar.bz2`, `.phar.zip`, `.tar`, `tar.gz`, `.tar.bz2` and `.zip`. It is, however, recommended to avoid compressing your packages (be it either *gz*, *bz2* or *zip* compression), as it introduces additional CPU overhead to every request.

154.2 Quickstart

As the `ModuleAutoloader` is meant to be used with the `ModuleManager`, for examples of it's usage and how to configure it, please see the *Module Autoloader Usage* section of the `ModuleManager` documentation.

154.3 Configuration Options

The `ModuleAutoloader` defines the following options.

ModuleAutoloader Options

$options The `ModuleAutoloader` expects an array of options, where each option is either a path to scan for modules, or a key/value pair of explicit module paths. In the case of explicit module paths, the key is the module's name, and the value is the path to that module.

```
$options = array(
    '/path/to/modules',
    '/path/to/other/modules',
    'MyModule' => '/explicit/path/mymodule-v1.2'
);
```

154.4 Available Methods

__construct Initialize and configure the object `__construct($options = null)`

Constructor Used during instantiation of the object. Optionally, pass options, which may be either an array or `Traversable` object; this argument will be passed to *setOptions()*.

setOptions Configure the module autoloader `setOptions($options)`

setOptions() Configures the state of the autoloader, registering paths to modules. Expects an array or `Traversable` object; the argument will be passed to *registerPaths()*.

autoload Attempt to load a `Module` class. `autoload($class)`

autoload() Attempts to load the specified `Module` class. Returns a boolean `false` on failure, or a string indicating the class loaded on success.

register Register with spl_autoload. `register()`

register() Registers the `autoload()` method of the current instance with `spl_autoload_register()`.

unregister Unregister with spl_autoload. `unregister()`

unregister() Unregisters the `autoload()` method of the current instance with `spl_autoload_unregister()`.

registerPaths Register multiple paths with the autoloader. `registerPaths($paths)`

registerPaths() Register a paths to modules. Expects an array or `Traversable` object. For an example array, please see the *Configuration options* section.

registerPath Register a single path with the autoloader. `registerPath($path, $moduleName=false)`

registerPath() Register a single path with the autoloader. The first parameter, `$path`, is expected to be a string. The second parameter, `$moduleName`, is expected to be a module name, which allows for registering an explicit path to that module.

getPaths Get all paths registered with the autoloader. `getPaths()`

getPaths() Returns an array of all the paths registered with the current instance of the autoloader.

154.5 Examples

Please review the *examples in the quick start* for usage.

CHAPTER

FIVE

THE SPLAUTOLOADER INTERFACE

155.1 Overview

While any valid PHP callback may be registered with `spl_autoload_register()`, Zend Framework autoloaders often provide more flexibility by being stateful and allowing configuration. To provide a common interface, Zend Framework provides the `SplAutoloader` interface.

Objects implementing this interface provide a standard mechanism for configuration, a method that may be invoked to attempt to load a class, and a method for registering with the SPL autoloading mechanism.

155.2 Quick Start

To create your own autoloading mechanism, simply create a class implementing the `SplAutoloader` interface (you may review the methods defined in the *Methods section*). As a simple example, consider the following autoloader, which will look for a class file named after the class within a list of registered directories.

```
namespace Custom;

use Zend\Loader\SplAutoloader;

class ModifiedIncludePathAutoloader implements SplAutoloader
{
    protected $paths = array();

    public function __construct($options = null)
    {
        if (null !== $options) {
            $this->setOptions($options);
        }
    }

    public function setOptions($options)
    {
        if (!is_array($options) && !($options instanceof \Traversable)) {
            throw new \InvalidArgumentException();
        }

        foreach ($options as $path) {
            if (!in_array($path, $this->paths)) {
                $this->paths[] = $path;
            }
```

```
        }
        return $this;
    }

    public function autoload($classname)
    {
        $filename = $classname . '.php';
        foreach ($this->paths as $path) {
            $test = $path . DIRECTORY_SEPARATOR . $filename;
            if (file_exists($test)) {
                return include($test);
            }
        }
        return false;
    }

    public function register()
    {
        spl_autoload_register(array($this, 'autoload'));
    }
}
```

To use this `ModifiedIncludePathAutoloader` from the previous example:

```
$options = array(
   '/path/one',
   '/path/two'
);
$autoloader = new Custom\ModifiedIncludePathAutoloader($options);
$autoloader->register();
```

155.3 Configuration Options

This component defines no configuration options, as it is an interface.

155.4 Available Methods

__construct Initialize and configure an autoloader `__construct($options = null)`

> **Constructor** Autoloader constructors should optionally receive configuration options. Typically, if received, these will be passed to the `setOptions()` method to process.

setOptions Configure the autoloader state `setOptions($options)`

> **setOptions()** Used to configure the autoloader. Typically, it should expect either an array or a `Traversable` object, though validation of the options is left to implementation. Additionally, it is recommended that the method return the autoloader instance in order to implement a fluent interface.

autoload Attempt to resolve a class name to the file defining it `autoload($classname)`

> **autoload()** This method should be used to resolve a class name to the file defining it. When a positive match is found, return the class name; otherwise, return a boolean false.

register Register the autoloader with the SPL autoloader `register()`

register() Should be used to register the autoloader instance with `spl_autoload_register()`. Invariably, the method should look like the following:

```
public function register()
{
    spl_autoload_register(array($this, 'autoload'));
}
```

155.5 Examples

Please see the *Quick Start* for a complete example.

CHAPTER

SIX

THE PLUGINCLASSLOADER

156.1 Overview

Resolving plugin names to class names is a common requirement within Zend Framework applications. The `PluginClassLoader` implements the interfaces *PluginClassLocator*, *ShortNameLocator*, and `IteratorAggregate`, providing a simple mechanism for aliasing plugin names to classnames for later retrieval.

While it can act as a standalone class, it is intended that developers will extend the class to provide a per-component plugin map. This allows seeding the map with the most often-used plugins, while simultaneously allowing the end-user to overwrite existing or register new plugins.

Additionally, `PluginClassLoader` provides the ability to statically seed all new instances of a given `PluginClassLoader` or one of its extensions (via Late Static Binding). If your application will always call for defining or overriding particular plugin maps on given `PluginClassLoader` extensions, this is a powerful capability.

156.2 Quick Start

Typical use cases involve simply instantiating a `PluginClassLoader`, seeding it with one or more plugin/class name associations, and then using it to retrieve the class name associated with a given plugin name.

```
use Zend\Http\HeaderLoader;

// Provide a global map, or override defaults:
HeaderLoader::addStaticMap(array(
    'xrequestedfor' => 'My\Http\Header\XRequestedFor',
));

// Instantiate the loader:
$loader = new Zend\Http\HeaderLoader();

// Register a new plugin:
$loader->registerPlugin('xForwardedFor', 'My\Http\Header\XForwardedFor');

// Load/retrieve the associated plugin class:
$class = $loader->load('xrequestedfor'); // 'My\Http\Header\XRequestedFor'
```

Note: Case Sensitivity

The `PluginClassLoader` is designed to do case-insensitive plugin name lookups. While the above example defines a "xForwardedFor" plugin name, internally, this will be stored as simply "xforwardedfor". If another plugin is

registered with simply a different word case, it will overwrite this entry.

156.3 Configuration Options

PluginClassLoader Options

$map The constructor may take a single option, an array or `Traversable` object of key/value pairs corresponding to a plugin name and class name, respectively.

156.4 Available Methods

__construct Instantiate and initialize the loader `__construct($map = null)`

__construct() The constructor is used to instantiate and initialize the plugin class loader. If passed a string, an array, or a `Traversable` object, it will pass this to the *registerPlugins()* method in order to seed (or overwrite) the plugin class map.

addStaticMap Statically seed the plugin loader map `addStaticMap($map)`

addStaticMap() Static method for globally pre-seeding the loader with a class map. It accepts either an array or `Traversable` object of plugin name/class name pairs.

When using this method, be certain you understand the precedence in which maps will be merged; in decreasing order of preference:

- Manually registered plugin/class name pairs (e.g., via *registerPlugin()* or *registerPlugins()*).
- A map passed to the constructor .
- The static map.
- The map defined within the class itself.

Also, please note that calling the method will **not** affect any instances already created.

registerPlugin Register a plugin/class association `registerPlugin($shortName, $className)`

registerPlugin() Defined by the *PluginClassLocator* interface. Expects two string arguments, the plugin `$shortName`, and the class `$className` which it represents.

registerPlugins Register many plugin/class associations at once `registerPlugins($map)`

registerPlugins() Expects a string, an array or `Traversable` object of plugin name/class name pairs representing a plugin class map.

If a string argument is provided, `registerPlugins()` assumes this is a class name. If the class does not exist, an exception will be thrown. If it does, it then instantiates the class and checks to see whether or not it implements `Traversable`.

unregisterPlugin Remove a plugin/class association from the map `unregisterPlugin($shortName)`

unregisterPlugin() Defined by the `PluginClassLocator` interface; remove a plugin/class association from the plugin class map.

getRegisteredPlugins Return the complete plugin class map `getRegisteredPlugins()`

getRegisteredPlugins() Defined by the `PluginClassLocator` interface; return the entire plugin class map as an array.

isLoaded Determine if a given plugin name resolves `isLoaded($name)`

isLoaded() Defined by the `ShortNameLocator` interface; determine if the given plugin has been resolved to a class name.

getClassName Return the class name to which a plugin resolves `getClassName($name)`

getClassName() Defined by the `ShortNameLocator` interface; return the class name to which a plugin name resolves.

load Resolve a plugin name `load($name)`

load() Defined by the `ShortNameLocator` interface; attempt to resolve a plugin name to a class name. If successful, returns the class name; otherwise, returns a boolean `false`.

getIterator Return iterator capable of looping over plugin class map `getIterator()`

getIterator() Defined by the `IteratorAggregate` interface; allows iteration over the plugin class map. This can come in useful for using `PluginClassLoader` instances to other `PluginClassLoader` instances in order to merge maps.

156.5 Examples

Using Static Maps

It's often convenient to provide global overrides or additions to the maps in a `PluginClassLoader` instance. This can be done using the `addStaticMap()` method:

```
use Zend\Loader\PluginClassLoader;

PluginClassLoader::addStaticMap(array(
    'xrequestedfor' => 'My\Http\Header\XRequestedFor',
));
```

Any later instances created will now have this map defined, allowing you to load that plugin.

```
use Zend\Loader\PluginClassLoader;

$loader = new PluginClassLoader();
$class = $loader->load('xrequestedfor'); // My\Http\Header\XRequestedFor
```

Creating a pre-loaded map

In many cases, you know exactly which plugins you may be drawing upon on a regular basis, and which classes they will refer to. In this case, simply extend the `PluginClassLoader` and define the map within the extending class.

```
namespace My\Plugins;

use Zend\Loader\PluginClassLoader;

class PluginLoader extends PluginClassLoader
{
    /**
     * @var array Plugin map
     */
    protected $plugins = array(
        'foo'    => 'My\Plugins\Foo',
```

```
        'bar'     => 'My\Plugins\Bar',
        'foobar' => 'My\Plugins\FooBar',
    );
}
```

At this point, you can simply instantiate the map and use it.

```
$loader = new My\Plugins\PluginLoader();
$class  = $loader->load('foobar'); // My\Plugins\FooBar
```

`PluginClassLoader` makes use of late static binding, allowing per-class static maps. If you want to allow defining a *static map* specific to this extending class, simply declare a protected static `$staticMap` property:

```
namespace My\Plugins;

use Zend\Loader\PluginClassLoader;

class PluginLoader extends PluginClassLoader
{
    protected static $staticMap = array();

    // ...
}
```

To inject the static map, use the extending class' name to call the static `addStaticMap()` method.

```
PluginLoader::addStaticMap(array(
    'baz'     => 'My\Plugins\Baz',
));
```

Extending a plugin map using another plugin map

In some cases, a general map class may already exist; as an example, most components in Zend Framework that utilize a plugin broker have an associated `PluginClassLoader` extension defining the plugins available for that component within the framework. What if you want to define some additions to these? Where should that code go?

One possibility is to define the map in a configuration file, and then inject the configuration into an instance of the plugin loader. This is certainly trivial to implement, but removes the code defining the plugin map from the library.

An alternate solution is to define a new plugin map class. The class name or an instance of the class may then be passed to the constructor or `registerPlugins()`.

```
namespace My\Plugins;

use Zend\Loader\PluginClassLoader;
use Zend\Http\HeaderLoader;

class PluginLoader extends PluginClassLoader
{
    /**
     * @var array Plugin map
     */
    protected $plugins = array(
        'foo'     => 'My\Plugins\Foo',
        'bar'     => 'My\Plugins\Bar',
        'foobar' => 'My\Plugins\FooBar',
    );
}
```

```

// Inject in constructor:
$loader = new HeaderLoader('My\Plugins\PluginLoader');
$loader = new HeaderLoader(new PluginLoader());

// Or via registerPlugins():
$loader->registerPlugins('My\Plugins\PluginLoader');
$loader->registerPlugins(new PluginLoader());
```

CHAPTER

SEVEN

THE SHORTNAMELOCATOR INTERFACE

157.1 Overview

Within Zend Framework applications, it's often expedient to provide a mechanism for using class aliases instead of full class names to load adapters and plugins, or to allow using aliases for the purposes of slipstreaming alternate implementations into the framework.

In the first case, consider the adapter pattern. It's often unwieldy to utilize a full class name (e.g., `Zend\Cloud\DocumentService\Adapter\SimpleDb`); using the short name of the adapter, `SimpleDb`, would be much simpler.

In the second case, consider the case of helpers. Let us assume we have a "url" helper; you may find that while the shipped helper does 90% of what you need, you'd like to extend it or provide an alternate implementation. At the same time, you don't want to change your code to reflect the new helper. In this case, a short name allows you to alias an alternate class to utilize.

Classes implementing the `ShortNameLocator` interface provide a mechanism for resolving a short name to a fully qualified class name; how they do so is left to the implementers, and may combine strategies defined by other interfaces, such as *PluginClassLocator*.

157.2 Quick Start

Implementing a `ShortNameLocator` is trivial, and requires only three methods, as shown below.

```
namespace Zend\Loader;

interface ShortNameLocator
{
    public function isLoaded($name);
    public function getClassName($name);
    public function load($name);
}
```

157.3 Configuration Options

This component defines no configuration options, as it is an interface.

157.4 Available Methods

isLoaded Is the requested plugin loaded? `isLoaded($name)`

isLoaded() Implement this method to return a boolean indicating whether or not the class has been able to resolve the plugin name to a class.

getClassName Get the class name associated with a plugin name `getClassName($name)`

getClassName() Implement this method to return the class name associated with a plugin name.

load Resolve a plugin to a class name `load($name)`

load() This method should resolve a plugin name to a class name.

157.5 Examples

Please see the *Quick Start* for the interface specification.

CHAPTER

EIGHT

THE PLUGINCLASSLOCATOR INTERFACE

158.1 Overview

The `PluginClassLocator` interface describes a component capable of maintaining an internal map of plugin names to actual class names. Classes implementing this interface can register and unregister plugin/class associations, and return the entire map.

158.2 Quick Start

Classes implementing the `PluginClassLocator` need to implement only three methods, as illustrated below.

```
namespace Zend\Loader;

interface PluginClassLocator
{
    public function registerPlugin($shortName, $className);
    public function unregisterPlugin($shortName);
    public function getRegisteredPlugins();
}
```

158.3 Configuration Options

This component defines no configuration options, as it is an interface.

158.4 Available Methods

registerPlugin Register a mapping of plugin name to class name `registerPlugin($shortName, $className)`

registerPlugin() Implement this method to add or overwrite plugin name/class name associations in the internal plugin map. `$shortName` will be aliased to `$className`.

unregisterPlugin Remove a plugin/class name association `unregisterPlugin($shortName)`

unregisterPlugin() Implement this to allow removing an existing plugin mapping corresponding to `$shortName`.

getRegisteredPlugins Retrieve the map of plugin name/class name associations `getRegisteredPlugins()`

getRegisteredPlugins() Implement this to allow returning the plugin name/class name map.

158.5 Examples

Please see the *Quick Start* for the interface specification.

CHAPTER

NINE

THE CLASS MAP GENERATOR UTILITY: BIN/CLASSMAP_GENERATOR.PHP

159.1 Overview

The script `bin/classmap_generator.php` can be used to generate class map files for use with *the ClassMapAutoloader*.

Internally, it consumes both *Zend\Console\Getopt* (for parsing command-line options) and *Zend\File\ClassFileLocator* for recursively finding all PHP class files in a given tree.

159.2 Quick Start

You may run the script over any directory containing source code. By default, it will look in the current directory, and will write the script to `autoloader_classmap.php` in the directory you specify.

```
php classmap_generator.php Some/Directory/
```

159.3 Configuration Options

Class Map Generator Options

–help or -h Returns the usage message. If any other options are provided, they will be ignored.

–library or -l Expects a single argument, a string specifying the library directory to parse. If this option is not specified, it will assume the current working directory.

–output or -o Where to write the autoload class map file. If not provided, assumes "autoload_classmap.php" in the library directory.

–append or -a Append to autoload file if it exists.

–overwrite or -w If an autoload class map file already exists with the name as specified via the `--output` option, you can overwrite it by specifying this flag. Otherwise, the script will not write the class map and return a warning.

CHAPTER

OVERVIEW OF ZEND\LOG

`Zend\Log\Logger` is a component for general purpose logging. It supports multiple log backends, formatting messages sent to the log, and filtering messages from being logged. These functions are divided into the following objects:

- A Logger (instance of `Zend\Log\Logger`) is the object that your application uses the most. You can have as many Logger objects as you like; they do not interact. A Logger object must contain at least one Writer, and can optionally contain one or more Filters.
- A Writer (inherits from `Zend\Log\Writer\AbstractWriter`) is responsible for saving data to storage.
- A Filter (implements `Zend\Log\Filter\FilterInterface`) blocks log data from being saved. A filter is applied to an individual writer. Filters can be chained.
- A Formatter (implements `Zend\Log\Formatter\FormatterInterface`) can format the log data before it is written by a Writer. Each Writer has exactly one Formatter.

160.1 Creating a Log

To get started logging, instantiate a Writer and then pass it to a Logger instance:

```
$logger = new Zend\Log\Logger;
$writer = new Zend\Log\Writer\Stream('php://output');

$logger->addWriter($writer);
```

It is important to note that the Logger must have at least one Writer. You can add any number of Writers using the Log's `addWriter()` method.

You can also add a priority to each writer. The priority is specified as number and passed as second argument in the `addWriter()` method.

Another way to add a writer to a Logger is to use the name of the writer as follow:

```
$logger = new Zend\Log\Logger;

$logger->addWriter('stream', null, array('stream' => 'php://output'));
```

In this example we passed the stream `php://output` as parameter (as array).

160.2 Logging Messages

To log a message, call the `log()` method of a Log instance and pass it the message with a corresponding priority:

```
$logger->log(Zend\Log\Logger::INFO, 'Informational message');
```

The first parameter of the `log()` method is an integer `priority` and the second parameter is a string `message`. The priority must be one of the priorities recognized by the Logger instance. This is explained in the next section. There is also an optional third parameter used to pass extra informations to the writer's log.

A shortcut is also available. Instead of calling the `log()` method, you can call a method by the same name as the priority:

```
$logger->log(Zend\Log\Logger::INFO, 'Informational message');
$logger->info('Informational message');

$logger->log(Zend\Log\Logger::EMERG, 'Emergency message');
$logger->emerg('Emergency message');
```

160.3 Destroying a Log

If the Logger object is no longer needed, set the variable containing it to `NULL` to destroy it. This will automatically call the `shutdown()` instance method of each attached Writer before the Log object is destroyed:

```
$logger = null;
```

Explicitly destroying the log in this way is optional and is performed automatically at *PHP* shutdown.

160.4 Using Built-in Priorities

The `Zend\Log\Logger` class defines the following priorities:

```
EMERG   = 0;  // Emergency: system is unusable
ALERT   = 1;  // Alert: action must be taken immediately
CRIT    = 2;  // Critical: critical conditions
ERR     = 3;  // Error: error conditions
WARN    = 4;  // Warning: warning conditions
NOTICE  = 5;  // Notice: normal but significant condition
INFO    = 6;  // Informational: informational messages
DEBUG   = 7;  // Debug: debug messages
```

These priorities are always available, and a convenience method of the same name is available for each one.

The priorities are not arbitrary. They come from the BSD syslog protocol, which is described in RFC-3164. The names and corresponding priority numbers are also compatible with another *PHP* logging system, PEAR Log, which perhaps promotes interoperability between it and `Zend\Log\Logger`.

Priority numbers descend in order of importance. `EMERG` (0) is the most important priority. `DEBUG` (7) is the least important priority of the built-in priorities. You may define priorities of lower importance than `DEBUG`. When selecting the priority for your log message, be aware of this priority hierarchy and choose appropriately.

160.5 Understanding Log Events

When you call the `log()` method or one of its shortcuts, a log event is created. This is simply an associative array with data describing the event that is passed to the writers. The following keys are always created in this array: `timestamp`, `message`, `priority`, and `priorityName`.

The creation of the `event` array is completely transparent.

160.6 Log PHP Errors

`Zend\Log\Logger` can also be used to log *PHP* errors and intercept Exceptions. Calling the static method `registerErrorHandler($logger)` will add the $logger object before the current PHP error handler, and will pass the error along as well.

```
$logger = new Zend\Log\Logger;
$writer = new Zend\Log\Writer\Stream('php://output');

$logger->addWriter($writer);

Zend\Log\Logger::registerErrorHandler($logger);
```

If you want to unregister the error handler you can use the `unregisterErrorHandler()` static method.

Table 160.1: Zend\Log\Logger events from PHP errors fields matching handler (int $errno , string $errstr [, string $errfile [, int $errline [, array $errcontext]]]) from set_error_handler

Name	Error Handler Parameter	Description
message	errstr	Contains the error message, as a string.
errno	errno	Contains the level of the error raised, as an integer.
file	errfile	Contains the filename that the error was raised in, as a string.
line	errline	Contains the line number the error was raised at, as an integer.
context	errcontext	(optional) An array that points to the active symbol table at the point the error occurred. In other words, errcontext will contain an array of every variable that existed in the scope the error was triggered in. User error handler must not modify error context.

You can also configure a Logger to intercept Exceptions using the static method `registerExceptionHandler($logger)`.

CHAPTER

ONE

WRITERS

A Writer is an object that inherits from `Zend\Log\Writer\AbstractWriter`. A Writer's responsibility is to record log data to a storage backend.

161.1 Writing to Streams

`Zend\Log\Writer\Stream` sends log data to a PHP stream.

To write log data to the *PHP* output buffer, use the URL `php://output`. Alternatively, you can send log data directly to a stream like `STDERR` (`php://stderr`).

```
$writer = new Zend\Log\Writer\Stream('php://output');
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('Informational message');
```

To write data to a file, use one of the Filesystem URLs:

```
$writer = new Zend\Log\Writer\Stream('/path/to/logfile');
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('Informational message');
```

By default, the stream opens in the append mode ("a"). To open it with a different mode, the `Zend\Log\Writer\Stream` constructor accepts an optional second parameter for the mode.

The constructor of `Zend\Log\Writer\Stream` also accepts an existing stream resource:

```
$stream = @fopen('/path/to/logfile', 'a', false);
if (! $stream) {
    throw new Exception('Failed to open stream');
}

$writer = new Zend\Log\Writer\Stream($stream);
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('Informational message');
```

You cannot specify the mode for existing stream resources. Doing so causes a `Zend\Log\Exception` to be thrown.

161.2 Writing to Databases

`Zend\Log\Writer\Db` writes log information to a database table using `Zend\Db\Adapter\Adapter`. The constructor of `Zend\Log\Writer\Db` receives a `Zend\Db\Adapter\Adapter` instance, a table name, an optional mapping of event data to database columns, and an optional string contains the character separator for the log array:

```
$dbconfig = array(
    // Sqlite Configuration
    'driver' => 'Pdo',
    'dsn' => 'sqlite:' . __DIR__ . '/tmp/sqlite.db',
);
$db = new Zend\Db\Adapter\Adapter($dbconfig);

$writer = new Zend\Log\Writer\Db($db, 'log_table_name');
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('Informational message');
```

The example above writes a single row of log data to the database table named 'log_table_name' table. The database column will be created according to the event array generated by the `Zend\Log\Logger` instance.

If we specify the mapping of the events with the database columns the log will store in the database only the selected fields.

```
$dbconfig = array(
    // Sqlite Configuration
    'driver' => 'Pdo',
    'dsn' => 'sqlite:' . __DIR__ . '/tmp/sqlite.db',
);
$db = new Zend\Db\Adapter\Adapter($dbconfig);

$mapping = array(
    'timestamp' => 'date',
    'priority'  => 'type',
    'message'   => 'event'
);
$writer = new Zend\Log\Writer\Db($db, 'log_table_name', $mapping);
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('Informational message');
```

The previous example will store only the log information timestamp, priority and message in the database fields date, type and event.

The `Zend\Log\Writer\Db` has a fourth optional parameter in the constructor. This parameter is the character separator for the log events managed by an array. For instance, if we have a log that contains an array extra fields, this will be translated in 'extra-field', where '-' is the character separator (default) and field is the subname of the specific extra field.

161.3 Writing to FirePHP

`Zend\Log\Writer\FirePHP` writes log information to the FirePHP Firefox extension. In order to use this you have to install the FirePHPCore Server Library and the FirePHP browser extension.

To install the FirePHPCore Library you can use composer. Add the repository and the required line to your topmost composer.json:

161.4 Stubbing Out the Writer

The `Zend\Log\Writer\Null` is a stub that does not write log data to anything. It is useful for disabling logging or stubbing out logging during tests:

```
$writer = new Zend\Log\Writer\Null;
$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

// goes nowhere
$logger->info('Informational message');
```

161.5 Testing with the Mock

The `Zend\Log\Writer\Mock` is a very simple writer that records the raw data it receives in an array exposed as a public property.

```
$mock = new Zend\Log\Writer\Mock;
$logger = new Zend\Log\Logger();
$logger->addWriter($mock);

$logger->info('Informational message');

var_dump($mock->events[0]);

// Array
// (
//     [timestamp] => 2007-04-06T07:16:37-07:00
//     [message] => Informational message
//     [priority] => 6
//     [priorityName] => INFO
// )
```

To clear the events logged by the mock, simply set `$mock->events = array()`.

161.6 Compositing Writers

There is no composite Writer object. However, a Log instance can write to any number of Writers. To do this, use the `addWriter()` method:

```
$writer1 = new Zend\Log\Writer\Stream('/path/to/first/logfile');
$writer2 = new Zend\Log\Writer\Stream('/path/to/second/logfile');

$logger = new Zend\Log\Logger();
$logger->addWriter($writer1);
$logger->addWriter($writer2);

// goes to both writers
$logger->info('Informational message');
```

You can also specify the priority number for each writer to change the order of writing. The priority number is an integer number (greater or equal to 1) passed as second parameter in the `addWriter()` method.

CHAPTER

TWO

FILTERS

A Filter object blocks a message from being written to the log.

You can add a filter to a specific Writer using `addFilter()` method of that Writer:

```
use Zend\Log\Logger;

$logger = new Logger();

$writer1 = new Zend\Log\Writer\Stream('/path/to/first/logfile');
$logger->addWriter($writer1);

$writer2 = new Zend\Log\Writer\Stream('/path/to/second/logfile');
$logger->addWriter($writer2);

// add a filter only to writer2
$filter = new Zend\Log\Filter\Priority(Logger::CRIT);
$writer2->addFilter($filter);

// logged to writer1, blocked from writer2
$logger->info('Informational message');

// logged by both writers
$logger->emerg('Emergency message');
```

162.1 Available filters

The Zend\Log\Filter available are:

- **Priority**, filter logging by $priority. By default, it will accept any log event whose priority value is less than or equal to $priority.
- **Regex**, filter out any log messages not matching the regex pattern. This filter use the preg_match() function of PHP.
- **SuppressFilter**, this is a simple boolean filter. Call suppress(true) to suppress all log events. Call suppress(false) to accept all log events.
- **Validator**, filter out any log messages not matching the Zend\Validator\Validator object passed to the filter.

CHAPTER

THREE

FORMATTERS

A Formatter is an object that is responsible for taking an `event` array describing a log event and outputting a string with a formatted log line.

Some Writers are not line-oriented and cannot use a Formatter. An example is the Database Writer, which inserts the event items directly into database columns. For Writers that cannot support a Formatter, an exception is thrown if you attempt to set a Formatter.

163.1 Simple Formatting

`Zend\Log\Formatter\Simple` is the default formatter. It is configured automatically when you specify no formatter. The default configuration is equivalent to the following:

```
$format = '%timestamp% %priorityName% (%priority%): %message%' . PHP_EOL;
$formatter = new Zend\Log\Formatter\Simple($format);
```

A formatter is set on an individual Writer object using the Writer's `setFormatter()` method:

```
$writer = new Zend\Log\Writer\Stream('php://output');
$formatter = new Zend\Log\Formatter\Simple('hello %message%' . PHP_EOL);
$writer->setFormatter($formatter);

$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('there');

// outputs "hello there"
```

The constructor of `Zend\Log\Formatter\Simple` accepts a single parameter: the format string. This string contains keys surrounded by percent signs (e.g. `%message%`). The format string may contain any key from the event data array. You can retrieve the default keys by using the DEFAULT_FORMAT constant from `Zend\Log\Formatter\Simple`.

163.2 Formatting to XML

`Zend\Log\Formatter\Xml` formats log data into *XML* strings. By default, it automatically logs all items in the event data array:

```
$writer = new Zend\Log\Writer\Stream('php://output');
$formatter = new Zend\Log\Formatter\Xml();
$writer->setFormatter($formatter);

$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above outputs the following *XML* (space added for clarity):

```
<logEntry>
  <timestamp>2007-04-06T07:24:37-07:00</timestamp>
  <message>informational message</message>
  <priority>6</priority>
  <priorityName>INFO</priorityName>
</logEntry>
```

It's possible to customize the root element as well as specify a mapping of *XML* elements to the items in the event data array. The constructor of `Zend\Log\Formatter\Xml` accepts a string with the name of the root element as the first parameter and an associative array with the element mapping as the second parameter:

```
$writer = new Zend\Log\Writer\Stream('php://output');
$formatter = new Zend\Log\Formatter\Xml('log',
                                        array('msg' => 'message',
                                              'level' => 'priorityName')
                                       );
$writer->setFormatter($formatter);

$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above changes the root element from its default of `logEntry` to `log`. It also maps the element `msg` to the event data item `message`. This results in the following output:

```
<log>
  <msg>informational message</msg>
  <level>INFO</level>
</log>
```

163.3 Formatting to FirePhp

`Zend\Log\Formatter\FirePhp` formats log data for the Firebug extension for Firefox.

CHAPTER

FOUR

INTRODUCTION TO ZEND\MAIL

164.1 Getting started

`Zend\Mail` provides generalized functionality to compose and send both text and *MIME*-compliant multipart email messages. Mail can be sent with `Zend\Mail` via the `Mail\Transport\Sendmail`, `Mail\Transport\Smtp` or the `Mail\Transport\File` transport. Of course, you can also implement your own transport by implementing the `Mail\Transport\TransportInterface`.

Simple email with ZendMail

A simple email consists of one or more recipients, a subject, a body and a sender. To send such a mail using `Zend\Mail\Transport\Sendmail`, do the following:

```
use Zend\Mail;

$mail = new Mail\Message();
$mail->setBody('This is the text of the email.');
$mail->setFrom('Freeaqingme@example.org', 'Sender\'s name');
$mail->addTo('Matthew@example.com', 'Name o. recipient');
$mail->setSubject('TestSubject');

$transport = new Mail\Transport\Sendmail();
$transport->send($mail);
```

Note: Minimum definitions

In order to send an email using `Zend\Mail` you have to specify at least one recipient as well as a message body. Please note that each Transport may require additional parameters to be set.

For most mail attributes there are "get" methods to read the information stored in the message object. for further details, please refer to the *API* documentation.

You also can use most methods of the `Mail\Message` object with a convenient fluent interface.

```
use Zend\Mail;

$mail = new Mail\Message();
$mail->setBody('This is the text of the mail.')
     ->setFrom('somebody@example.com', 'Some Sender')
     ->addTo('somebody_else@example.com', 'Some Recipient')
     ->setSubject('TestSubject');
```

164.2 Configuring the default sendmail transport

The most simple to use transport is the `Mail\Transport\Sendmail` transport class. It is essentially a wrapper to the *PHP* mail() function. If you wish to pass additional parameters to the mail() function, simply create a new transport instance and pass your parameters to the constructor.

Passing additional parameters to the `Zend\Mail\Transport\Sendmail` transport.

This example shows how to change the Return-Path of the mail() function.

```
use Zend\Mail;

$mail = new Mail\Message();
$mail->setBody('This is the text of the email.');
$mail->setFrom('Freeaqingme@example.org', 'Dolf');
$mail->addTo('matthew@example.com', 'Matthew');
$mail->setSubject('TestSubject');

$transport = new Mail\Transport\Sendmail('-freturn_to_me@example.com');
$transport->send($mail);
```

Note: Safe mode restrictions

Supplying additional parameters to the transport will cause the mail() function to fail if *PHP* is running in safe mode.

Note: Choosing your transport wisely

Although the sendmail transport is the transport that requires only minimal configuration, it may not be suitable for your production environment. This is because emails sent using the sendmail transport will be more often delivered to SPAM-boxes. This can partly be remedied by using the *SMTP Transport* combined with an SMTP server that has an overall good reputation. Additionally, techniques such as SPF and DKIM may be employed to ensure even more email messages are delivered as should.

Warning: Sendmail Transport and Windows
As the *PHP* manual states the `mail()` function has different behaviour on Windows and on *nix based systems. Using the Sendmail Transport on Windows will not work in combination with `addBcc()`. The `mail()` function will sent to the BCC recipient such that all the other recipients can see him as recipient!
Therefore if you want to use BCC on a windows server, use the SMTP transport for sending!

CHAPTER

FIVE

ZEND\MAIL\MESSAGE

165.1 Overview

The `Message` class encapsulates a single email message as described in RFCs 822 and 2822. It acts basically as a value object for setting mail headers and content.

If desired, multi-part email messages may also be created. This is as trivial as creating the message body using the *Zend\Mime* component, assigning it to the mail message body.

The `Message` class is simply a value object. It is not capable of sending or storing itself; for those purposes, you will need to use, respectively, a *Transport adapter* or *Storage adapter*.

165.2 Quick Start

Creating a `Message` is simple: simply instantiate it.

```
use Zend\Mail\Message;

$message = new Message();
```

Once you have your `Message` instance, you can start adding content or headers. Let's set who the mail is from, who it's addressed to, a subject, and some content:

```
$message->addFrom("matthew@zend.com", "Matthew Weier O'Phinney")
        ->addTo("foobar@example.com")
        ->setSubject("Sending an email from Zend\Mail!");
$message->setBody("This is the message body.");
```

You can also add recipients to carbon-copy ("Cc:") or blind carbon-copy ("Bcc:").

```
$message->addCc("ralph.schindler@zend.com")
        ->addBcc("enrico.z@zend.com");
```

If you want to specify an alternate address to which replies may be sent, that can be done, too.

```
$message->addReplyTo("matthew@weierophinney.net", "Matthew");
```

Interestingly, RFC822 allows for multiple "From:" addresses. When you do this, the first one will be used as the sender, **unless** you specify a "Sender:" header. The `Message` class allows for this.

```
/*
 * Mail headers created:
 * From: Ralph Schindler <ralph.schindler@zend.com>, Enrico Zimuel <enrico.z@zend.com>
```

```
 * Sender: Matthew Weier O'Phinney <matthew@zend.com></matthew>
 */
$message->addFrom("ralph.schindler@zend.com", "Ralph Schindler")
        ->addFrom("enrico.z@zend.com", "Enrico Zimuel")
        ->setSender("matthew@zend.com", "Matthew Weier O'Phinney");
```

By default, the `Message` class assumes ASCII encoding for your email. If you wish to use another encoding, you can do so; setting this will ensure all headers and body content are properly encoded using quoted-printable encoding.

```
$message->setEncoding("UTF-8");
```

If you wish to set other headers, you can do that as well.

```
/*
 * Mail headers created:
 * X-API-Key: FOO-BAR-BAZ-BAT
 */
$message->getHeaders()->addHeaderLine('X-API-Key', 'FOO-BAR-BAZ-BAT');
```

Sometimes you may want to provide HTML content, or multi-part content. To do that, you'll first create a MIME message object, and then set it as the body of your mail message object. When you do so, the `Message` class will automatically set a "MIME-Version" header, as well as an appropriate "Content-Type" header.

```
use Zend\Mail\Message;
use Zend\Mime\Message as MimeMessage;
use Zend\Mime\Part as MimePart;

$text = new MimePart($textContent);
$text->type = "text/plain";

$html = new MimePart($htmlMarkup);
$html->type = "text/html";

$image = new MimePart(fopen($pathToImage, 'r'));
$image->type = "image/jpeg";

$body = new MimeMessage();
$body->setParts(array($text, $html, $image));

$message = new Message();
$message->setBody($body);
```

If you want a string representation of your email, you can get that:

```
echo $message->toString();
```

Finally, you can fully introspect the message – including getting all addresses of recipients and senders, all headers, and the message body.

```
// Headers
// Note: this will also grab all headers for which accessors/mutators exist in
// the Message object itself.
foreach ($message->getHeaders() as $header) {
    echo $header->toString();
    // or grab values: $header->getFieldName(), $header->getFieldValue()
}

// The logic below also works for the methods cc(), bcc(), to(), and replyTo()
foreach ($message->from() as $address) {
```

```
    printf("%s: %s\n", $address->getEmail(), $address->getName());
}

// Sender
$address = $message->getSender();
printf("%s: %s\n", $address->getEmail(), $address->getName());

// Subject
echo "Subject: ", $message->getSubject(), "\n";

// Encoding
echo "Encoding: ", $message->getEncoding(), "\n";

// Message body:
echo $message->getBody();      // raw body, or MIME object
echo $message->getBodyText(); // body as it will be sent
```

Once your message is shaped to your liking, pass it to a *mail transport* in order to send it!

```
$transport->send($message);
```

165.3 Configuration Options

The `Message` class has no configuration options, and is instead a value object.

165.4 Available Methods

isValid `isValid()`

Is the message valid?

If we don't have any From addresses, we're invalid, according to RFC2822.

Returns bool

setEncoding `setEncoding(string $encoding)`

Set the message encoding.

Implements a fluent interface.

getEncoding `getEncoding()`

Get the message encoding.

Returns string.

setHeaders `setHeaders(Zend\Mail\Headers $headers)`

Compose headers.

Implements a fluent interface.

getHeaders `getHeaders()`

Access headers collection.

Lazy-loads a Zend\Mail\Headers instance if none is already attached.

Returns a Zend\Mail\Headers instance.

setFrom `setFrom(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, string|null $name)`

Set (overwrite) From addresses.

Implements a fluent interface.

addFrom `addFrom(string|Zend\Mail\Address|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, string|null $name)`

Add a "From" address.

Implements a fluent interface.

from `from()`

Retrieve list of From senders

Returns Zend\Mail\AddressList instance.

setTo `setTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, null|string $name)`

Overwrite the address list in the To recipients.

Implements a fluent interface.

addTo `addTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, null|string $name)`

Add one or more addresses to the To recipients.

Appends to the list.

Implements a fluent interface.

to `to()`

Access the address list of the To header.

Lazy-loads a Zend\Mail\AddressList and populates the To header if not previously done.

Returns a Zend\Mail\AddressList instance.

setCc `setCc(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, string|null $name)`

Set (overwrite) CC addresses.

Implements a fluent interface.

addCc `addCc(string|Zend\Mail\Address|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, string|null $name)`

Add a "Cc" address.

Implements a fluent interface.

cc `cc()`

Retrieve list of CC recipients

Lazy-loads a Zend\Mail\AddressList and populates the Cc header if not previously done.

Returns a Zend\Mail\AddressList instance.

setBcc `setBcc(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, string|null $name)`

Set (overwrite) BCC addresses.

Implements a fluent interface.

addBcc `addBcc(string|Zend\Mail\Address|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, string|null $name)`

Add a "Bcc" address.

Implements a fluent interface.

bcc `bcc()`

Retrieve list of BCC recipients.

Lazy-loads a Zend\Mail\AddressList and populates the Bcc header if not previously done.

Returns a Zend\Mail\AddressList instance.

setReplyTo `setReplyTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, null|string $name)`

Overwrite the address list in the Reply-To recipients.

Implements a fluent interface.

addReplyTo `addReplyTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, null|string $name)`

Add one or more addresses to the Reply-To recipients.

Implements a fluent interface.

replyTo `replyTo()`

Access the address list of the Reply-To header

Lazy-loads a Zend\Mail\AddressList and populates the Reply-To header if not previously done.

Returns a Zend\Mail\AddressList instance.

setSender `setSender(mixed $emailOrAddress, mixed $name)`

Set the message envelope Sender header.

Implements a fluent interface.

getSender `getSender()`

Retrieve the sender address, if any.

Returns null or a Zend\Mail\AddressDescription instance.

setSubject `setSubject(string $subject)`

Set the message subject header value.

Implements a fluent interface.

getSubject `getSubject()`

Get the message subject header value.

Returns null or a string.

setBody `setBody(null|string|Zend\Mime\Message|object $body)`

Set the message body.

Implements a fluent interface.

getBody `getBody()`

Return the currently set message body.

Returns null, a string, or an object.

getBodyText `getBodyText()`

Get the string-serialized message body text.

Returns null or a string.

toString `toString()`

Serialize to string.

Returns string.

165.5 Examples

Please *see the Quick Start section*.

CHAPTER

SIX

ZEND\MAIL\TRANSPORT

166.1 Overview

Transports take care of the actual delivery of mail. Typically, you only need to worry about two possibilities: using PHP's native `mail()` functionality, which uses system resources to deliver mail, or using the *SMTP* protocol for delivering mail via a remote server. Zend Framework also includes a "File" transport, which creates a mail file for each message sent; these can later be introspected as logs or consumed for the purposes of sending via an alternate transport mechanism later.

The `Zend\Mail\Transport` interface defines exactly one method, `send()`. This method accepts a `Zend\Mail\Message` instance, which it then introspects and serializes in order to send.

166.2 Quick Start

Using a mail transport is typically as simple as instantiating it, optionally configuring it, and then passing a message to it.

Sendmail Transport Usage

```
use Zend\Mail\Message;
use Zend\Mail\Transport\Sendmail as SendmailTransport;

$message = new Message();
$message->addTo('matthew@zend.com')
        ->addFrom('ralph.schindler@zend.com')
        ->setSubject('Greetings and Salutations!')
        ->setBody("Sorry, I'm going to be late today!");

$transport = new SendmailTransport();
$transport->send($message);
```

SMTP Transport Usage

```
use Zend\Mail\Message;
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

$message = new Message();
```

```
$message->addTo('matthew@zend.com')
        ->addFrom('ralph.schindler@zend.com')
        ->setSubject('Greetings and Salutations!')
        ->setBody("Sorry, I'm going to be late today!");

// Setup SMTP transport using LOGIN authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'login',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
$transport->send($message);
```

File Transport Usage

```
use Zend\Mail\Message;
use Zend\Mail\Transport\File as FileTransport;
use Zend\Mail\Transport\FileOptions;

$message = new Message();
$message->addTo('matthew@zend.com')
        ->addFrom('ralph.schindler@zend.com')
        ->setSubject('Greetings and Salutations!')
        ->setBody("Sorry, I'm going to be late today!");

// Setup SMTP transport using LOGIN authentication
$transport = new FileTransport();
$options   = new FileOptions(array(
    'path'              => 'data/mail/',
    'callback'  => function (FileTransport $transport) {
        return 'Message_' . microtime(true) . '_' . mt_rand() . '.txt';
    },
));
$transport->setOptions($options);
$transport->send($message);
```

166.3 Configuration Options

Configuration options are per transport. Please follow the links below for transport-specific options.

- *SMTP Transport Options*
- *File Transport Options*

166.4 Available Methods

send `send(Zend\Mail\Message $message)`

Send a mail message.

Returns void

166.5 Examples

Please see the *Quick Start section* for examples.

CHAPTER

SEVEN

ZEND\MAIL\TRANSPORT\SMTPOPTIONS

167.1 Overview

This document details the various options available to the `Zend\Mail\Transport\Smtp` mail transport.

167.2 Quick Start

Basic SMTP Transport Usage

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name' => 'localhost.localdomain',
    'host' => '127.0.0.1',
    'port' => 25,
));
$transport->setOptions($options);
```

SMTP Transport Usage with PLAIN AUTH

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport using PLAIN authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'plain',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
```

SMTP Transport Usage with LOGIN AUTH

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport using LOGIN authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'login',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
```

SMTP Transport Usage with CRAM-MD5 AUTH

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport using CRAM-MD5 authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'crammd5',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
```

SMTP Transport Usage with PLAIN AUTH over TLS

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport using PLAIN authentication over TLS
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'example.com',
    'host'              => '127.0.0.1',
    'port'              => 587, // Notice port change for TLS is 587
    'connection_class'  => 'plain',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
        'ssl'      => 'tls',
    ),
));
$transport->setOptions($options);
```

167.3 Configuration Options

Configuration Options

name Name of the SMTP host; defaults to "localhost".

host Remote hostname or IP address; defaults to "127.0.0.1".

port Port on which the remote host is listening; defaults to "25".

connection_class Fully-qualified classname or short name resolvable via `Zend\Mail\Protocol\SmtpLoader`. Typically, this will be one of "smtp", "plain", "login", or "crammd5", and defaults to "smtp".

Typically, the connection class should extend the `Zend\Mail\Protocol\AbstractProtocol` class, and specifically the SMTP variant.

connection_config Optional associative array of parameters to pass to the *connection class* in order to configure it. By default this is empty. For connection classes other than the default, you will typically need to define the "username" and "password" options. For secure connections you will use the "ssl" => "tls" and port 587 for TLS or "ssl" => "ssl" and port 465 for SSL.

167.4 Available Methods

getName `getName()`

Returns the string name of the local client hostname.

setName `setName(string $name)`

Set the string name of the local client hostname.

Implements a fluent interface.

getConnectionClass `getConnectionClass()`

Returns a string indicating the connection class name to use.

setConnectionClass `setConnectionClass(string $connectionClass)`

Set the connection class to use.

Implements a fluent interface.

getConnectionConfig `getConnectionConfig()`

Get configuration for the connection class.

Returns array.

setConnectionConfig `setConnectionConfig(array $config)`

Set configuration for the connection class. Typically, if using anything other than the default connection class, this will be an associative array with the keys "username" and "password".

Implements a fluent interface.

getHost `getHost()`

Returns a string indicating the IP address or host name of the SMTP server via which to send messages.

setHost `setHost(string $host)`

Set the SMTP host name or IP address.

Implements a fluent interface.

getPort `getPort()`

Retrieve the integer port on which the SMTP host is listening.

setPort `setPort(int $port)`

Set the port on which the SMTP host is listening.

Implements a fluent interface.

__construct `__construct(null|array|Traversable $config)`

Instantiate the class, and optionally configure it with values provided.

167.5 Examples

Please see the *Quick Start* for examples.

CHAPTER

EIGHT

ZEND\MAIL\TRANSPORT\FILEOPTIONS

168.1 Overview

This document details the various options available to the `Zend\Mail\Transport\File` mail transport.

168.2 Quick Start

File Transport Usage

```
use Zend\Mail\Transport\File as FileTransport;
use Zend\Mail\Transport\FileOptions;

// Setup File transport
$transport = new FileTransport();
$options   = new FileOptions(array(
    'path'              => 'data/mail/',
    'callback'  => function (FileTransport $transport) {
        return 'Message_' . microtime(true) . '_' . mt_rand() . '.txt';
    },
));
$transport->setOptions($options);
```

168.3 Configuration Options

Configuration Options

path The path under which mail files will be written.

callback A PHP callable to be invoked in order to generate a unique name for a message file. By default, the following is used:

```
function (Zend\Mail\FileTransport $transport) {
    return 'ZendMail_' . time() . '_' . mt_rand() . '.tmp';
}
```

168.4 Available Methods

`Zend\Mail\Transport\FileOptions` extends `Zend\Stdlib\Options`, and inherits all functionality from that class; this includes `ArrayAccess` and property overloading. Additionally, the following explicit setters and getters are provided.

__construct `setPath(string $path)`

Set the path under which mail files will be written.

Implements fluent interface.

getPath `getPath()`

Get the path under which mail files will be written.

Returns string

setCallback `setCallback(Callable $callback)`

Set the callback used to generate unique filenames for messages.

Implements fluent interface.

getCallback `getCallback()`

Get the callback used to generate unique filenames for messages.

Returns PHP callable argument.

__construct `__construct(null|array|Traversable $config)`

Initialize the object. Allows passing a PHP array or `Traversable` object with which to populate the instance.

168.5 Examples

Please see the *Quick Start* for examples.

CHAPTER

NINE

INTRODUCTION TO ZEND\MATH

`Zend\Math` namespace provides general mathematical functions. So far the supported functionalities are:

- `Zend\Math\Rand`, a random number generator;
- `Zend\Math\BigInteger`, a library to manage big integers.

We expect to add more functionalities in the future.

169.1 Random number generator

`Zend\Math\Rand` implements a random number generator that is able to generate random numbers for general purpose usage and for cryptographic scopes. To generate good random numbers this component uses the OpenSSL and the Mcrypt extension of PHP. If you don't have the OpenSSL or the Mcrypt extension installed in your environment the component will use the mt_rand function of PHP as fallback. The `mt_rand` is not considered secure for cryptographic purpose, that means if you will try to use it to generate secure random number the class will throw an exception.

In particular, the algorithm that generates random bytes in `Zend\Math\Rand` tries to call the openssl_random_pseudo_bytes function of the OpenSSL extension if installed. If the OpenSSL extension is not present in the system the algorithm tries to use the the mcrypt_create_iv function of the Mcrypt extension (using the `MCRYPT_DEV_URANDOM` parameter). Finally, if the OpenSSL and Mcrypt are not installed the generator uses the `mt_rand` function of PHP.

The `Zend\Math\Rand` class offers the following methods to generate random values:

- `getBytes($length, $strong = false)` to generate a random set of `$length` bytes;
- `getBoolean($strong = false)` to generate a random boolean value (true or false);
- `getInteger($min, $max, $strong = false)` to generate a random integer between `$min` and `$max`;
- `getFloat($strong = false)` to generate a random float number between 0 and 1;
- `getString($length, $charlist = null, $strong = false)` to generate a random string of $length characters using the alphabet $charlist (if not provided the default alphabet is the Base64).

In all these methods the parameter `$strong` specify the usage of a strong random number generator. We suggest to set the $strong to true if you need to generate random number for cryptographic and security implementation.

If `$strong` is set to true and you try to generate random values in a PHP environment without the OpenSSL and the Mcrypt extensions the component will throw an Exception.

Below we reported an example on how to generate random data using `Zend\Math\Rand`.

```
use Zend\Math\Rand;

$bytes = Rand::getBytes(32, true);
printf("Random bytes (in Base64): %s\n", base64_encode($bytes));

$boolean = Rand::getBoolean();
printf("Random boolean: %s\n", $boolean ? 'true' : 'false');

$integer = Rand::getInteger(0,1000);
printf("Random integer in [0-1000]: %d\n", $integer);

$float = Rand::getFloat();
printf("Random float in [0-1): %f\n", $float);

$string = Rand::getString(32, 'abcdefghijklmnopqrstuvwxyz', true);
printf("Random string in latin alphabet: %s\n", $string);
```

169.2 Big integers

`Zend\Math\BigInteger\BigInteger` offers a class to manage arbitrary length integer. PHP supports integer numbers with a maximum value of `PHP_INT_MAX`. If you need to manage integers bigger than `PHP_INT_MAX` you have to use external libraries or PHP extensions like GMP or BC Math.

`Zend\Math\BigInteger\BigInteger` is able to manage big integers using the GMP or the BC Math extensions as adapters.

The mathematical functions implemented in `Zend\Math\BigInteger\BigInteger` are:

- `add($leftOperand, $rightOperand)`, add two big integers;
- `sub($leftOperand, $rightOperand)`, subtract two big integers;
- `mul($leftOperand, $rightOperand)`, multiply two big integers;
- `div($leftOperand, $rightOperand)`, divide two big integers (this method returns only integer part of result);
- `pow($operand, $exp)`, raise a big integers to another;
- `sqrt($operand)`, get the square root of a big integer;
- `abs($operand)`, get the absolute value of a big integer;
- `mod($leftOperand, $modulus)`, get modulus of a big integer;
- `powmod($leftOperand, $rightOperand, $modulus)`, raise a big integer to another, reduced by a specified modulus;
- `comp($leftOperand, $rightOperand)`, compare two big integers, returns < 0 if leftOperand is less than rightOperand; > 0 if leftOperand is greater than rightOperand, and 0 if they are equal;
- `intToBin($int, $twoc = false)`, convert big integer into it's binary number representation;
- `binToInt($bytes, $twoc = false)`, convert binary number into big integer;
- `baseConvert($operand, $fromBase, $toBase = 10)`, convert a number between arbitrary bases;

Below is reported an example using the BC Math adapter to calculate the sum of two integer random numbers with 100 digits.

```
use Zend\Math\BigInteger\BigInteger;
use Zend\Math\Rand;

$bigInt = BigInteger::factory('bcmath');

$x = Rand::getString(100,'0123456789');
$y = Rand::getString(100,'0123456789');

$sum = $bigInt->add($x, $y);
$len = strlen($sum);

printf("%{$len}s +\n%{$len}s =\n%s\n%s\n", $x, $y, str_repeat('-', $len), $sum);
```

As you can see in the code the big integers are managed using strings. Even the result of the sum is represented as a string.

Below is reported another example using the BC Math adapter to generate the binary representation of a negative big integer of 100 digits.

```
use Zend\Math\BigInteger\BigInteger;
use Zend\Math\Rand;

$bigInt = BigInteger::factory('bcmath');

$digit = 100;
$x = '-' . Rand::getString($digit,'0123456789');

$byte = $bigInt->intToBin($x);

printf("The binary representation of the big integer with $digit digit:\n%s\nis (in Base64 format): %
       $x, base64_encode($byte));
printf("Length in bytes: %d\n", strlen($byte));

$byte = $bigInt->intToBin($x, true);

printf("The two's complement binary representation of the big integer with $digit digit:\n%s\nis (in
       $x, base64_encode($byte));
printf("Length in bytes: %d\n", strlen($byte));
```

We generated the binary representation of the big integer number using the default binary format and the two's complement representation (specified with the `true` parameter in the `intToBin` function).

CHAPTER

ZEND\MIME

170.1 Introduction

`Zend\Mime\Mime` is a support class for handling multipart *MIME* messages. It is used by *Zend\Mail* and *Zend\Mime\Message* and may be used by applications requiring *MIME* support.

170.2 Static Methods and Constants

`Zend\Mime\Mime` provides a simple set of static helper methods to work with *MIME*:

- `Zend\Mime\Mime::isPrintable()`: Returns `TRUE` if the given string contains no unprintable characters, `FALSE` otherwise.
- `Zend\Mime\Mime::encode()`: Encodes a string with specified encoding.
- `Zend\Mime\Mime::encodeBase64()`: Encodes a string into base64 encoding.
- `Zend\Mime\Mime::encodeQuotedPrintable()`: Encodes a string with the quoted-printable mechanism.
- `Zend\Mime\Mime::encodeBase64Header()`: Encodes a string into base64 encoding for Mail Headers.
- `Zend\Mime\Mime::encodeQuotedPrintableHeader()`: Encodes a string with the quoted-printable mechanism for Mail Headers.

`Zend\Mime\Mime` defines a set of constants commonly used with *MIME* messages:

- `Zend\Mime\Mime::TYPE_OCTETSTREAM`: 'application/octet-stream'
- `Zend\Mime\Mime::TYPE_TEXT`: 'text/plain'
- `Zend\Mime\Mime::TYPE_HTML`: 'text/html'
- `Zend\Mime\Mime::ENCODING_7BIT`: '7bit'
- `Zend\Mime\Mime::ENCODING_8BIT`: '8bit'
- `Zend\Mime\Mime::ENCODING_QUOTEDPRINTABLE`: 'quoted-printable'
- `Zend\Mime\Mime::ENCODING_BASE64`: 'base64'
- `Zend\Mime\Mime::DISPOSITION_ATTACHMENT`: 'attachment'
- `Zend\Mime\Mime::DISPOSITION_INLINE`: 'inline'
- `Zend\Mime\Mime::MULTIPART_ALTERNATIVE`: 'multipart/alternative'
- `Zend\Mime\Mime::MULTIPART_MIXED`: 'multipart/mixed'
- `Zend\Mime\Mime::MULTIPART_RELATED`: 'multipart/related'

170.3 Instantiating Zend\Mime

When instantiating a `Zend\Mime\Mime` object, a *MIME* boundary is stored that is used for all subsequent non-static method calls on that object. If the constructor is called with a string parameter, this value is used as a *MIME* boundary. If not, a random *MIME* boundary is generated during construction time.

A `Zend\Mime\Mime` object has the following methods:

- `boundary()`: Returns the *MIME* boundary string.
- `boundaryLine()`: Returns the complete *MIME* boundary line.
- `mimeEnd()`: Returns the complete *MIME* end boundary line.

CHAPTER

ONE

ZEND\MIME\MESSAGE

171.1 Introduction

`Zend\Mime\Message` represents a *MIME* compliant message that can contain one or more separate Parts (Represented as *Zend\Mime\Part* objects). With `Zend\Mime\Message`, *MIME* compliant multipart messages can be generated from `Zend\Mime\Part` objects. Encoding and Boundary handling are handled transparently by the class. `Zend\Mime\Message` objects can also be reconstructed from given strings. Used by *Zend\Mail*.

171.2 Instantiation

There is no explicit constructor for `Zend\Mime\Message`.

171.3 Adding MIME Parts

Zend\Mime\Part Objects can be added to a given `Zend\Mime\Message` object by calling `->addPart($part)`

An array with all *Zend\Mime\Part* objects in the `Zend\Mime\Message` is returned from the method `getParts()`. The `Zend\Mime\Part` objects can then be changed since they are stored in the array as references. If parts are added to the array or the sequence is changed, the array needs to be given back to the *Zend\Mime\Part* object by calling `setParts($partsArray)`.

The function `isMultiPart()` will return `TRUE` if more than one part is registered with the `Zend\Mime\Message` object and thus the object would generate a Multipart-Mime-Message when generating the actual output.

171.4 Boundary handling

`Zend\Mime\Message` usually creates and uses its own `Zend\Mime\Mime` Object to generate a boundary. If you need to define the boundary or want to change the behaviour of the `Zend\Mime\Mime` object used by `Zend\Mime\Message`, you can instantiate the `Zend\Mime\Mime` class yourself and then register it to `Zend\Mime\Message`. Usually you will not need to do this. `setMime(Zend\Mime\Mime $mime)` sets a special instance of `Zend\Mime\Mime` to be used by this `Zend\Mime\Message`.

`getMime()` returns the instance of `Zend\Mime\Mime` that will be used to render the message when `generateMessage()` is called.

`generateMessage()` renders the `Zend\Mime\Message` content to a string.

171.5 Parsing a string to create a Zend\Mime\Message object

A given *MIME* compliant message in string form can be used to reconstruct a `Zend\Mime\Message` object from it. `Zend\Mime\Message` has a static factory Method to parse this String and return a `Zend\Mime\Message` object.

`Zend\Mime\Message::createFromMessage($str, $boundary)` decodes the given string and returns a `Zend\Mime\Message` object that can then be examined using `getParts()`

171.6 Available methods

A `Zend\Mime\Message` object has the following methods:

- `getParts`: Get the all `Zend\Mime\Parts` in the message.
- `setParts($parts)`: Set the array of `Zend\Mime\Parts` for the message.
- `addPart(Zend\Mime\Part $part)`: Append a new `Zend\Mime\Part` to the message.
- `isMultiPart`: Check if the message needs to be sent as a multipart *MIME* message.
- `setMime(Zend\Mime\Mime $mime)`: Set a custom `Zend\Mime\Mime` object for the message.
- `getMime`: Get the `Zend\Mime\Mime` object for the message.
- `generateMessage($EOL=Zend\Mime\Mime::LINEEND)`: Generate a *MIME*-compliant message from the current configuration.
- `getPartHeadersArray($partnum)`: Get the headers of a given part as an array.
- `getPartHeaders($partnum,$EOL=Zend\Mime\Mime::LINEEND)`: Get the headers of a given part as a string.
- `getPartContent($partnum,$EOL=Zend\Mime\Mime::LINEEND)`: Get the encoded content of a given part as a string.

CHAPTER

TWO

ZEND\MIME\PART

172.1 Introduction

This class represents a single part of a *MIME* message. It contains the actual content of the message part plus information about its encoding, content type and original filename. It provides a method for generating a string from the stored data. `Zend\Mime\Part` objects can be added to *Zend\Mime\Message* to assemble a complete multipart message.

172.2 Instantiation

`Zend\Mime\Part` is instantiated with a string that represents the content of the new part. The type is assumed to be OCTET-STREAM, encoding is 8Bit. After instantiating a `Zend\Mime\Part`, meta information can be set by accessing its attributes directly:

```
public $type = Zend\Mime\Mime::TYPE_OCTETSTREAM;
public $encoding = Zend\Mime\Mime::ENCODING_8BIT;
public $id;
public $disposition;
public $filename;
public $description;
public $charset;
public $boundary;
public $location;
public $language;
```

172.3 Methods for rendering the message part to a string

`getContent()` returns the encoded content of the `Zend\Mime\Part` as a string using the encoding specified in the attribute `$encoding`. Valid values are ZendMimeMime::ENCODING_*. Characterset conversions are not performed.

`getHeaders()` returns the Mime-Headers for the `Zend\Mime\Part` as generated from the information in the publicly accessible attributes. The attributes of the object need to be set correctly before this method is called.

- `$charset` has to be set to the actual charset of the content if it is a text type (Text or *HTML*).
- `$id` may be set to identify a content-id for inline images in a *HTML* mail.
- `$filename` contains the name the file will get when downloading it.

- `$disposition` defines if the file should be treated as an attachment or if it is used inside the (HTML-) mail (inline).
- `$description` is only used for informational purposes.
- `$boundary` defines string as boundary.
- `$location` can be used as resource *URI* that has relation to the content.
- `$language` defines languages in the content.

172.4 Available methods

A `Zend\Mime\Part` object has the following methods:

- `isStream`: Check if this `Zend\Mime\Part` can be read as a stream.
- `getEncodedStream`: If this `Zend\Mime\Part` was created with a stream, return a filtered stream for reading the content. Useful for large file attachments.
- `getContent($EOL=Zend\Mime\Mime::LINEEND)`: Get the content of the current `Zend\Mime\Part` in the given encoding.
- `getRawContent`: Get the raw, unencoded for the current `Zend\Mime\Part`.
- `getHeadersArray($EOL=Zend\Mime\Mime::LINEEND)`: Create and return the array of headers for the current `Zend\Mime\Part`.
- `getHeaders($EOL=Zend\Mime\Mime::LINEEND)`: Return the headers for the current `Zend\Mime\Part` as a string.

CHAPTER

THREE

INTRODUCTION TO THE MODULE SYSTEM

Zend Framework 2.0 introduces a new and powerful approach to modules. This new module system is designed with flexibility, simplicity, and re-usability in mind. A module may contain just about anything: PHP code, including MVC functionality; library code; view scripts; and/or public assets such as images, CSS, and JavaScript. The possibilities are endless.

Note: The module system in ZF2 has been designed to be a generic and powerful foundation from which developers and other projects can build their own module or plugin systems.

For a better understanding of the event-driven concepts behind the ZF2 module system, it may be helpful to read the *EventManager documentation*.

The module system is made up of the following:

- *The Module Autoloader* - `Zend\Loader\ModuleAutoloader` is a specialized autoloader that is responsible for the locating and loading of modules' `Module` classes from a variety of sources.
- *The Module Manager* - `Zend\ModuleManager\ModuleManager` simply takes an array of module names and fires a sequence of events for each one, allowing the behavior of the module system to be defined entirely by the listeners which are attached to the module manager.
- **ModuleManager Listeners** - Event listeners can be attached to the module manager's various events. These listeners can do everything from resolving and loading modules to performing complex initialization tasks and introspection into each returned module object.

Note: The name of a module in a typical Zend Framework 2 application is simply a PHP namespace and must follow all of the same rules for naming.

The recommended structure of a typical MVC-oriented ZF2 module is as follows:

```
module_root/
    Module.php
    autoload_classmap.php
    autoload_function.php
    autoload_register.php
    config/
        module.config.php
    public/
        images/
        css/
```

```
    js/
src/
    <module_namespace>/
        <code files>
test/
    phpunit.xml
    bootstrap.php
    <module_namespace>/
        <test code files>
view/
    <dir-named-after-module-namespace>/
        <dir-named-after-a-controller>/
            <.phtml files>
```

173.1 The autoload_*.php Files

The three `autoload_*.php` files are not required, but recommended. They provide the following:

- `autoload_classmap.php` should return an array classmap of class name/filename pairs (with the filenames resolved via the `__DIR__` magic constant).
- `autoload_function.php` should return a PHP callback that can be passed to `spl_autoload_register()`. Typically, this callback should utilize the map returned by `autoload_classmap.php`.
- `autoload_register.php` should register a PHP callback (typically that returned by `autoload_function.php` with `spl_autoload_register()`.

The purpose of these three files is to provide reasonable default mechanisms for autoloading the classes contained in the module, thus providing a trivial way to consume the module without requiring `Zend\ModuleManager` (e.g., for use outside a ZF2 application).

CHAPTER

FOUR

THE MODULE MANAGER

The module manager, `Zend\ModuleManager\ModuleManager`, is a very simple class which is responsible for iterating over an array of module names and triggering a sequence of events for each. Instantiation of module classes, initialization tasks, and configuration are all performed by attached event listeners.

174.1 Module Manager Events

Events triggered by `Zend\ModuleManager\ModuleManager`

loadModules This event is primarily used internally to help encapsulate the work of loading modules in event listeners, and allow the loadModules.post event to be more user-friendly. Internal listeners will attach to this event with a negative priority instead of loadModules.post so that users can safely assume things like config merging have been done once loadModules.post is triggered, without having to worry about priorities at all.

loadModule.resolve Triggered for each module that is to be loaded. The listener(s) to this event are responsible for taking a module name and resolving it to an instance of some class. The default module resolver shipped with ZF2 simply looks for the class `{modulename}\Module`, instantiating and returning it if it exists.

The name of the module may be retrieved by listeners using the `getModuleName()` method of the `Event` object; a listener should then take that name and resolve it to an object instance representing the given module. Multiple listeners can be attached to this event, and the module manager will trigger them in order of their priority until one returns an object. This allows you to attach additional listeners which have alternative methods of resolving modules from a given module name.

loadModule Once a module resolver listener has resolved the module name to an object, the module manager then triggers this event, passing the newly created object to all listeners.

loadModules.post This event is triggered by the module manager to allow any listeners to perform work after every module has finished loading. For example, the default configuration listener, `Zend\ModuleManager\Listener\ConfigListener` (covered later), attaches to this event to merge additional user-supplied configuration which is meant to override the default supplied configurations of installed modules.

174.2 Module Manager Listeners

By default, Zend Framework provides several useful module manager listeners.

Provided Module Manager Listeners

Zend\ModuleManager\Listener\DefaultListenerAggregate To help simplify the most common use case of the module manager, ZF2 provides this default aggregate listener. In most cases, this will be the only listener you will need to attach to use the module manager, as it will take care of properly attaching the requisite listeners (those listed below) for the module system to function properly.

Zend\ModuleManager\Listener\AutoloaderListener This listener checks each module to see if it has implemented `Zend\ModuleManager\Feature\AutoloaderProviderInterface` or simply defined the `getAutoloaderConfig()` method. If so, it calls the `getAutoloaderConfig()` method on the module class and passes the returned array to `Zend\Loader\AutoloaderFactory`.

Zend\ModuleManager\Listener\ModuleDependencyCheckerListener This listener checks each module to verify if all the modules it depends on were loaded. When a module class implements `Zend\ModuleManager\Feature\DependencyIndicatorInterface` or simply has a defined `getDependencyModules()` method, the listener will call `getDependencyModules()`. Each of the values returned by the method is checked against the loaded modules list: if one of the values is not in that list, a `Zend\ModuleManager\Exception\MissingDependencyModuleException` is be thrown.

Zend\ModuleManager\Listener\ConfigListener If a module class has a `getConfig()` method, or implements `Zend\ModuleManager\Feature\ConfigProviderInterface`, this listener will call it and merge the returned array (or `Traversable` object) into the main application configuration.

Zend\ModuleManager\Listener\InitTrigger If a module class either implements `Zend\ModuleManager\Feature\InitProviderInterface`, or simply defines an `init()` method, this listener will call `init()` and pass the current instance of `Zend\ModuleManager\ModuleManager` as the sole parameter.

Like the `OnBootstrapListener`, the `init()` method is called for **every** module implementing this feature, on **every** page request and should **only** be used for performing **lightweight** tasks such as registering event listeners.

Zend\ModuleManager\Listener\LocatorRegistrationListener If a module class implements `Zend\ModuleManager\Feature\LocatorRegisteredInterface`, this listener will inject the module class instance into the `ServiceManager` using the module class name as the service name. This allows you to later retrieve the module class from the `ServiceManager`.

Zend\ModuleManager\Listener\ModuleResolverListener This is the default module resolver. It attaches to the "loadModule.resolve" event and simply returns an instance of `{moduleName}\Module`.

Zend\ModuleManager\Listener\OnBootstrapListener If a module class implements `Zend\ModuleManager\Feature\BootstrapListenerInterface`, or simply defines an `onBootstrap()` method, this listener will register the `onBootstrap()` method with the `Zend\Mvc\Application` `bootstrap` event. This method will then be triggered during the `bootstrap` event (and passed an `MvcEvent` instance).

Like the `InitTrigger`, the `onBootstrap()` method is called for **every** module implementing this feature, on **every** page request, and should **only** be used for performing **lightweight** tasks such as registering event listeners.

Zend\ModuleManager\Listener\ServiceListener If a module class implements `Zend\ModuleManager\Feature\ServiceProviderInterface`, or simply defines an `getServiceConfig()` method, this listener will call that method and aggregate the return values for use in configuring the `ServiceManager`.

The `getServiceConfig()` method may return either an array of configuration compatible with `Zend\ServiceManager\Config`, an instance of that class, or the string name of a class that extends it. Values are merged and aggregated on completion, and then merged with any configuration

from the `ConfigListener` falling under the `service_manager` key. For more information, see the `ServiceManager` documentation.

Unlike the other listeners, this listener is not managed by the `DefaultListenerAggregate`; instead, it is created and instantiated within the `Zend\Mvc\Service\ModuleManagerFactory`, where it is injected with the current `ServiceManager` instance before being registered with the `ModuleManager` events.

Additionally, this listener manages a variety of plugin managers, including *view helpers*, *controllers*, and *controller plugins*. In each case, you may either specify configuration to define plugins, or provide configuration via a `Module` class. Configuration follows the same format as for the `ServiceManager`. The following table outlines the plugin managers that may be configured this way (including the `ServiceManager`), the configuration key to use, the `ModuleManager` feature interface to optionally implement (all interfaces specified live in the `Zend\ModuleManager\Feature` namespace) , and the module method to optionally define to provide configuration.

Plugin Manager	Config Key	Interface	Module Method
`Zend\ServiceManager\ServiceManager`	`service_manager`	`ServiceProviderInterface`	`getServiceConfig`
`Zend\View\HelperPluginManager`	`view_helpers`	`ViewHelperProviderInterface`	`getViewHelperConfig`
`Zend\Mvc\Controller\ControllerManager`	`controllers`	`ControllerProviderInterface`	`getControllerConfig`
`Zend\Mvc\Controller\PluginManager`	`controller_plugins`	`ControllerPluginProviderInterface`	`getControllerPluginConfig`

Configuration follows the examples in the *ServiceManager configuration section*. As a brief recap, the following configuration keys and values are allowed:

Config Key	Allowed values
`services`	service name/instance pairs (these should likely be defined only in `Module` classes)
`invokables`	service name/class name pairs of classes that may be invoked without constructor arguments
`factories`	service names pointing to factories. Factories may be any PHP callable, or a string class name of a class implementing `Zend\ServiceManager\FactoryInterface`, or of a class implementing the `__invoke` method (if a callable is used, it should be defined only in `Module` classes)
`abstract_factories`	array of either concrete instances of `Zend\ServiceManager\AbstractFactoryInterface`, or string class names of classes implementing that interface (if an instance is used, it should be defined only in `Module` classes)
`initializers`	array of PHP callables or string class names of classes implementing `Zend\ServiceManager\InitializerInterface` (if a callable is used, it should be defined only in `Module` classes)

When working with plugin managers, you will be passed the plugin manager instance to factories, abstract factories, and initializers. If you need access to the application services, you can use the `getServiceLocator()` method, as in the following example:

```
public function getViewHelperConfig()
{
    return array('factories' => array(
        'foo' => function ($helpers) {
            $services = $helpers->getServiceLocator();
            $someService = $services->get('SomeService');
            $helper = new Helper\Foo($someService);
            return $helper;
        },
    ));
}
```

This is a powerful technique, as it allows your various plugins to remain agnostic with regards to where and how dependencies are injected, and thus allows you to use Inversion of Control principals even with plugins.

CHAPTER

FIVE

THE MODULE CLASS

By default, the Zend Framework 2 module system simply expects each module name to be capable of resolving to an object instance. The default module resolver, `Zend\ModuleManager\Listener\ModuleResolverListener`, simply instantiates an instance of `{moduleName}\Module` for each enabled module.

175.1 A Minimal Module

As an example, provided the module name "MyModule", `Zend\ModuleManager\Listener\ModuleResolverListener` will simply expect the class `MyModule\Module` to be available. It relies on a registered autoloader (typically `Zend\Loader\ModuleAutoloader`) to find and include the `MyModule\Module` class if it isn't already available.

The directory structure of a module named "MyModule" might start out looking something like this:

```
MyModule/
    Module.php
```

Within `Module.php`, you define your `MyModule\Module` class:

```
namespace MyModule;

class Module
{
}
```

Though it will not serve any purpose at this point, this "MyModule" module now has everything required to be considered a valid module and to be loaded by the module system!

This `Module` class serves as the single entry point for `ModuleManager` listeners to interact with a module. From within this simple - yet powerful - class, modules can override or provide additional application configuration, perform initialization tasks such as registering autoloader(s), services and event listeners, declaring dependencies, and much more.

175.2 A Typical Module Class

The following example shows a more typical usage of the `Module` class:

```
namespace MyModule;

class Module
```

```
{
    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\ClassMapAutoloader' => array(
                __DIR__ . '/autoload_classmap.php',
            ),
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
                ),
            ),
        );
    }

    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php';
    }
}
```

For a list of the provided module manager listeners and the interfaces and methods that `Module` classes may implement in order to interact with the module manager and application, see the :ref:‘module manager listeners

<zend.module-manager.module-manager.module-manager-listeners>‘ and the *module mananger events* documentations.

175.3 The “loadModules.post” Event

It is not safe for a module to assume that any other modules have already been loaded at the time `init()` method is called. If your module needs to perform any actions after all other modules have been loaded, the module manager’s “loadModules.post” event makes this easy.

Note: For more information on methods like `init()` and `getConfig()`, refer to the *module manager listeners documentation*.

175.3.1 Sample Usage of “loadModules.post” Event

```
use Zend\EventManager\EventInterface as Event;
use Zend\ModuleManager\ModuleManager;

class Module
{
    public function init(ModuleManager $moduleManager)
    {
        // Remember to keep the init() method as lightweight as possible
        $events = $moduleManager->getEventManager();
        $events->attach('loadModules.post', array($this, 'modulesLoaded'));
    }

    public function modulesLoaded(Event $e)
    {
        // This method is called once all modules are loaded.
```

```
        $moduleManager = $e->getTarget();
        $loadedModules = $moduleManager->getLoadedModules();
        // To get the configuration from another module named 'FooModule'
        $config = $moduleManager->getModule('FooModule')->getConfig();
    }
}
```

Note: The `init()` method is called for **every** module implementing this feature, on **every** page request, and should **only** be used for performing **lightweight** tasks such as registering event listeners.

175.4 The MVC "bootstrap" Event

If you are writing an MVC-oriented module for Zend Framework 2, you may need access to additional parts of the application in your `Module` class such as the instance of `Zend\Mvc\Application` or its registered `ServiceManager` instance. For this, you may utilize the MVC "bootstrap" event. The bootstrap event is triggered after the "loadModule.post" event, once *$application->bootstrap()* is called.

175.4.1 Sample Usage of the MVC "bootstrap" Event

```
use Zend\EventManager\EventInterface as Event;

class Module
{
    public function onBootstrap(Event $e)
    {
        // This method is called once the MVC bootstrapping is complete
        $application = $e->getApplication();
        $services    = $application->getServiceManager();
    }
}
```

Note: The `onBootstrap()` method is called for **every** module implementing this feature, on **every** page request, and should **only** be used for performing **lightweight** tasks such as registering event listeners.

CHAPTER

SIX

THE MODULE AUTOLOADER

Zend Framework 2 ships with the default module autoloader `Zend\Loader\ModuleAutoloader`. It is a specialized autoloader responsible for locating and on-demand loading of, the `Module` classes from a variety of sources.

176.1 Module Autoloader Usage

By default, the provided `Zend\ModuleManager\Listener\DefaultListenerAggregate` sets up the `ModuleAutoloader`; as a developer, you need only provide an array of module paths, either absolute or relative to the application's root, for the `ModuleAutoloader` to check when loading modules. The `DefaultListenerAggregate` will take care of instantiating and registering the `ModuleAutoloader` for you.

Note: In order for paths relative to your application directory to work, you must have the directive `chdir(dirname(__DIR__));` in your `public/index.php` file.

Registering module paths with the `DefaultListenerAggregate`

The following example will search for modules in three different `module_paths`. Two are local directories of this application and the third is a system-wide shared directory.

```
// public/index.php
use Zend\ModuleManager\Listener;
use Zend\ModuleManager\ModuleManager;

chdir(dirname(__DIR__));

// Instantiate and configure the default listener aggregate
$listenerOptions = new Listener\ListenerOptions(array(
    'module_paths' => array(
        './module',
        './vendor',
        '/usr/share/zfmodules',
    )
));
$defaultListeners = new Listener\DefaultListenerAggregate($listenerOptions);

// Instantiate the module manager
$moduleManager = new ModuleManager(array(
    'Application',
    'FooModule',
```

```
    'BarModule',
));

// Attach the default listener aggregate and load the modules
$moduleManager->getEventManager()->attachAggregate($defaultListeners);
$moduleManager->loadModules();
```

Note: Module paths behave very similar to PHP's `include_path` and are searched in the order they are defined. If you have modules with the same name in more than one registered module path, the module autoloader will return the first one it finds.

176.2 Non-Standard / Explicit Module Paths

Sometimes you may want to specify exactly where a module is instead of having `Zend\Loader\ModuleAutoloader` try to find it in the registered paths.

Registering a Non-Standard / Explicit Module Path

In this example, the autoloader will first check for `MyModule\Module` in `/path/to/mymoduledir-v1.2/Module.php`. If it's not found, then it will fall back to searching any other registered module paths.

```
// ./public/index.php
use Zend\Loader\ModuleAutoloader;
use Zend\ModuleManager\Listener;
use Zend\ModuleManager\ModuleManager;

chdir(dirname(__DIR__));

// Instantiate and configure the default listener aggregate
$listenerOptions = new Listener\ListenerOptions(array(
    'module_paths' => array(
        './module',
        './vendor',
        '/usr/share/zfmodules',
        'MyModule' => '/path/to/mymoduledir-v1.2',
    )
));
$defaultListeners = new Listener\DefaultListenerAggregate($listenerOptions);

/**
 * Without DefaultListenerAggregate:
 *
 * $moduleAutoloader = new ModuleAutoloader(array(
 *     './module',
 *     './vendor',
 *     '/usr/share/zfmodules',
 *     'MyModule' => '/path/to/mymoduledir-v1.2',
 * ));
 * $moduleAutoloader->register();
 *
 */

```

```
// Instantiate the module manager
$moduleManager = new ModuleManager(array(
    'MyModule',
    'FooModule',
    'BarModule',
));

// Attach the default listener aggregate and load the modules
$moduleManager->getEventManager()->attachAggregate($defaultListeners);
$moduleManager->loadModules();
```

This same method works if you provide the path to a phar archive.

176.3 Packaging Modules with Phar

If you prefer, you may easily package your module as a phar archive. The module autoloader is able to autoload modules in the following archive formats: .phar, .phar.gz, .phar.bz2, .phar.tar, .phar.tar.gz, .phar.tar.bz2, .phar.zip, .tar, .tar.gz, .tar.bz2, and .zip.

The easiest way to package your module is to simply tar the module directory. You can then replace the `MyModule/` directory with `MyModule.tar`, and it should still be autoloaded without any additional changes!

Note: If possible, avoid using any type of compression (bz2, gz, zip) on your phar archives, as it introduces unnecessary CPU overhead to each request.

CHAPTER

SEVEN

BEST PRACTICES WHEN CREATING MODULES

When creating a ZF2 module, there are some best practices you should keep in mind.

- **Keep the ‘‘init()‘‘ and ‘‘onBootstrap()‘‘ methods lightweight.** Be conservative with the actions you perform in the `init()` and `onBootstrap()` methods of your `Module` class. These methods are run for **every** page request, and should not perform anything heavy. As a rule of thumb, registering event listeners is an appropriate task to perform in these methods. Such lightweight tasks will generally not have a measurable impact on the performance of your application, even with many modules enabled. It is considered bad practice to utilize these methods for setting up or configuring instances of application resources such as a database connection, application logger, or mailer. Tasks such as these are better served through the `ServiceManager` capabilities of Zend Framework 2.
- **Do not perform writes within a module.** You should **never** code your module to perform or expect any writes within the module's directory. Once installed, the files within a module's directory should always match the distribution verbatim. Any user-provided configuration should be performed via overrides in the Application module or via application-level configuration files. Any other required filesystem writes should be performed in some writeable path that is outside of the module's directory.

  There are two primary advantages to following this rule. First, any modules which attempt to write within themselves will not be compatible with phar packaging. Second, by keeping the module in sync with the upstream distribution, updates via mechanisms such as Git will be simple and trouble-free. Of course, the Application module is a special exception to this rule, as there is typically no upstream distribution for this module, and it's unlikely you would want to run this package from within a phar archive.
- **Utilize a vendor prefix for module names.** To avoid module naming conflicts, you are encouraged to prefix your module namespace with a vendor prefix. As an example, the (incomplete) developer tools module distributed by Zend is named "ZendDeveloperTools" instead of simply "DeveloperTools".
- **Utilize a module prefix for service names.** If you define services in the top-level Service Manager, you are encouraged to prefix these services with the name of your module to avoid conflicts with other modules' services. For example, the database adapter used by MyModule should be called "MyModuleDbAdapter" rather than simply "DbAdapter." If you need to share a service with other modules, remember that the Service Manager "alias" feature can be used in a merged configuration to override factories defined by individual modules. Ideally, modules should define their own service dependencies, but aliases can be configured at the application level to ensure that common services in individual modules all refer to the same instance.

CHAPTER

EIGHT

INTRODUCTION TO THE MVC LAYER

`Zend\Mvc` is a brand new MVC implementation designed from the ground up for Zend Framework 2, focusing on performance and flexibility.

The MVC layer is built on top of the following components:

- `Zend\ServiceManager` - Zend Framework provides a set of default service definitions set up at `Zend\Mvc\Service`. The `ServiceManager` creates and configures your application instance and workflow.
- `Zend\EventManager` - The MVC is event driven. This component is used everywhere from initial bootstrapping of the application, through returning response and request calls, to setting and retrieving routes and matched routes, as well as render views.
- `Zend\Http` - specifically the request and response objects, used within:
- `Zend\Stdlib\DispatchableInterface`. All "controllers" are simply dispatchable objects.

Within the MVC layer, several sub-components are exposed:

- `Zend\Mvc\Router` contains classes pertaining to routing a request. In other words, it matches the request to its respective controller (or dispatchable).
- `Zend\Http\PhpEnvironment` provides a set of decorators for the HTTP `Request` and `Response` objects that ensure the request is injected with the current environment (including query parameters, POST parameters, HTTP headers, etc.)
- `Zend\Mvc\Controller`, a set of abstract "controller" classes with basic responsibilities such as event wiring, action dispatching, etc.
- `Zend\Mvc\Service` provides a set of `ServiceManager` factories and definitions for the default application workflow.
- `Zend\Mvc\View` provides default wiring for renderer selection, view script resolution, helper registration, and more; additionally, it provides a number of listeners that tie into the MVC workflow, providing features such as automated template name resolution, automated view model creation and injection, and more.

The gateway to the MVC is the Zend\Mvc\Application object (referred to as `Application` henceforth). Its primary responsibilities are to **bootstrap** resources, **route** the request, and to retrieve and **dispatch** the controller matched during routing. Once accomplished, it will **render** the view, and **finish** the request, returning and sending the response.

178.1 Basic Application Structure

The basic application structure follows:

```
application_root/
    config/
        application.config.php
        autoload/
            global.php
            local.php
            // etc.
    data/
    module/
    vendor/
    public/
        .htaccess
        index.php
    init_autoloader.php
```

The `public/index.php` marshalls all user requests to your website, retrieving an array of configuration located in `config/application.config.php`. On return, it `run()`s the `Application`, processing the request and returning a response to the user.

The `config` directory as described above contains configuration used by the `Zend\ModuleManager` to load modules and merge configuration (e.g., database configuration and credentials); we will detail this more later.

The `vendor` sub-directory should contain any third-party modules or libraries on which your application depends. This might include Zend Framework, custom libraries from your organization, or other third-party libraries from other projects. Libraries and modules placed in the `vendor` sub-directory should not be modified from their original, distributed state.

Finally, the `module` directory will contain one or more modules delivering your application's functionality.

Let's now turn to modules, as they are the basic units of a web application.

178.2 Basic Module Structure

A module may contain anything: PHP code, including MVC functionality; library code; view scripts; and/or or public assets such as images, CSS, and JavaScript. The only requirement – and even this is optional – is that a module acts as a PHP namespace and that it contains a `Module.php` class under that namespace. This class is eventually consumed by `Zend\ModuleManager` to perform a number of tasks.

The recommended module structure follows:

```
module_root<named-after-module-namespace>/
    Module.php
    autoload_classmap.php
    autoload_function.php
    autoload_register.php
    config/
        module.config.php
    public/
        images/
        css/
        js/
    src/
        <module_namespace>/
            <code files>
    test/
        phpunit.xml
        bootstrap.php
```

```
        <module_namespace>/
            <test code files>
    view/
        <dir-named-after-module-namespace>/
            <dir-named-after-a-controller>/
                <.phtml files>
```

Since a module acts as a namespace, the module root directory should be that namespace. This namespace could also include a vendor prefix of sorts. As an example a module centered around "User" functionality delivered by Zend might be named "ZendUser", and this is also what the module root directory will be named.

The `Module.php` file directly under the module root directory will be in the module namespace shown below.

```
namespace ZendUser;

class Module
{
}
```

When an `init()` method is defined, this method will be triggered by a `Zend\ModuleManager` listener when it loads the module class, and passed an instance of the manager by default. This allows you to perform tasks such as setting up module-specific event listeners. But be cautious, the `init()` method is called for **every** module on **every** page request and should **only** be used for performing **lightweight** tasks such as registering event listeners. Similarly, an `onBootstrap()` method (which accepts an `MvcEvent` instance) may be defined; it is also triggered for every page request, and should be used for lightweight tasks as well.

The three `autoload_*.php` files are not required, but recommended. They provide the following:

Table 178.1: autoload_*.php Files

File	Description
`autoload_classmap.php`	Should return an array classmap of class name/filename pairs (with the filenames resolved via the `__DIR__` magic constant).
`autoload_function.php`	Should return a PHP callback that can be passed to `spl_autoload_register()`. Typically, this callback should utilize the map returned by `autoload_classmap.php`.
`autoload_register.php`	Should register a PHP callback (is typically returned by `autoload_function.php` with `spl_autoload_register()`.

The point of these three files is to provide reasonable default mechanisms for autoloading the classes contained in the module, thus providing a trivial way to consume the module without requiring `Zend\ModuleManager` (e.g., for use outside a ZF2 application).

The `config` directory should contain any module-specific configuration. These files may be in any format `Zend\Config` supports. We recommend naming the main configuration "module.format", and for PHP-based configuration, "module.config.php". Typically, you will create configuration for the router as well as for the dependency injector.

The `src` directory should be a PSR-0 compliant directory structure with your module's source code. Typically, you should at least have one sub-directory named after your module namespace; however, you can ship code from multiple namespaces if desired.

The `test` directory should contain your unit tests. Typically, these are written using PHPUnit, and contain artifacts related to its configuration (e.g., `phpunit.xml`, `bootstrap.php`).

The `public` directory can be used for assets that you may want to expose in your application's document root. These might include images, CSS files, JavaScript files, etc. How these are exposed is left to the developer.

The `view` directory contains view scripts related to your controllers.

178.3 Bootstrapping an Application

The `Application` has six basic dependencies.

- **configuration**, usually an array or object implementing `Traversable`.
- **ServiceManager** instance.
- **EventManager** instance, which, by default, is pulled from the `ServiceManager`, by the service name "EventManager".
- **ModuleManager** instance, which, by default, is pulled from the `ServiceManager`, by the service name "ModuleManager".
- **Request** instance, which, by default, is pulled from the `ServiceManager`, by the service name "Request".
- **Response** instance, which, by default, is pulled from the `ServiceManager`, by the service name "Response".

These may be satisfied at instantiation:

```
use Zend\EventManager\EventManager;
use Zend\Http\PhpEnvironment;
use Zend\ModuleManager\ModuleManager;
use Zend\Mvc\Application;
use Zend\ServiceManager\ServiceManager;

$config = include 'config/application.config.php';

$serviceManager = new ServiceManager();
$serviceManager->setService('EventManager', new EventManager());
$serviceManager->setService('ModuleManager', new ModuleManager());
$serviceManager->setService('Request', new PhpEnvironment\Request());
$serviceManager->setService('Response', new PhpEnvironment\Response());

$application = new Application($config, $serviceManager);
```

Once you've done this, there are two additional actions you can take. The first is to "bootstrap" the application. In the default implementation, this does the following:

- Attaches the default route listener (`Zend\Mvc\RouteListener`).
- Attaches the default dispatch listener (`Zend\Mvc\DispatchListener`).
- Attaches the `ViewManager` listener (`Zend\Mvc\View\ViewManager`).
- Creates the `MvcEvent`, and injects it with the application, request, and response; it also retrieves the router (`Zend\Mvc\Router\Http\TreeRouteStack`) at this time and attaches it to the event.
- Triggers the "bootstrap" event.

If you do not want these actions, or want to provide alternatives, you can do so by extending the `Application` class and/or simply coding what actions you want to occur.

The second action you can take with the configured `Application` is to `run()` it. Calling this method simply does the following: it triggers the "route" event, followed by the "dispatch" event, and, depending on execution, the "render" event; when done, it triggers the "finish" event, and then returns the response instance. If an error occurs during either the "route" or "dispatch" event, a "dispatch.error" event is triggered as well.

This is a lot to remember in order to bootstrap the application; in fact, we haven't covered all the services available by default yet. You can greatly simplify things by using the default `ServiceManager` configuration shipped with the MVC.

```
use Zend\Loader\AutoloaderFactory;
use Zend\Mvc\Service\ServiceManagerConfig;
use Zend\ServiceManager\ServiceManager;

// setup autoloader
AutoloaderFactory::factory();

// get application stack configuration
$configuration = include 'config/application.config.php';

// setup service manager
$serviceManager = new ServiceManager(new ServiceManagerConfig());
$serviceManager->setService('ApplicationConfig', $configuration);

// load modules -- which will provide services, configuration, and more
$serviceManager->get('ModuleManager')->loadModules();

// bootstrap and run application
$application = $serviceManager->get('Application');
$application->bootstrap();
$response = $application->run();
$response->send();
```

You can make this even simpler by using the `init()` method of the `Application`. This is a static method for quick and easy initialization of the Application.

```
use Zend\Loader\AutoloaderFactory;
use Zend\Mvc\Application;
use Zend\Mvc\Service\ServiceManagerConfig;
use Zend\ServiceManager\ServiceManager;

// setup autoloader
AutoloaderFactory::factory();

// get application stack configuration
$configuration = include 'config/application.config.php';

// The init() method does something very similar with the previous example.
Application::init($configuration)->run();
```

The `init()` method will basically do the following:

- **Grabs the application configuration and pulls from the `service_manager` key, creating a `ServiceManager`** instance with it and with the default services shipped with `Zend\Mvc`;
- Create a service named `ApplicationConfig` with the application configuration array;
- Grabs the `ModuleManager` service and load the modules;
- `bootstrap()`s the `Application` and returns its instance;

Note: If you use the `init()` method, you cannot specify a service with the name of 'ApplicationConfig' in your service manager config. This name is reserved to hold the array from application.config.php.

The following services can only be overridden from application.config.php:

- `ModuleManager`
- `SharedEventManager`
- `EventManager` & `Zend\EventManager\EventManagerInterface`

All other services are configured after module loading, thus can be overridden by modules.

You'll note that you have a great amount of control over the workflow. Using the `ServiceManager`, you have fine-grained control over what services are available, how they are instantiated, and what dependencies are injected into them. Using the `EventManager`'s priority system, you can intercept any of the application events ("bootstrap", "route", "dispatch", "dispatch.error", "render", and "finish") anywhere during execution, allowing you to craft your own application workflows as needed.

178.4 Bootstrapping a Modular Application

While the previous approach largely works, where does the configuration come from? When we create a modular application, the assumption will be that it's from the modules themselves. How do we get that information and aggregate it, then?

The answer is via `Zend\ModuleManager\ModuleManager`. This component allows you to specify where modules exist. Then, it will locate each module and initialize it. Module classes can tie into various listeners on the `ModuleManager` in order to provide configuration, services, listeners, and more to the application. Sounds complicated? It's not.

178.4.1 Configuring the Module Manager

The first step is configuring the module manager. Simply inform the module manager which modules to load, and potentially provide configuration for the module listeners.

Remember the `application.config.php` from earlier? We're going to provide some configuration.

```
<?php
// config/application.config.php
return array(
    'modules' => array(
        /* ... */
    ),
    'module_listener_options' => array(
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

As we add modules to the system, we'll add items to the `modules` array.

Each `Module` class that has configuration it wants the `Application` to know about should define a `getConfig()` method. That method should return an array or `Traversable` object such as `Zend\Config\Config`. As an example:

```
namespace ZendUser;

class Module
{
    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php'
    }
}
```

There are a number of other methods you can define for tasks ranging from providing autoloader configuration, to providing services to the `ServiceManager`, to listening to the bootstrap event. The *ModuleManager documentation* goes into more detail on these.

178.5 Conclusion

The ZF2 MVC layer is incredibly flexible, offering an opt-in, easy to create modular infrastructure, as well as the ability to craft your own application workflows via the `ServiceManager` and `EventManager`. The `ModuleManager` is a lightweight and simple approach to enforcing a modular architecture that encourages clean separation of concerns and code re-use.

CHAPTER

NINE

QUICK START

Now that you have basic knowledge of applications, modules, and how they are each structured, we'll show you the easy way to get started.

179.1 Install the Zend Skeleton Application

The easiest way to get started is to grab the sample application and module repositories. This can be done in the following ways.

179.1.1 Using Composer

Simply clone the `ZendSkeletonApplication` repository:

```
prompt> git clone git://github.com/zendframework/ZendSkeletonApplication.git my-application
```

Then run Composer's `install` command to install the ZF library and any other configured dependencies:

```
prompt> php ./composer.phar install
```

179.1.2 Using Git

Simply clone the `ZendSkeletonApplication` repository, using the `--recursive` option, which will also grab ZF.

```
prompt> git clone --recursive git://github.com/zendframework/ZendSkeletonApplication.git my-applicati
```

179.1.3 Manual Installation

- Download a tarball of the `ZendSkeletonApplication` repository:
  - Zip: https://github.com/zendframework/ZendSkeletonApplication/zipball/master
  - Tarball: https://github.com/zendframework/ZendSkeletonApplication/tarball/master
- Deflate the archive you selected and rename the parent directory according to your project needs; we use "my-application" throughout this document.
- Install Zend Framework, and either have its library on your PHP `include_path`, symlink the library into your project's "library", or install it directly into your application using Pyrus.

179.2 Create a New Module

By default, one module is provided with the `ZendSkeletonApplication`, named "Application". It simply provides a controller to handle the "home" page of the application, the layout template, and templates for 404 and error pages.

Typically, you will not need to touch this other than to provide an alternate entry page for your site and/or alternate error page.

Additional functionality will be provided by creating new modules.

To get you started with modules, we recommend using the `ZendSkeletonModule` as a base. Download it from here:

- Zip: https://github.com/zendframework/ZendSkeletonModule/zipball/master
- Tarball: https://github.com/zendframework/ZendSkeletonModule/tarball/master

Deflate the package, and rename the directory "ZendSkeletonModule" to reflect the name of the new module you want to create; when done, move the module into your new project's `module/` directory.

At this point, it's time to create some functionality.

179.3 Update the Module Class

Let's update the module class. We'll want to make sure the namespace is correct, configuration is enabled and returned, and that we setup autoloading on initialization. Since we're actively working on this module, the class list will be in flux, we probably want to be pretty lenient in our autoloading approach, so let's keep it flexible by using the `StandardAutoloader`. Let's begin.

First, let's have `autoload_classmap.php` return an empty array:

```
<?php
// autoload_classmap.php
return array();
```

We'll also edit our `config/module.config.php` file to read as follows:

```
return array(
    'view_manager' => array(
        'template_path_stack' => array(
            '<module-name>' => __DIR__ . '/../view'
        ),
    ),
);
```

Fill in "module-name" with a lowercased, dash-separated version of your module name – e.g., "ZendUser" would become "zend-user".

Next, edit the `Module.php` file to read as follows:

```
namespace <your module name here>;

use Zend\ModuleManager\Feature\AutoloaderProviderInterface;
use Zend\ModuleManager\Feature\ConfigProviderInterface;

class Module implements AutoloaderProviderInterface, ConfigProviderInterface
{
    public function getAutoloaderConfig()
```

```
    {
        return array(
            'Zend\Loader\ClassMapAutoloader' => array(
                __DIR__ . '/autoload_classmap.php',
            ),
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
                ),
            ),
        );
    }

    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php';
    }
}
```

At this point, you now have your module configured properly. Let's create a controller!

179.4 Create a Controller

Controllers are simply objects that implement `Zend\Stdlib\DispatchableInterface`. This means they need to implement a `dispatch()` method that takes minimally a `Request` object as an argument.

In practice, though, this would mean writing logic to branch based on matched routing within every controller. As such, we've created two base controller classes for you to start with:

- `Zend\Mvc\Controller\AbstractActionController` allows routes to match an "action". When matched, a method named after the action will be called by the controller. As an example, if you had a route that returned "foo" for the "action" key, the "fooAction" method would be invoked.
- `Zend\Mvc\Controller\AbstractRestfulController` introspects the `Request` to determine what HTTP method was used, and calls a method according to that.
  - `GET` will call either the `getList()` method, or, if an "id" was matched during routing, the `get()` method (with that identifer value).
  - `POST` will call the `create()` method, passing in the `$_POST` values.
  - `PUT` expects an "id" to be matched during routing, and will call the `update()` method, passing in the identifier, and any data found in the raw post body.
  - `DELETE` expects an "id" to be matched during routing, and will call the `delete()` method.

To get started, we'll simply create a "hello world"-style controller, with a single action. First, create the directory `src/<module name>/Controller`, and then create the file `HelloController.php` inside it. Edit it in your favorite text editor or IDE, and insert the following contents:

```
<?php
namespace <module name>\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class HelloController extends AbstractActionController
{
```

```
    public function worldAction()
    {
        $message = $this->params()->fromQuery('message', 'foo');
        return new ViewModel(array('message' => $message));
    }
}
```

So, what are we doing here?

- We're creating an action controller.
- We're defining an action, "world".
- We're pulling a message from the query parameters (yes, this is a superbly bad idea in production! Always sanitize your inputs!).
- We're returning a ViewModel with an array of values to be processed later.

We return a `ViewModel`. The view layer will use this when rendering the view, pulling variables and the template name from it. By default, you can omit the template name, and it will resolve to "lowercase-controller-name/lowercase-action-name". However, you can override this to specify something different by calling `setTemplate()` on the `ViewModel` instance. Typically, templates will resolve to files with a ".phtml" suffix in your module's `view` directory.

So, with that in mind, let's create a view script.

179.5 Create a View Script

Create the directory `view/<module-name>/hello`. Inside that directory, create a file named `world.phtml`. Inside that, paste in the following:

```
<h1>Greetings!</h1>

<p>You said "<?php echo $this->escapeHtml($message) ?>".</p>
```

That's it. Save the file.

Note: What is the method `escapeHtml()`? It's actually a *view helper*, and it's designed to help mitigate *XSS* attacks. Never trust user input; if you are at all uncertain about the source of a given variable in your view script, escape it using one of the *provided escape view helper* depending on the type of data you have.

179.6 Create a Route

Now that we have a controller and a view script, we need to create a route to it.

Note: `ZendSkeletonApplication` ships with a "default route" that will likely get you to this action. That route basically expects "/{module}/{controller}/{action}", which allows you to specify this: "/zend-user/hello/world". We're going to create a route here mainly for illustration purposes, as creating explicit routes is a recommended practice. The application will look for a `Zend\Mvc\Router\RouteStack` instance to setup routing. The default generated router is a `Zend\Mvc\Router\Http\TreeRouteStack`.

To use the "default route" functionality, you will need to have the following route definition in your module. Replace <module-name> with the name of your module.

```
// module.config.php
return array(
    '<module-name>' => array(
        'type'    => 'Literal',
        'options' => array(
            'route'    => '/<module-name>',
            'defaults' => array(
                'controller'    => '<module-namespace>\Controller\Index',
                'action'        => 'index',
            ),
        ),
        'may_terminate' => true,
        'child_routes' => array(
            'default' => array(
                'type'    => 'Segment',
                'options' => array(
                    'route'    => '/[:controller[/:action]]',
                    'constraints' => array(
                        'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
                    ),
                    'defaults' => array(
                    ),
                ),
            ),
        ),
    ),
    // ... other configuration ...
);
```

Additionally, we need to tell the application we have a controller:

```
// module.config.php
return array(
    'controllers' => array(
        'invokables' => array(
            '<module-namespace>\Controller\Index' => '<module-namespace>\Controller\IndexControl
            // Do similar for each other controller in your module
        ),
    ),
    // ... other configuration ...
);
```

Note: We inform the application about controllers we expect to have in the application. This is to prevent somebody requesting any service the `ServiceManager` knows about in an attempt to break the application. The dispatcher uses a special, scoped container that will only pull controllers that are specifically registered with it, either as invokable classes or via factories.

Open your `config/module.config.php` file, and modify it to add to the "routes" and "controller" parameters so it reads as follows:

```
return array(
    'router' => array(
        'routes' => array(
            '<module name>-hello-world' => array(
                'type'    => 'Literal',
```

```
                        'options' => array(
                        'route' => '/hello/world',
                        'defaults' => array(
                            'controller' => '<module name>\Controller\Hello',
                            'action'     => 'world',
                        ),
                    ),
                ),
            ),
        ),
    'controllers' => array(
        'invokables' => array(
            '<module namespace>\Controller\Hello' => '<module namespace>\Controller\HelloController',
        ),
    ),
    // ... other configuration ...
);
```

179.7 Tell the Application About our Module

One problem: we haven't told our application about our new module!

By default, modules are not parsed unless we tell the module manager about them. As such, we need to notify the application about them.

Remember the `config/application.config.php` file? Let's modify it to add our new module. Once done, it should read as follows:

```
<?php
return array(
    'modules' => array(
        'Application',
        '<module namespace>',
    ),
    'module_listener_options' => array(
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

Replace `<module namespace>` with the namespace of your module.

179.8 Test it Out!

Now we can test things out! Create a new vhost pointing its document root to the `public` directory of your application, and fire it up in a browser. You should see the default homepage template of ZendSkeletonApplication.

Now alter the location in your URL to append the path "/hello/world", and load the page. You should now get the following content:

```
<h1>Greetings!</h1>

<p>You said "foo".</p>
```

Now alter the location to append "?message=bar" and load the page. You should now get:

```
<h1>Greetings!</h1>

<p>You said "bar".</p>
```

Congratulations! You've created your first ZF2 MVC module!

CHAPTER

DEFAULT SERVICES

The default and recommended way to write Zend Framework applications uses a set of services defined in the `Zend\Mvc\Service` namespace. This chapter details what each of those services are, the classes they represent, and the configuration options available.

180.1 Theory of Operation

To allow easy configuration of all the different parts of the *MVC* system, a somewhat complex set of services and their factories has been created. We'll try to give a simplified explanation of the process.

When a `Zend\Mvc\Application` is created, a `Zend\ServiceManager\ServiceManager` object is created and configured via `Zend\Mvc\Service\ServiceManagerConfig`. The `ServiceManagerConfig` gets the configuration from `application.config.php` (or some other *application* configuration you passed to the `Application` when creating it). From all the service and factories provided in the `Zend\Mvc\Service` namespace, `ServiceManagerConfig` is responsible of configuring only three: `SharedEventManager`, `EventManager`, and `ModuleManager`.

After this, the `Application` calls for the `ModuleManager`. At this point, the `ModuleManager` further configures the `ServiceManager` with services and factories provided in `Zend\Mvc\Service\ServiceLocator`. This approach allows to keep the main application configuration as simple as possible, and to give the developer the power to configure different parts of the *MVC* system from within the modules, overriding any default configuration in these *MVC* services.

180.2 ServiceManager

This is the one service class referenced directly in the application bootstrapping. It provides the following:

- **Invokable services**
  - `DispatchListener`, mapping to `Zend\Mvc\DispatchListener`.
  - `RouteListener`, mapping to `Zend\Mvc\RouteListener`.
  - `SendResponseListener`, mapping to `Zend\Mvc\SendResponseListener`.
  - `SharedEventManager`, mapping to `Zend\EventManager\SharedEventManager`.
- **Factories**
  - `Application`, mapping to `Zend\Mvc\Service\ApplicationFactory`.
  - `Config`, mapping to `Zend\Mvc\Service\ConfigFactory`. Internally, this pulls the `ModuleManager` service, and calls its `loadModules()` method, and retrieves the merged configuration from the module event. As such, this service contains the entire, merged application configuration.

- `ControllerLoader`, mapping to `Zend\Mvc\Service\ControllerLoaderFactory`. This creates an instance of `Zend\Mvc\Controller\ControllerManager`, passing the service manager instance.

  Additionally, it uses the `DiStrictAbstractServiceFactory` service – effectively allowing you to fall back to DI in order to retrieve your controllers. If you want to use `Zend\Di` to retrieve your controllers, you must white-list them in your DI configuration under the `allowed_controllers` key (otherwise, they will just be ignored).

  The `ControllerManager` will add an initializer that will do the following:

  * If the controller implements the `Zend\ServiceManager\ServiceLocatorAwareInterface` interface, an instance of the `ServiceManager` will be injected into it.
  * If the controller implements the `Zend\EventManager\EventManagerAwareInterface` interface, an instance of the `EventManager` will be injected into it.
  * Finally, an initializer will inject it with the `ControllerPluginManager` service, as long as the `setPluginManager` method is implemented.

- `ControllerPluginManager`, mapping to `Zend\Mvc\Service\ControllerPluginManagerFactory`. This instantiates the `Zend\Mvc\Controller\PluginManager` instance, passing it the service manager instance. It also uses the `DiAbstractServiceFactory` service – effectively allowing you to fall back to DI in order to retrieve your *controller plugins*.

  It registers a set of default controller plugins, and contains an initializer for injecting plugins with the current controller.

- `ConsoleAdapter`, mapping to `Zend\Mvc\Service\ConsoleAdapterFactory`. This grabs the `Config` service, pulls from the `console` key, and do the following:

  * If the `adapter` subkey is present, it is used to get the adapter instance, otherwise,

  `Zend\Console\Console::detectBestAdapter()` will be called to configure an adapter instance.

  * If the `charset` subkey is present, the is used to set the adapter charset.

- `ConsoleRouter`, mapping to `Zend\Mvc\Service\RouterFactory`. This grabs the `Config` service, and pulls from the `console` key and `router` subkey, configuring a `Zend\Mvc\Router\Console\SimpleRouteStack` instance.

- `DependencyInjector`, mapping to `Zend\Mvc\Service\DiFactory`. This pulls the `Config` service, and looks for a "di" key; if found, that value is used to configure a new `Zend\Di\Di` instance.

- `DiAbstractServiceFactory`, mapping to `Zend\Mvc\Service\DiAbstractServiceFactoryFactory`. This creates an instance of `Zend\ServiceManager\Di\DiAbstractServiceFactory` injecting the `Di` service instance. That instance is attached to the service manager as an abstract factory – effectively enabling DI as a fallback for providing services.

- `DiServiceInitializer`, mapping to `Zend\Mvc\Service\DiServiceInitializerFactory`. This creates an instance of `Zend\ServiceManager\Di\DiServiceInitializer` injecting the `Di` service and the service manager itself.

- `DiStrictAbstractServiceFactory`, mapping to `Zend\Mvc\Service\DiStrictAbstractServiceFact` This creates an instance of `Zend\Mvc\Service\DiStrictAbstractServiceFactoryFactory` injecting the `Di` service instance.

- `EventManager`, mapping to `Zend\Mvc\Service\EventManagerFactory`. This factory composes a static reference to a `SharedEventManager`, which is injected in a new `EventManager` instance. This service is not shared by default, allowing the ability to have an `EventManager` per service, with a shared `SharedEventManager` injected in each.

- `FilterManager`, mapping to `Zend\Mvc\Service\FilterManagerFactory`. This instantiates the `Zend\Filter\FilterPluginManager` instance, passing it the service manager instance – this is used to manage filters for the *filter chains*. It also uses the `DiAbstractServiceFactory` service – effectively allowing you to fall back to DI in order to retrieve filters.
- `FormElementManager`, mapping to `Zend\Mvc\Service\FormElementManagerFactory`. This instantiates the `Zend\Form\FormElementManager` instance, passing it the service manager instance – this is used to manage *form elements*. It also uses the `DiAbstractServiceFactory` service – effectively allowing you to fall back to DI in order to retrieve form elements.
- `HttpRouter`, mapping to `Zend\Mvc\Service\RouterFactory`. This grabs the `Config` service, and pulls from the `router` key, configuring a `Zend\Mvc\Router\Http\TreeRouteStack` instance.
- `ModuleManager`, mapping to `Zend\Mvc\Service\ModuleManagerFactory`.

  This is perhaps the most complex factory in the MVC stack. It expects that an `ApplicationConfig` service has been injected, with keys for `module_listener_options` and `modules`; see the quick start for samples.

  It instantiates an instance of `Zend\ModuleManager\Listener\DefaultListenerAggregate`, using the "module_listener_options" retrieved. Checks if a service with the name `ServiceListener` exists, otherwise it sets a factory with that name mapping to `Zend\Mvc\Service\ServiceListenerFactory`. A bunch of service listeners will be added to the `ServiceListener`, like listeners for the `getServiceConfig`, `getControllerConfig`, `getControllerPluginConfig`, `getViewHelperConfig` module methods.

  Next, it retrieves the `EventManager` service, and attaches the above listeners.

  It instantiates a `Zend\ModuleManager\ModuleEvent` instance, setting the "ServiceManager" parameter to the service manager object.

  Finally, it instantiates a `Zend\ModuleManager\ModuleManager` instance, and injects the `EventManager` and `ModuleEvent`.
- `PaginatorPluginManager`, mapping to `Zend\Mvc\Service\PaginatorPluginManagerFactory`. This instantiates the `Zend\Paginator\AdapterPluginManager` instance, passing it the service manager instance – this is used to manage *paginator adapters*. It also uses the `DiAbstractServiceFactory` service – effectively allowing you to fall back to DI in order to retrieve paginator adapters.
- `Request`, mapping to `Zend\Mvc\Service\RequestFactory`. The factory is used to create and return a request instance, according to the current environment. If the current environment is `cli`, it will create a `Zend\Console\Request`, or a `Zend\Http\PhpEnvironment\Request` if the current environment is *HTTP*.
- `Response`, mapping to `Zend\Mvc\Service\ResponseFactory`. The factory is used to create and return a response instance, according to the current environment. If the current environment is `cli`, it will create a `Zend\Console\Response`, or a `Zend\Http\PhpEnvironment\Response` if the current environment is *HTTP*.
- `Router`, mapping to `Zend\Mvc\Service\RouterFactory`. If in a console enviroment, this will behave the same way as the `ConsoleRouter` service, if not, it will behave the same way as `HttpRouter` service.
- `RoutePluginManager`, mapping to `Zend\Mvc\Service\RoutePluginManagerFactory`. This instantiates the `Zend\Mvc\Router\RoutePluginManager` instance, passing it the service manager instance – this is used to manage *route types*. It also uses the `DiAbstractServiceFactory` service – effectively allowing you to fall back to DI in order to retrieve route types.

- `ServiceListener`, mapping to `Zend\Mvc\Service\ServiceListenerFactory`. The factory is used to instantiate the `ServiceListener`, while allowing easy extending. It checks if a service with the name `ServiceListenerInterface` exists, which must implement `Zend\ModuleManager\Listener\ServiceListenerInterface`, before instantiating the default `ServiceListener`.

  In addition to this, it retrieves the `ApplicationConfig` and looks for the `service_listener_options` key. This allows you to register own listeners for module methods and configuration keys to create an own service manager; see the *application configuration options* for samples.
- `ValidatorManager`, mapping to `Zend\Mvc\Service\ValidatorManagerFactory`. This instantiates the `Zend\Validator\ValidatorPluginManager` instance, passing it the service manager instance – this is used to manage *validators*. It also uses the `DiAbstractServiceFactory` service – effectively allowing you to fall back to DI in order to retrieve validators.
- `ViewManager`, mapping to `Zend\Mvc\Service\ViewManagerFactory`. The factory is used to create and return a view manager, according to the current environment. If the current environment is `cli`, it will create a `Zend\Mvc\View\Console\ViewManager`, or a `Zend\Mvc\View\Http\ViewManager` if the current environment is *HTTP*.
- `ViewResolver`, mapping to `Zend\Mvc\Service\ViewResolverFactory`, which creates and returns the aggregate view resolver. It also attaches the `ViewTemplateMapResolver` and `ViewTemplatePathStack` services to it.
- `ViewTemplateMapResolver`, mapping to `Zend\Mvc\Service\ViewTemplateMapResolverFactory` which creates, configures and returns the `Zend\View\Resolver\TemplateMapResolver`.
- `ViewTemplatePathStack`, mapping to `Zend\Mvc\Service\ViewTemplatePathStackFactory` which creates, configures and returns the `Zend\View\Resolver\TemplatePathStack`.
- `ViewHelperManager`, mapping to `Zend\Mvc\Service\ViewHelperManagerFactory`, which creates, configures and returns the view helper manager.
- `ViewFeedRenderer`, mapping to `Zend\Mvc\Service\ViewFeedRendererFactory`, which simply returns a `Zend\View\Renderer\FeedRenderer` instance.
- `ViewFeedStrategy`, mapping to `Zend\Mvc\Service\ViewFeedStrategyFactory`. This instantiates a `Zend\View\Strategy\FeedStrategy` instance with the `ViewFeedRenderer` service.
- `ViewJsonRenderer`, mapping to `Zend\Mvc\Service\ViewJsonRendererFactory`, which simply returns a `Zend\View\Renderer\JsonRenderer` instance.
- `ViewJsonStrategy`, mapping to `Zend\Mvc\Service\ViewJsonStrategyFactory`. This instantiates a `Zend\View\Strategy\JsonStrategy` instance with the `ViewJsonRenderer` service.

- **Aliases**
  - `Configuration`, mapping to the `Config` service.
  - `Console`, mapping to the `ConsoleAdapter` service.
  - `Di`, mapping to the `DependencyInjector` service.
  - `Zend\Di\LocatorInterface`, mapping to the `DependencyInjector` service.
  - `Zend\EventManager\EventManagerInterface`, mapping to the `EventManager` service. This is mainly to ensure that when falling through to DI, classes are still injected via the `ServiceManager`.

  - `Zend\Mvc\Controller\PluginManager`, mapping to the `ControllerPluginManager` service. This is mainly to ensure that when falling through to DI, classes are still injected via the `ServiceManager`.
  - `Zend\View\Resolver\TemplateMapResolver`, mapping to the `ViewTemplateMapResolver` service.
  - `Zend\View\Resolver\TemplatePathStack`, mapping to the `ViewTemplatePathStack` service.
  - `Zend\View\Resolver\AggregateResolver`, mapping to the `ViewResolver` service.
  - `Zend\View\Resolver\ResolverInterface`, mapping to the `ViewResolver` service.

Additionally, two initializers are registered. Initializers are run on created instances, and may be used to further configure them. The two initializers the `ServiceManagerConfig` class creates and registers do the following:

- For objects that implement `Zend\EventManager\EventManagerAwareInterface`, the `EventManager` service will be retrieved and injected. This service is **not** shared, though each instance it creates is injected with a shared instance of `SharedEventManager`.
- For objects that implement `Zend\ServiceManager\ServiceLocatorAwareInterface`, the `ServiceManager` will inject itself into the object.

Finally, the `ServiceManager` registers itself as the `ServiceManager` service, and aliases itself to the class names `Zend\ServiceManager\ServiceLocatorInterface` and `Zend\ServiceManager\ServiceManager`.

180.3 ViewManager

The View layer within `Zend\Mvc` consists of a large number of collaborators and event listeners. As such, `Zend\Mvc\View\ViewManager` was created to handle creation of the various objects, as well as wiring them together and establishing event listeners.

The `ViewManager` itself is an event listener on the `bootstrap` event. It retrieves the `ServiceManager` from the `Application` object, as well as its composed `EventManager`.

Configuration for all members of the `ViewManager` fall under the `view_manager` configuration key, and expect values as noted below. The following services are created and managed by the `ViewManager`:

- `ViewHelperManager`, representing and aliased to `Zend\View\HelperPluginManager`. It is seeded with the `ServiceManager`. Created via the `Zend\Mvc\Service\ViewHelperManagerFactory`.
  - The `Router` service is retrieved, and injected into the `Url` helper.
  - If the `base_path` key is present, it is used to inject the `BasePath` view helper; otherwise, the `Request` service is retrieved, and the value of its `getBasePath()` method is used.
  - If the `doctype` key is present, it will be used to set the value of the `Doctype` view helper.
- `ViewTemplateMapResolver`, representing and aliased to `Zend\View\Resolver\TemplateMapResolver`. If a `template_map` key is present, it will be used to seed the template map.
- `ViewTemplatePathStack`, representing and aliased to `Zend\View\Resolver\TemplatePathStack`. If a `template_path_stack` key is present, it will be used to seed the stack.
- `ViewResolver`, representing and aliased to `Zend\View\Resolver\AggregateResolver` and `Zend\View\Resolver\ResolverInterface`. It is seeded with the `ViewTemplateMapResolver` and `ViewTemplatePathStack` services as resolvers.

- ViewRenderer, representing and aliased to Zend\View\Renderer\PhpRenderer and Zend\View\Renderer\RendererInterface. It is seeded with the ViewResolver and ViewHelperManager services. Additionally, the ViewModel helper gets seeded with the ViewModel as its root (layout) model.
- ViewPhpRendererStrategy, representing and aliased to Zend\View\Strategy\PhpRendererStrategy. It gets seeded with the ViewRenderer service.
- View, representing and aliased to Zend\View\View. It gets seeded with the EventManager service, and attaches the ViewPhpRendererStrategy as an aggregate listener.
- DefaultRenderingStrategy, representing and aliased to Zend\Mvc\View\DefaultRenderingStrategy. If the layout key is present, it is used to seed the strategy's layout template. It is seeded with the View service.
- ExceptionStrategy, representing and aliased to Zend\Mvc\View\ExceptionStrategy. If the display_exceptions or exception_template keys are present, they are used to configure the strategy.
- RouteNotFoundStrategy, representing and aliased to Zend\Mvc\View\RouteNotFoundStrategy and 404Strategy. If the display_not_found_reason or not_found_template keys are present, they are used to configure the strategy.
- ViewModel. In this case, no service is registered; the ViewModel is simply retrieved from the MvcEvent and injected with the layout template name.

The ViewManager also creates several other listeners, but does not expose them as services; these include Zend\Mvc\View\CreateViewModelListener, Zend\Mvc\View\InjectTemplateListener, and Zend\Mvc\View\InjectViewModelListener. These, along with RouteNotFoundStrategy, ExceptionStrategy, and DefaultRenderingStrategy are attached as listeners either to the application EventManager instance or the SharedEventManager instance.

Finally, if you have a strategies key in your configuration, the ViewManager will loop over these and attach them in order to the View service as listeners, at a priority of 100 (allowing them to execute before the DefaultRenderingStrategy).

180.4 Application Configuration Options

The following options may be used to provide initial configuration for the ServiceManager, ModuleManager, and Application instances, allowing them to then find and aggregate the configuration used for the Config service, which is intended for configuring all other objects in the system. These configuration directives go to the config/application.config.php file.

```
<?php
return array(
    // This should be an array of module namespaces used in the application.
    'modules' => array(
    ),

    // These are various options for the listeners attached to the ModuleManager
    'module_listener_options' => array(
        // This should be an array of paths in which modules reside.
        // If a string key is provided, the listener will consider that a module
        // namespace, the value of that key the specific path to that module's
        // Module class.
        'module_paths' => array(
        ),

```

```
        // An array of paths from which to glob configuration files after
        // modules are loaded. These effectively override configuration
        // provided by modules themselves. Paths may use GLOB_BRACE notation.
        'config_glob_paths' => array(
        ),

        // Whether or not to enable a configuration cache.
        // If enabled, the merged configuration will be cached and used in
        // subsequent requests.
        'config_cache_enabled' => $booleanValue,

        // The key used to create the configuration cache file name.
        'config_cache_key' => $stringKey,

        // Whether or not to enable a module class map cache.
        // If enabled, creates a module class map cache which will be used
        // by in future requests, to reduce the autoloading process.
        'module_map_cache_enabled' => $booleanValue,

        // The key used to create the class map cache file name.
        'module_map_cache_key' => $stringKey,

        // The path in which to cache merged configuration.
        'cache_dir' => $stringPath,

        // Whether or not to enable modules dependency checking.
        // Enabled by default, prevents usage of modules that depend on other modules
        // that weren't loaded.
        'check_dependencies' => $booleanValue,
    ),

    // Used to create an own service manager. May contain one or more child arrays.
    'service_listener_options' => array(
       array(
         'service_manager' => $stringServiceManagerName,
         'config_key'      => $stringConfigKey,
         'interface'       => $stringOptionalInterface,
         'method'          => $stringRequiredMethodName,
       ),
    )

    // Initial configuration with which to seed the ServiceManager.
    // Should be compatible with Zend\ServiceManager\Config.
    'service_manager' => array(
    ),
);
```

For an example, see the ZendSkeletonApplication configuration file.

180.5 Default Configuration Options

The following options are available when using the default services configured by the `ServiceManagerConfig` and `ViewManager`.

These configuration directives can go to the `config/autoload/{,*.}{global,local}.php` files, or in the `module/<module name>/config/module.config.php` configuration files. The merging of these config-

uration files is done by the `ModuleManager`. It first merges each module's `module.config.php` file, and then the files in `config/autoload` (first the `*.global.php` and then the `*.local.php` files). The order of the merge is relevant so you can override a module's configuration with your application configuration. If you have both a `config/autoload/my.global.config.php` and `config/autoload/my.local.config.php`, the local configuration file overrides the global configuration.

> **Warning:** Local configuration files are intended to keep sensitive information, such as database credentials, and as such, it is highly recommended to keep these local configuration files out of your VCS. The `ZendSkeletonApplication`'s `config/autoload/.gitignore` file ignores `*.local.php` files by default.

```php
<?php
return array(
    // The following are used to configure controller loader
    // Should be compatible with Zend\ServiceManager\Config.
    'controllers' => array(
        // Map of controller "name" to class
        // This should be used if you do not need to inject any dependencies
        // in your controller
        'invokables' => array(
        ),

        // Map of controller "name" to factory for creating controller instance
        // You may provide either the class name of a factory, or a PHP callback.
        'factories' => array(
        ),
    ),

    // The following are used to configure controller plugin loader
    // Should be compatible with Zend\ServiceManager\Config.
    'controller_plugins' => array(
    ),

    // The following are used to configure view helper manager
    // Should be compatible with Zend\ServiceManager\Config.
    'view_helpers' => array(
    ),

    // The following is used to configure a Zend\Di\Di instance.
    // The array should be in a format that Zend\Di\Config can understand.
    'di' => array(
    ),

    // Configuration for the Router service
    // Can contain any router configuration, but typically will always define
    // the routes for the application. See the router documentation for details
    // on route configuration.
    'router' => array(
        'routes' => array(
        ),
    ),

    // ViewManager configuration
    'view_manager' => array(
        // Base URL path to the application
        'base_path' => $stringBasePath,

```

```
        // Doctype with which to seed the Doctype helper
        'doctype' => $doctypeHelperConstantString, // e.g. HTML5, XHTML1

        // TemplateMapResolver configuration
        // template/path pairs
        'template_map' => array(
        ),

        // TemplatePathStack configuration
        // module/view script path pairs
        'template_path_stack' => array(
        ),

        // Layout template name
        'layout' => $layoutTemplateName, // e.g., 'layout/layout'

        // ExceptionStrategy configuration
        'display_exceptions' => $bool, // display exceptions in template
        'exception_template' => $stringTemplateName, // e.g. 'error'

        // RouteNotFoundStrategy configuration
        'display_not_found_reason' => $bool, // display 404 reason in template
        'not_found_template' => $stringTemplateName, // e.g. '404'

        // Additional strategies to attach
        // These should be class names or service names of View strategy classes
        // that act as ListenerAggregates. They will be attached at priority 100,
        // in the order registered.
        'strategies' => array(
            'ViewJsonStrategy', // register JSON renderer strategy
            'ViewFeedStrategy', // register Feed renderer strategy
        ),
    ),
);
```

For an example, see the Application module configuration file in the *ZendSkeletonApplication*.

CHAPTER

ONE

ROUTING

Routing is the act of matching a request to a given controller.

Typically, routing will examine the request URI, and attempt to match the URI path segment against provided constraints. If the constraints match, a set of "matches" are returned, one of which should be the controller name to execute. Routing can utilize other portions of the request URI or environment as well – for example, the host or scheme, query parameters, headers, request method, and more.

Routing has been written from the ground up for Zend Framework 2.0. Execution is quite similar, but the internal workings are more consistent, performant, and often simpler.

Note: If you are a developer with knowledge of the routing system in Zend Framework 1.x, you should know that some of the old terminology does not apply in Zend Framework 2.x. In the new routing system we don't have a router as such, as every route can match and assemble URIs by themselves, which makes them routers, too.

That said, in most cases the developer does not need to worry about this, because Zend Framework 2.x will take care of this "under the hood". The work of the router will be done by `Zend\Mvc\Router\SimpleRouteStack` or `Zend\Mvc\Router\Http\TreeRouteStack`.

The base unit of routing is a `Route`:

```
namespace Zend\Mvc\Router;

use zend\Stdlib\RequestInterface as Request;

interface RouteInterface
{
    public static function factory(array $options = array());
    public function match(Request $request);
    public function assemble(array $params = array(), array $options = array());
}
```

A `Route` accepts a `Request`, and determines if it matches. If so, it returns a `RouteMatch` object:

```
namespace Zend\Mvc\Router;

class RouteMatch
{
    public function __construct(array $params);
    public function setMatchedRouteName($name);
    public function getMatchedRouteName();
    public function setParam($name, $value);
    public function getParams();
```

```
    public function getParam($name, $default = null);
}
```

Typically, when a `Route` matches, it will define one or more parameters. These are passed into the `RouteMatch`, and objects may query the `RouteMatch` for their values.

```
$id = $routeMatch->getParam('id', false);
if (!$id) {
    throw new Exception('Required identifier is missing!');
}
$entity = $resource->get($id);
```

Usually you will have multiple routes you wish to test against. In order to facilitate this, you will use a route aggregate, usually implementing `RouteStack`:

```
namespace Zend\Mvc\Router;

interface RouteStackInterface extends RouteInterface
{
    public function addRoute($name, $route, $priority = null);
    public function addRoutes(array $routes);
    public function removeRoute($name);
    public function setRoutes(array $routes);
}
```

Typically, routes should be queried in a LIFO order, and hence the reason behind the name `RouteStack`. Zend Framework provides two implementations of this interface, `SimpleRouteStack` and `TreeRouteStack`. In each, you register routes either one at a time using `addRoute()`, or in bulk using `addRoutes()`.

```
// One at a time:
$route = Literal::factory(array(
    'route' => '/foo',
    'defaults' => array(
        'controller' => 'foo-index',
        'action'     => 'index',
    ),
));
$router->addRoute('foo', $route);

// In bulk:
$router->addRoutes(array(
    // using already instantiated routes:
    'foo' => $route,

    // providing configuration to allow lazy-loading routes:
    'bar' => array(
        'type' => 'literal',
        'options' => array(
            'route' => '/bar',
            'defaults' => array(
                'controller' => 'bar-index',
                'action'     => 'index',
            ),
        ),
    ),
));
```

181.1 Router Types

Two routers are provided, the `SimpleRouteStack` and `TreeRouteStack`. Each works with the above interface, but utilize slightly different options and execution paths. By default, the `Zend\Mvc` uses the `TreeRouteStack` as the router.

181.1.1 SimpleRouteStack

This router simply takes individual routes that provide their full matching logic in one go, and loops through them in LIFO order until a match is found. As such, routes that will match most often should be registered last, and least common routes first. Additionally, you will need to ensure that routes that potentially overlap are registered such that the most specific match will match first (i.e., register later). Alternatively, you can set priorities by giving the priority as third parameter to the `addRoute()` method, specifying the priority in the route specifications or setting the priority property within a route instance before adding it to the route stack.

181.1.2 TreeRouteStack

`Zend\Mvc\Router\Http\TreeRouteStack` provides the ability to register trees of routes, and will use a B-tree algorithm to match routes. As such, you register a single route with many children.

A `TreeRouteStack` will consist of the following configuration:

- A base "route", which describes the base match needed, the root of the tree.
- An optional "route_plugins", which is a configured `Zend\Mvc\Router\RoutePluginManager` that can lazy-load routes.
- The option "may_terminate", which hints to the router that no other segments will follow it.
- An optional "child_routes" array, which contains additional routes that stem from the base "route" (i.e., build from it). Each child route can itself be a `TreeRouteStack` if desired; in fact, the `Part` route works exactly this way.

When a route matches against a `TreeRouteStack`, the matched parameters from each segment of the tree will be returned.

A `TreeRouteStack` can be your sole route for your application, or describe particular path segments of the application.

An example of a `TreeRouteStack` is provided in the documentation of the `Part` route.

181.2 HTTP Route Types

Zend Framework 2.0 ships with the following HTTP route types.

181.2.1 Zend\Mvc\Router\Http\Hostname

The `Hostname` route attempts to match the hostname registered in the request against specific criteria. Typically, this will be in one of the following forms:

- "subdomain.domain.tld"
- ":subdomain.domain.tld"

In the above, the second route would return a "subdomain" key as part of the route match.

For any given hostname segment, you may also provide a constraint. As an example, if the "subdomain" segment needed to match only if it started with "fw" and contained exactly 2 digits following, the following route would be needed:

```
$route = Hostname::factory(array(
    'route' => ':subdomain.domain.tld',
    'constraints' => array(
        'subdomain' => 'fw\d{2}',
    ),
));
```

In the above example, only a "subdomain" key will be returned in the `RouteMatch`. If you wanted to also provide other information based on matching, or a default value to return for the subdomain, you need to also provide defaults.

```
$route = Hostname::factory(array(
    'route' => ':subdomain.domain.tld',
    'constraints' => array(
        'subdomain' => 'fw\d{2}',
    ),
    'defaults' => array(
        'type' => 'json',
    ),
));
```

When matched, the above will return two keys in the `RouteMatch`, "subdomain" and "type".

181.2.2 Zend\Mvc\Router\Http\Literal

The `Literal` route is for doing exact matching of the URI path. Configuration therefore is solely the path you want to match, and the "defaults", or parameters you want returned on a match.

```
$route = Literal::factory(array(
    'route' => '/foo',
    'defaults' => array(
        'controller' => 'Application\Controller\IndexController',
        'action' => 'foo',
    ),
));
```

The above route would match a path "/foo", and return the key "action" in the `RouteMatch`, with the value "foo".

181.2.3 Zend\Mvc\Router\Http\Method

The `Method` route is used to match the http method or 'verb' specified in the request (See RFC 2616 Sec. 5.1.1). It can optionally be configured to match against multiple methods by providing a comma-separated list of method tokens.

```
$route = Method::factory(array(
    'verb' => 'post,put',
    'defaults' => array(
        'controller' => 'Application\Controller\IndexController',
        'action' => 'form-submit',
    ),
));
```

The above route would match an http "POST" or "PUT" request and return a `RouteMatch` object containing a key "action" with a value of "form-submit".

181.2.4 Zend\Mvc\Router\Http\Part

A `Part` route allows crafting a tree of possible routes based on segments of the URI path. It actually extends the `TreeRouteStack`.

`Part` routes are difficult to describe, so we'll simply provide a sample one here.

```
$route = Part::factory(array(
    'route' => array(
        'type' => 'literal',
        'options' => array(
            'route' => '/',
            'defaults' => array(
                'controller' => 'Application\Controller\IndexController',
                'action' => 'index',
            ),
        ),
    ),
    'route_plugins' => $routePlugins,
    'may_terminate' => true,
    'child_routes' => array(
        'blog' => array(
            'type' => 'literal',
            'options' => array(
                'route' => '/blog',
                'defaults' => array(
                    'controller' => 'Application\Controller\BlogController',
                    'action' => 'index',
                ),
            ),
            'may_terminate' => true,
            'child_routes' => array(
                'rss' => array(
                    'type' => 'literal',
                    'options' => array(
                        'route' => '/rss',
                        'defaults' => array(
                            'action' => 'rss',
                        )
                    ),
                    'may_terminate' => true,
                    'child_routes' => array(
                        'subrss' => array(
                            'type' => 'literal',
                            'options' => array(
                                'route' => '/sub',
                                'defaults' => array(
                                    'action' => 'subrss',
                                ),
                            ),
                        ),
                    ),
                ),
            ),
        ),
```

```
        'forum' => array(
            'type' => 'literal',
            'options' => array(
                'route' => 'forum',
                'defaults' => array(
                    'controller' => 'Application\Controller\ForumController',
                    'action' => 'index',
                ),
            ),
        ),
    ),
));
```

The above would match the following:

- "/" would load the "Index" controller, "index" action.
- "/blog" would load the "Blog" controller, "index" action.
- "/blog/rss" would load the "Blog" controller, "rss" action.
- "/blog/rss/sub" would load the "Blog" controller, "subrss" action.
- "/forum" would load the "Forum" controller, "index" action.

You may use any route type as a child route of a `Part` route.

Note: `Part` routes are not meant to be used directly. When you add definitions for `child_routes` to any route type, that route will become a `Part` route. As already said, describing `Part` routes with words is difficult, so hopefully the additional *examples at the end* will provide further insight.

Note: In the above example, the `$routePlugins` is an instance of `Zend\Mvc\Router\RoutePluginManager`.

```
$routePlugins = new Zend\Mvc\Router\RoutePluginManager();
$plugins = array(
    'hostname' => 'Zend\Mvc\Router\Http\Hostname',
    'literal'  => 'Zend\Mvc\Router\Http\Literal',
    'part'     => 'Zend\Mvc\Router\Http\Part',
    'regex'    => 'Zend\Mvc\Router\Http\Regex',
    'scheme'   => 'Zend\Mvc\Router\Http\Scheme',
    'segment'  => 'Zend\Mvc\Router\Http\Segment',
    'wildcard' => 'Zend\Mvc\Router\Http\Wildcard',
    'query'    => 'Zend\Mvc\Router\Http\Query',
    'method'   => 'Zend\Mvc\Router\Http\Method',
);
$foreach ($plugins as $name => $class) {
    $routePlugins->setInvokableClass($name, $class);
}
```

When using `Zend\Mvc\Router\Http\TreeRouteStack`, the `RoutePluginManager` is set up by default, and the developer does not need to worry about the autoloading of standard HTTP routes.

181.2.5 Zend\Mvc\Router\Http\Regex

A `Regex` route utilizes a regular expression to match against the URI path. Any valid regular expression is allowed; our recommendation is to use named captures for any values you want to return in the `RouteMatch`.

Since regular expression routes are often complex, you must specify a "spec" or specification to use when assembling URLs from regex routes. The spec is simply a string; replacements are identified using "%keyname%" within the string, with the keys coming from either the captured values or named parameters passed to the `assemble()` method.

Just like other routes, the `Regex` route can accept "defaults", parameters to include in the `RouteMatch` when successfully matched.

```
$route = Regex::factory(array(
    'regex' => '/blog/(?<id>[a-zA-Z0-9_-]+)(\.(?<format>(json|html|xml|rss)))?',
    'defaults' => array(
        'controller' => 'Application\Controller\BlogController',
        'action'     => 'view',
        'format'     => 'html',
    ),
    'spec' => '/blog/%id%.%format%',
));
```

The above would match "/blog/001-some-blog_slug-here.html", and return four items in the `RouteMatch`, an "id", the "controller", the "action", and the "format". When assembling a URL from this route, the "id" and "format" values would be used to fill the specification.

181.2.6 Zend\Mvc\Router\Http\Scheme

The `Scheme` route matches the URI scheme only, and must be an exact match. As such, this route, like the `Literal` route, simply takes what you want to match and the "defaults", parameters to return on a match.

```
$route = Scheme::factory(array(
    'scheme' => 'https',
    'defaults' => array(
        'https' => true,
    ),
));
```

The above route would match the "https" scheme, and return the key "https" in the `RouteMatch` with a boolean `true` value.

181.2.7 Zend\Mvc\Router\Http\Segment

A `Segment` route allows matching any segment of a URI path. Segments are denoted using a colon, followed by alphanumeric characters; if a segment is optional, it should be surrounded by brackets. As an example, "/:foo[/:bar]" would match a "/" followed by text and assign it to the key "foo"; if any additional "/" characters are found, any text following the last one will be assigned to the key "bar".

The separation between literal and named segments can be anything. For example, the above could be done as "/:foo{-}[-:bar] as well. The {-} after the :foo parameter indicates a set of one or more delimiters, after which matching of the parameter itself ends.

Each segment may have constraints associated with it. Each constraint should simply be a regular expression expressing the conditions under which that segment should match.

Also, as you can in other routes, you may provide defaults to use; these are particularly useful when using optional segments.

As a complex example:

```
$route = Segment::factory(array(
    'route' => '/:controller[/:action]',
    'constraints' => array(
        'controller' => '[a-zA-Z][a-zA-Z0-9_-]+',
        'action'     => '[a-zA-Z][a-zA-Z0-9_-]+',
    ),
    'defaults' => array(
        'controller' => 'Application\Controller\IndexController',
        'action'     => 'index',
    ),
));
```

181.2.8 Zend\Mvc\Router\Http\Query (Deprecated)

This route part is deprecated since you can now add query parameters without a query route.

The `Query` route part allows you to specify and capture query string parameters for a given route.

The intention of the `Query` part is that you do not instantiate it in its own right but to use it as a child of another route part.

An example of its usage would be

```
$route = Part::factory(array(
    'route' => array(
        'type'    => 'literal',
        'options' => array(
            'route'    => 'page',
            'defaults' => array(
            ),
        ),
    ),
    'may_terminate' => true,
    'route_plugins' => $routePlugins,
    'child_routes'  => array(
        'query' => array(
            'type' => 'Query',
            'options' => array(
                'defaults' => array(
                    'foo' => 'bar',
                ),
            ),
        ),
    ),
));
```

As you can see, it's pretty straight forward to specify the query part. This then allows you to create query strings using the url view helper.

```
$this->url(
    'page/query',
    array(
        'name'   => 'my-test-page',
        'format' => 'rss',
        'limit'  => 10,
    )
);
```

As you can see above, you must add "/query" to your route name in order to append a query string. If you do not specify "/query" in the route name then no query string will be appended.

Our example "page" route has only one defined parameter of "name" ("/page[/:name]"), meaning that the remaining parameters of "format" and "limit" will then be appended as a query string.

The output from our example should then be "/page/my-test-page?format=rss&limit=10"

181.3 HTTP Routing Examples

Most of the routing definitions will be done in module configuration files, so the following examples will show how to set up routes in config files.

Simple example with two literal routes

```
return array(
    'router' => array(
        'routes' => array(
            // Literal route named "home"
            'home' => array(
                'type' => 'literal',
                'options' => array(
                    'route' => '/',
                    'defaults' => array(
                        'controller' => 'Application\Controller\IndexController',
                        'action' => 'index',
                    ),
                ),
            ),
            // Literal route named "contact"
            'contact' => array(
                'type' => 'literal',
                'options' => array(
                    'route' => 'contact',
                    'defaults' => array(
                        'controller' => 'Application\Controller\ContactController',
                        'action' => 'form',
                    ),
                ),
            ),
        ),
    ),
);
```

A complex example with child routes

```
return array(
    'router' => array(
        'routes' => array(
            // Literal route named "home"
            'home' => array(
                'type' => 'literal',
                'options' => array(
                    'route' => '/',
```

```
                    'defaults' => array(
                        'controller' => 'Application\Controller\IndexController',
                        'action' => 'index',
                    ),
                ),
            ),
            // Literal route named "blog", with child routes
            'blog' => array(
                'type' => 'literal',
                'options' => array(
                    'route' => '/blog',
                    'defaults' => array(
                        'controller' => 'Application\Controller\BlogController',
                        'action' => 'index',
                    ),
                ),
                'may_terminate' => true,
                'child_routes' => array(
                    // Segment route for viewing one blog post
                    'post' => array(
                        'type' => 'segment',
                        'options' => array(
                            'route' => '/[:slug]',
                            'constraints' => array(
                                'slug' => '[a-zA-Z0-9_-]+',
                            ),
                            'defaults' => array(
                                'action' => 'view',
                            ),
                        ),
                    ),
                    // Literal route for viewing blog RSS feed
                    'rss' => array(
                        'type' => 'literal',
                        'options' => array(
                            'route' => '/rss',
                            'defaults' => array(
                                'action' => 'rss',
                            ),
                        ),
                    ),
                ),
            ),
        ),
    ),
);
```

When using child routes, naming of the routes follows the `parent/child` pattern, so to use the child routes from the above example:

```
echo $this->url('blog'); // gives "/blog"
echo $this->url('blog/post', array('slug' => 'my-post')); // gives "/blog/my-post"
echo $this->url('blog/rss'); // gives "/blog/rss"
```

> **Warning:** When defining child routes pay attention that the `may_terminate` and `child_routes` definitions are in same level as the `options` and `type` definitions. A common pitfall is to have those two definitions nested in `options`, which will not result in the desired routes.

181.4 Console Route Types

Zend Framework 2.0 also comes with routes for writing Console based applications, which is explained in the *Console routes and routing* section.

CHAPTER

TWO

THE MVCEVENT

The MVC layer of Zend Framework 2 incorporates and utilizes a custom `Zend\EventManager\Event` implementation - `Zend\Mvc\MvcEvent`. This event is created during `Zend\Mvc\Application::bootstrap()` and is passed directly to all the events that method triggers. Additionally, if your controllers implement the `Zend\Mvc\InjectApplicationEventInterface`, `MvcEvent` will be injected into those controllers.

The `MvcEvent` adds accessors and mutators for the following:

- `Application` object.
- `Request` object.
- `Response` object.
- `Router` object.
- `RouteMatch` object.
- Result - usually the result of dispatching a controller.
- `ViewModel` object, typically representing the layout view model.

The methods it defines are:

- `setApplication($application)`
- `getApplication()`
- `setRequest($request)`
- `getRequest()`
- `setResponse($response)`
- `getResponse()`
- `setRouter($router)`
- `getRouter()`
- `setRouteMatch($routeMatch)`
- `getRouteMatch()`
- `setResult($result)`
- `getResult()`
- `setViewModel($viewModel)`
- `getViewModel()`
- `isError()`

- `setError()`
- `getError()`
- `getController()`
- `setController($name)`
- `getControllerClass()`
- `setControllerClass($class)`

The `Application`, `Request`, `Response`, `Router`, and `ViewModel` are all injected during the `bootstrap` event. Following the `route` event, it will be injected also with the `RouteMatch` object encapsulating the results of routing.

Since this object is passed around throughout the MVC, it is a common location for retrieving the results of routing, the router, and the request and response objects. Additionally, we encourage setting the results of execution in the event, to allow event listeners to introspect them and utilize them within their execution. As an example, the results could be passed into a view renderer.

182.1 Order of events

The following events are triggered, in the following order:

Table 182.1: MvcEvent Events

Name	Constant	Description
`bootstrap`	`MvcEvent::EVENT_BOOTSTRAP`	Bootstrap the application by creating the ViewManager.
`route`	`MvcEvent::EVENT_ROUTE`	Perform all the route work (matching...).
`dispatch`	`MvcEvent::EVENT_DISPATCH`	Dispatch the matched route to a controller/action.
`dispatch.error`	`MvcEvent::EVENT_DISPATCH_ERROR`	Event triggered in case of a problem during dispatch process (unknown controller...).
`render`	`MvcEvent::EVENT_RENDER`	Prepare the data and delegate the rendering to the view layer.
`render.error`	`MvcEvent::EVENT_RENDER_ERROR`	Event triggered in case of a problem during the render process (no renderer found...).
`finish`	`MvcEvent::EVENT_FINISH`	Perform any task once everything is done.

Those events are extensively describe in the following sections.

182.2 MvcEvent::EVENT_BOOTSTRAP

182.2.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Table 182.2: MvcEvent::EVENT_BOOTSTRAP Listeners

Class	Priority	Method Called	Itself Triggers	Description
`Zend\Mvc\View\Http\ViewManager`	10000	`onBootstrap`	*none*	Prepares the view layer (instantiate a `Zend\Mvc\View\Http\ViewManager`).

182.2.2 Triggerers

This event is triggered by the following classes:

Table 182.3: MvcEvent::EVENT_BOOTSTRAP Triggerers

Class	In Method
`Zend\Mvc\Application`	`bootstrap`

182.3 MvcEvent::EVENT_ROUTE

182.3.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Table 182.4: MvcEvent::EVENT_ROUTE Listeners

Class	Priority	Method Called	Itself Triggers	Description
`Zend\Mvc\ModuleRouteListener`	1	`onRoute`	*none*	This listener determines if the module namespace should be prepended to the controller name. This is the case if the route match contains a parameter key matching the MODULE_NAMESPACE constant.
`Zend\Mvc\RouteListener`	1	`onRoute`	`MvcEvent::EVENT_DISPATCH_ERROR` (if no route is matched)	This a route request to the router and return a RouteMatch object.

182.3.2 Triggerers

This event is triggered by the following classes:

Table 182.5: MvcEvent::EVENT_ROUTE Triggerers

Class	In Method	Description
`Zend\Mvc\Application`	`run`	It also has a short circuit callback that allows to stop the propagation of the event if an error is raised during the routing.

182.4 MvcEvent::EVENT_DISPATCH

182.4.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Console context only

Those listeners are only attached in a Console context:

Table 182.6: MvcEvent::EVENT_DISPATCH Listeners for Console context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Console\`	100	`injectNamedConsole`	Merge all the params (route matched params and params in the command) and add them to the Request object.
`Zend\Mvc\View\Console\`	80	`createViewModelFromArray`	If the controller action returned an associative array, it casts it to a `ConsoleModel` object.
`Zend\Mvc\View\Console\`	80	`createViewModelFromString`	If the controller action returned a string, it casts it to a `ConsoleModel` object.
`Zend\Mvc\View\Console\`	80	`createViewModelFromNull`	If the controller action returned null, it casts it to a `ConsoleModel` object.
`Zend\Mvc\View\Console\`	`Injector` 100	`injectViewModel`	Inserts the `ViewModel` (in this case, a `ConsoleModel`) and adds it to the MvcEvent object. It either (a) adds it as a child to the default, composed view model, or (b) replaces it if the result is marked as terminable.

Http context only

Those listeners are only attached in a Http context:

Table 182.7: MvcEvent::EVENT_DISPATCH Listeners for Http context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Http\`	80	`createViewModelFromArray`	If the controller action returned an associative array, it casts it to a `ViewModel` object.
`Zend\Mvc\View\Http\`	80	`createViewModelFromNull`	If the controller action returned null, it casts it to a `ViewModel` object.
`Zend\Mvc\View\Http\`	90	`routeNotFoundStrategy`	If a strategy was not found, return a 404 `ViewModel`.
`Zend\Mvc\View\Http\`	90	`injectTemplate`	Inject a template into the view model, if none present. Template is derived from the controller found in the route match, and, optionally, the action, if present.
`Zend\Mvc\View\Http\`	`Injector` 100	`injectViewModel`	Inserts the `ViewModel` (in this case, a `ViewModel`) and adds it to the MvcEvent object. It either (a) adds it as a child to the default, composed view model, or (b) replaces it if the result is marked as terminable.

All contexts

Those listeners are attached for both contexts:

Table 182.8: MvcEvent::EVENT_DISPATCH Listeners for both contexts

Class	Priority	Method Called	Itself Triggers	Description
`Zend\Mvc\Dispatchlistener`	1	`onDispatch`	`MvcEvent::EVENT_DISPATCH_ERROR` (if an exception is raised during dispatch processes)	Try to load the matched controller from the service manager (and throws various exceptions if it does not).
`Zend\Mvc\AbstractController`	1	`onDispatch`	*none*	The `onDispatch` method of the `AbstractController` is an abstract method. In `AbstractActionController` for instance, it simply calls the action method.

182.4.2 Triggerers

This event is triggered by the following classes:

Table 182.9: MvcEvent::EVENT_DISPATCH Triggerers

Class	In Method	Description
`Zend\Mvc\Application`	`run`	It also has a short circuit callback that allows to stop the propagation of the event if an error is raised during the routing.
`Zend\Mvc\Controller\AbstractController`	`dispatch`	If a listener returns a `Response` object, it stops propagation. Note: every `AbstractController` listen to this event and execute the `onBootstrap` method when it is triggered.

182.5 MvcEvent::EVENT_DISPATCH_ERROR

182.5.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Console context only

Table 182.10: MvcEvent::EVENT_DISPATCH_ERROR Listeners for Console context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Console\RouteNotFoundStrategy`	1	`prepareNotFoundViewModel`	Detect if an error is a route not found condition. If a "controller not found" or "invalid controller" error type is encountered, sets the response status code to 404.
`Zend\Mvc\View\Console\ExceptionStrategy`	1	`prepareExceptionViewModel`	Create an exception view model and set the status code to 404.
`Zend\Mvc\View\Console\InjectViewModelListener`	-100	`injectViewModel`	Inject the `ViewModel` (in this case, a `ConsoleModel`) and adds it to the MvcEvent object. It either (a) adds it as a child to the default, composed view model, or (b) replaces it if the result is marked as terminable.

Http context only

Those listeners are only attached in a Http context:

Table 182.11: MvcEvent::EVENT_DISPATCH_ERROR Listeners for Http context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Http\RouteNotFoundStrategy`	1	`detectNotFoundError`	Detects if an error is a 404 condition. If a "controller not found" or "invalid controller" error type is encountered, sets the response status code to 404.
`Zend\Mvc\View\Http\RouteNotFoundStrategy`	1	`prepareNotFoundViewModel`	Creates and returns a 404 view model.
`Zend\Mvc\View\Http\ExceptionStrategy`	1	`prepareExceptionViewModel`	Creates an exception view model and set the status code to 404
`Zend\Mvc\View\Http\InjectViewModelListener`	-100	`injectViewModel`	Injects the `ViewModel` (in this case, a `ViewModel`) and adds it to the MvcEvent object. It either (a) adds it as a child to the default, composed view model, or (b) replaces it if the result is marked as terminable.

All contexts

Those listeners are attached for both contexts:

Table 182.12: MvcEvent::EVENT_DISPATCH_ERROR Listeners for both contexts

Class	Priority	Method Called	Description
`Zend\Mvc\DispatchListener`	1	`reportMonitorEvent`	Used to monitoring when Zend Server is used.

182.5.2 Triggerers

Table 182.13: MvcEvent::EVENT_DISPATCH_ERROR Triggerers

Class	In Method
`Zend\Mvc\DispatchListener`	`onDispatch`
`Zend\Mvc\DispatchListener`	`marshallControllerNotFoundEvent`
`Zend\Mvc\DispatchListener`	`marshallBadControllerEvent`

182.6 MvcEvent::EVENT_RENDER

182.6.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Console context only

Those listeners are only attached in a Console context:

Table 182.14: MvcEvent::EVENT_RENDER Listeners for Console context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Console\DefaultRenderingStrategy`	-10000	`render`	Render the view.

Http context only

Those listeners are only attached in a Http context:

Table 182.15: MvcEvent::EVENT_RENDER Listeners for Http context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Http\DefaultRenderingStrategy`	-10000	`render`	Render the view.

182.6.2 Triggerers

This event is triggered by the following classes:

Table 182.16: MvcEvent::EVENT_RENDER Triggerers

Class	In Method	Description
`Zend\Mvc\Application`	`completeRequest`	This event is triggered just before the MvcEvent::FINISH event.

182.7 MvcEvent::EVENT_RENDER_ERROR

182.7.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Console context only

Those listeners are only attached in a Console context:

Table 182.17: MvcEvent::EVENT_RENDER_ERROR Listeners for Console context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Console\ExceptionStrategy`	1	`prepareExceptionViewModel`	Create an exception view model and set the status code to 404.
`Zend\Mvc\View\Console\InjectViewModelListener`	-100	`injectViewModel`	Inject the `ViewModel` (in this case, a `ConsoleModel`) and adds it to the MvcEvent object. It either (a) adds it as a child to the default, composed view model, or (b) replaces it if the result is marked as terminable.

Http context only

Those listeners are only attached in a Http context:

Table 182.18: MvcEvent::EVENT_RENDER_ERROR Listeners for Http context only

Class	Priority	Method Called	Description
`Zend\Mvc\View\Console\ExceptionStrategy`	1	`prepareExceptionViewModel`	Creates an exception view model and set the status code to 404.
`Zend\Mvc\View\Console\InjectViewModelListener`	-100	`injectViewModel`	Injects the `ViewModel` (in this case, a `ViewModel`) and adds it to the MvcEvent object. It either (a) adds it as a child to the default, composed view model, or (b) replaces it if the result is marked as terminable.

182.7.2 Triggerers

This event is triggered by the following classes:

Table 182.19: MvcEvent::EVENT_RENDER_ERROR Triggerers

Class	In Method	Description
`Zend\Mvc\View\Http\DefaultRenderingStrategy`	`render`	This event is triggered if an exception is raised during rendering.

182.8 MvcEvent::EVENT_FINISH

182.8.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Table 182.20: MvcEvent::EVENT_FINISH Listeners

Class	Priority	Method Called	Description
`Zend\Mvc\SendResponseListener`	10000	`sendResponse`	It triggers the `SendResponseEvent` in order to prepare the response (see the next page for more information about `SendResponseEvent`).

182.8.2 Triggerers

This event is triggered by the following classes:

Table 182.21: MvcEvent::EVENT_FINISH Triggerers

Class	In Method	Description
`Zend\Mvc\Application`	`run`	This event is triggered once the MvcEvent::ROUTE event returns a correct `ResponseInterface`.
`Zend\Mvc\Application`	`run`	This event is triggered once the MvcEvent::DISPATCH event returns a correct `ResponseInterface`.
`Zend\Mvc\Application`	`completeRequest`	This event is triggered after the MvcEvent::RENDER (this means that, at this point, the view is already rendered).

CHAPTER

THREE

THE SENDRESPONSEEVENT

The MVC layer of Zend Framework 2 also incorporates and utilizes a custom `Zend\EventManager\Event` implementation located at `Zend\Mvc\ResponseSender\SendResponseEvent`. This event allows listeners to update the response object, by setting headers and content.

The methods it defines are:

- `setResponse($response)`
- `getResponse()`
- `setContentSent()`
- `contentSent()`
- `setHeadersSent()`
- `headersSent()`

183.1 Listeners

Currently, three listeners are listening to this event at different priorities based on which listener is used most.

Table 183.1: SendResponseEvent Listeners

Class	Priority	Method Called	Description
`Zend\Mvc\SendResponseListener\PhpEnvironmentResponseSender`	-1000	`sendContent`	This is used in context of HTTP (this is the most often used).
`Zend\Mvc\SendResponseListener\ConsoleResponseSender`	-2000	`sendContent`	This is used in context of Console.
`Zend\Mvc\SendResponseListener\SimpleStreamResponseSender`	-3000	`sendContent`	

Because all these listeners have negative priorities, adding your own logic to modify `Response` object is easy: just add a new listener without any priority (it will default to 1) and it will always be executed first.

183.2 Triggerers

This event is executed when MvcEvent::FINISH event is triggered, with a priority of -10000.

CHAPTER

FOUR

AVAILABLE CONTROLLERS

Controllers in the MVC layer simply need to be objects implementing `Zend\Stdlib\DispatchableInterface`. That interface describes a single method:

```
use Zend\Stdlib\DispatchableInterface;
use Zend\Stdlib\RequestInterface as Request;
use Zend\Stdlib\ResponseInterface as Response;

class Foo implements DispatchableInterface
{
    public function dispatch(Request $request, Response $response = null)
    {
        // ... do something, and preferably return a Response ...
    }
}
```

While this pattern is simple enough, chances are you don't want to implement custom dispatch logic for every controller (particularly as it's not unusual or uncommon for a single controller to handle several related types of requests).

The MVC also defines several interfaces that, when implemented, can provide controllers with additional capabilities.

184.1 Common Interfaces Used With Controllers

184.1.1 InjectApplicationEvent

The `Zend\Mvc\InjectApplicationEventInterface` hints to the `Application` instance that it should inject its `MvcEvent` into the controller itself. Why would this be useful?

Recall that the `MvcEvent` composes a number of objects: the `Request` and `Response`, naturally, but also the router, the route matches (a `RouteMatch` instance), and potentially the "result" of dispatching.

A controller that has the `MvcEvent` injected, then, can retrieve or inject these. As an example:

```
$matches = $this->getEvent()->getRouteMatch();
$id      = $matches->getParam('id', false);
if (!$id) {
    $response = $this->getResponse();
    $response->setStatusCode(500);
    $this->getEvent()->setResult('Invalid identifier; cannot complete request');
    return;
}
```

The `InjectApplicationEventInterface` defines simply two methods:

```
public function setEvent(Zend\EventManager\EventInterface $event);
public function getEvent();
```

184.1.2 ServiceLocatorAware

In most cases, you should define your controllers such that dependencies are injected by the application's `ServiceManager`, via either constructor arguments or setter methods.

However, occasionally you may have objects you wish to use in your controller that are only valid for certain code paths. Examples include forms, paginators, navigation, etc. In these cases, you may decide that it doesn't make sense to inject those objects every time the controller is used.

The `ServiceLocatorAwareInterface` interface hints to the `ServiceManager` that it should inject itself into the controller. It defines two simple methods:

```
use Zend\ServiceManager\ServiceLocatorInterface;
use Zend\ServiceManager\ServiceLocatorAwareInterface;

public function setServiceLocator(ServiceLocatorInterface $serviceLocator);
public function getServiceLocator();
```

184.1.3 EventManagerAware

Typically, it's nice to be able to tie into a controller's workflow without needing to extend it or hardcode behavior into it. The solution for this at the framework level is to use the `EventManager`.

You can hint to the `ServiceManager` that you want an `EventManager` injected by implementing the interface `EventManagerAwareInterface`, which tells the `ServiceManager` to inject an `EventManager`.

You define two methods. The first, a setter, should also set any `EventManager` identifiers you want to listen on, and the second, a getter, should simply return the composed `EventManager` instance.

```
use Zend\EventManager\EventManagerAwareInterface;
use Zend\EventManager\EventManagerInterface;

public function setEventManager(EventManagerInterface $events);
public function getEventManager();
```

184.1.4 Controller Plugins

Code re-use is a common goal for developers. Another common goal is convenience. However, this is often difficult to achieve cleanly in abstract, general systems.

Within your controllers, you'll often find yourself repeating tasks from one controller to another. Some common examples:

- Generating URLs
- Redirecting
- Setting and retrieving flash messages (self-expiring session messages)
- Invoking and dispatching additional controllers

To facilitate these actions while also making them available to alternate controller implementations, we've created a `PluginManager` implementation for the controller layer, `Zend\Mvc\Controller\PluginManager`, building on the `Zend\ServiceManager\AbstractPluginManager` functionality. To utilize it, you simply need

to implement the `setPluginManager(PluginManager $plugins)` method, and set up your code to use the controller-specific implementation by default:

```
use Zend\Mvc\Controller\PluginManager;

public function setPluginManager(PluginManager $plugins)
{
    $this->plugins = $plugins;
    $this->plugins->setController($this);

    return $this;
}

public function getPluginManager()
{
    if (!$this->plugins) {
        $this->setPluginManager(new PluginManager());
    }

    return $this->plugins;
}

public function plugin($name, array $options = null)
{
    return $this->getPluginManager()->get($name, $options);
}
```

184.2 The AbstractActionController

Implementing each of the above interfaces is a lesson in redundancy; you won't often want to do it. As such, we've developed two abstract, base controllers you can extend to get started.

The first is `Zend\Mvc\Controller\AbstractActionController`. This controller implements each of the above interfaces, and uses the following assumptions:

- An "action" parameter is expected in the `RouteMatch` object composed in the attached `MvcEvent`. If none is found, a `notFoundAction()` is invoked.
- The "action" parameter is converted to a camelCased format and appended with the word "Action" to create a method name. As examples: "foo" maps to "fooAction", "foo-bar" or "foo.bar" or "foo_bar" to "fooBarAction". The controller then checks to see if that method exists. If not, the `notFoundAction()` method is invoked; otherwise, the discovered method is called.
- The results of executing the given action method are injected into the `MvcEvent`'s "result" property (via `setResult()`, and accessible via `getResult()`).

Essentially, a route mapping to an `AbstractActionController` needs to return both "controller" and "action" keys in its matches.

Creation of action controllers is then reasonably trivial:

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class BarController extends AbstractActionController
{
    public function bazAction()
```

```
    {
        return array('title' => __METHOD__);
    }

    public function batAction()
    {
        return array('title' => __METHOD__);
    }
}
```

184.2.1 Interfaces and Collaborators

`AbstractActionController` implements each of the following interfaces:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Mvc\InjectApplicationEventInterface`
- `Zend\ServiceManager\ServiceLocatorAwareInterface`
- `Zend\EventManager\EventManagerAwareInterface`

The composed `EventManager` will be configured to listen on the following contexts:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Mvc\Controller\AbstractActionController`
- `Zend\Mvc\Controller\AbstractController`

Additionally, if you extend the class, it will listen on the extending class's name.

184.3 The AbstractRestfulController

The second abstract controller ZF2 provides is `Zend\Mvc\Controller\AbstractRestfulController`. This controller provides a native RESTful implementation that simply maps HTTP request methods to controller methods, using the following matrix:

- **GET** maps to either `get()` or `getList()`, depending on whether or not an "id" parameter is found in the route matches. If one is, it is passed as an argument to `get()`; if not, `getList()` is invoked. In the former case, you should provide a representation of the given entity with that identification; in the latter, you should provide a list of entities.
- **POST** maps to `create()`. That method expects a `$data` argument, usually the `$_POST` superglobal array. The data should be used to create a new entity, and the response should typically be an HTTP 201 response with the Location header indicating the URI of the newly created entity and the response body providing the representation.
- **PUT** maps to `update()`, and requires that an "id" parameter exists in the route matches; that value is passed as an argument to the method. It should attempt to update the given entity, and, if successful, return either a 200 or 202 response status, as well as the representation of the entity.
- **DELETE** maps to `delete()`, and requires that an "id" parameter exists in the route matches; that value is passed as an argument to the method. It should attempt to delete the given entity, and, if successful, return either a 200 or 204 response status.

Additionally, you can map "action" methods to the `AbstractRestfulController`, just as you would in the `AbstractActionController`; these methods will be suffixed with "Action", differentiating them from the

RESTful methods listed above. This allows you to perform such actions as providing forms used to submit to the various RESTful methods, or to add RPC methods to your RESTful API.

184.3.1 Interfaces and Collaborators

`AbstractRestfulController` implements each of the following interfaces:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Mvc\InjectApplicationEventInterface`
- `Zend\ServiceManager\ServiceLocatorAwareInterface`
- `Zend\EventManager\EventManagerAwareInterface`

The composed `EventManager` will be configured to listen on the following contexts:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Mvc\Controller\AbstractRestfulController`
- `Zend\Mvc\Controller\AbstractController`

Additionally, if you extend the class, it will listen on the extending class's name.

CHAPTER

FIVE

CONTROLLER PLUGINS

When using the `AbstractActionController` or `AbstractRestfulController`, or if you implement the `setPluginManager` method in your custom controllers, you have access to a number of pre-built plugins. Additionally, you can register your own custom plugins with the manager.

The built-in plugins are:

- *Zend\Mvc\Controller\Plugin\AcceptableViewModelSelector*
- *Zend\Mvc\Controller\Plugin\FlashMessenger*
- *Zend\Mvc\Controller\Plugin\Forward*
- *Zend\Mvc\Controller\Plugin\Identity*
- *Zend\Mvc\Controller\Plugin\Layout*
- *Zend\Mvc\Controller\Plugin\Params*
- *Zend\Mvc\Controller\Plugin\PostRedirectGet*
- *Zend\Mvc\Controller\Plugin\Redirect*
- *Zend\Mvc\Controller\Plugin\Url*

If your controller implements the `setPluginManager`, `getPluginManager` and `plugin` methods, you can access these using their shortname via the `plugin()` method:

```
$plugin = $this->plugin('url');
```

For an extra layer of convenience, both `AbstractActionController` and `AbstractRestfulController` have `__call()` implementations that allow you to retrieve plugins via method calls:

```
$plugin = $this->url();
```

185.1 AcceptableViewModelSelector Plugin

The `AcceptableViewModelSelector` is a helper that can be used to select an appropriate view model based on user defined criteria will be tested against the Accept header in the request.

As an example:

```
use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\JsonModel;

class SomeController extends AbstractActionController
```

```
{
   protected $acceptCriteria = array(
      'Zend\View\Model\JsonModel' => array(
         'application/json',
      ),
      'Zend\View\Model\FeedModel' => array(
         'application/rss+xml',
      ),
   );

   public function apiAction()
   {
      $viewModel = $this->acceptableViewModelSelector($this->acceptCriteria);

      // Potentially vary execution based on model returned
      if ($viewModel instanceof JsonModel) {
         // ...
      }
   }
}
```

The above would return a standard `Zend\View\Model\ViewModel` instance if the criteria is not met, and the specified view model types if the specific criteria is met. Rules are matched in order, with the first match "winning."

185.2 FlashMessenger Plugin

The `FlashMessenger` is a plugin designed to create and retrieve self-expiring, session-based messages. It exposes a number of methods:

setSessionManager(*Zend\Session\ManagerInterface $manager*)
Allows you to specify an alternate session manager, if desired.

Return type `Zend\Mvc\Controller\Plugin\FlashMessenger`

getSessionManager()
Allows you to retrieve the session manager registered.

Return type `Zend\Session\ManagerInterface`

getContainer()
Returns the `Zend\Session\Container` instance in which the flash messages are stored.

Return type `Zend\Session\Container`

setNamespace(*string $namespace = 'default'*)
Allows you to specify a specific namespace in the container in which to store or from which to retrieve flash messages.

Return type `Zend\Mvc\Controller\Plugin\FlashMessenger`

- `getNamespace()` retrieves the name of the flash message namespace.

getNamespace()
Retrieves the name of the flash message namespace.

Return type `string`

addMessage(*string $message*)
Allows you to add a message to the current namespace of the session container.

Return type `Zend\Mvc\Controller\Plugin\FlashMessenger`

hasMessages()
Lets you determine if there are any flash messages from the current namespace in the session container.

Return type `boolean`

getMessages()
Retrieves the flash messages from the current namespace of the session container

Return type `array`

clearMessages()
Clears all flash messages in current namespace of the session container. Returns `true` if messages were cleared, `false` if none existed.

Return type `boolean`

hasCurrentMessages()
Indicates whether any messages were added during the current request.

Return type `boolean`

getCurrentMessages()
Retrieves any messages added during the current request.

Return type `array`

clearCurrentMessages()
Removes any messages added during the current request. Returns `true` if current messages were cleared, `false` if none existed.

Return type `boolean`

Additionally, the `FlashMessenger` implements both `IteratorAggregate` and `Countable`, allowing you to iterate over and count the flash messages in the current namespace within the session container.

Examples

```
public function processAction()
{
    // ... do some work ...
    $this->flashMessenger()->addMessage('You are now logged in.');
    return $this->redirect()->toRoute('user-success');
}

public function successAction()
{
    $return = array('success' => true);
    $flashMessenger = $this->flashMessenger();
    if ($flashMessenger->hasMessages()) {
        $return['messages'] = $flashMessenger->getMessages();
    }
    return $return;
}
```

185.3 Forward Plugin

Occasionally, you may want to dispatch additional controllers from within the matched controller – for instance, you might use this approach to build up "widgetized" content. The `Forward` plugin helps enable this.

For the `Forward` plugin to work, the controller calling it must be `ServiceLocatorAware`; otherwise, the plugin will be unable to retrieve a configured and injected instance of the requested controller.

The plugin exposes a single method, `dispatch()`, which takes two arguments:

- `$name`, the name of the controller to invoke. This may be either the fully qualified class name, or an alias defined and recognized by the `ServiceManager` instance attached to the invoking controller.
- `$params` is an optional array of parameters with which to see a `RouteMatch` object for purposes of this specific request.

`Forward` returns the results of dispatching the requested controller; it is up to the developer to determine what, if anything, to do with those results. One recommendation is to aggregate them in any return value from the invoking controller.

As an example:

```
$foo = $this->forward()->dispatch('foo', array('action' => 'process'));
return array(
    'somekey' => $somevalue,
    'foo'     => $foo,
);
```

185.4 Identity Plugin

The `Identity` plugin allows for getting the identity from the `AuthenticationService`.

For the `Identity` plugin to work, a `Zend\Authentication\AuthenticationService` name or alias must be defined and recognized by the `ServiceManager`.

`Identity` returns the identity in the `AuthenticationService` or *null* if no identity is available.

As an example:

```
public function testAction()
{
    if ($user = $this->identity()) {
         // someone is logged !
    } else {
         // not logged in
    }
}
```

When invoked, the `Identity` plugin will look for a service by the name or alias `Zend\Authentication\AuthenticationService` in the `ServiceManager`. You can provide this service to the `ServiceManager` in a configuration file:

```
// In a configuration file...
return array(
    'service_manager' => array(
        'alias' => array(
            'Zend\Authentication\AuthenticationService' => 'my_auth_service',
        ),
```

```
        'invokables' => array(
            'my_auth_service' => 'Zend\Authentication\AuthenticationService',
        ),
    ),
);
```

The `Identity` plugin exposes two methods:

setAuthenticationService(*Zend\Authentication\AuthenticationService $authenticationService*)
Sets the authentication service instance to be used by the plugin.

Return type `void`

getAuthenticationService()
Retrieves the current authentication service instance if any is attached.

Return type `Zend\Authentication\AuthenticationService`

185.5 Layout Plugin

The `Layout` plugin allows for changing layout templates from within controller actions.

It exposes a single method, `setTemplate()`, which takes one argument:

- `$template`, the name of the template to set.

As an example:

```
$this->layout()->setTemplate('layout/newlayout');
```

It also implements the `__invoke` magic method, which allows for even easier setting of the template:

```
$this->layout('layout/newlayout');
```

185.6 Params Plugin

The `Params` plugin allows for accessing parameters in actions from different sources.

It exposes several methods, one for each parameter source:

fromFiles(*string $name = null*, *mixed $default = null*)
For retrieving all or one single **file**. If `$name` is *null*, all files will be returned.

Return type `array|ArrayAccess|null`

fromHeader(*string $header = null*, *mixed $default = null*)
For retrieving all or one single **header** parameter. If `$header` is *null*, all header parameters will be returned.

Return type `null|Zend\Http\Header\HeaderInterface`

fromPost(*string $param = null*, *mixed $default = null*)
For retrieving all or one single **post** parameter. If `$param` is *null*, all post parameters will be returned.

Return type `mixed`

fromQuery(*string $param = null*, *mixed $default = null*)
For retrieving all or one single **query** parameter. If `$param` is *null*, all query parameters will be returned.

Return type `mixed`

fromRoute (*string $param = null*, *mixed $default = null*)
For retrieving all or one single **route** parameter. If `$param` is *null*, all route parameters will be returned.

Return type `mixed`

It also implements the `__invoke` magic method, which allows for short circuiting to the `fromRoute` method:

```
$this->params()->fromRoute('param', $default);
// or
$this->params('param', $default);
```

185.7 Post/Redirect/Get Plugin

When a user sends a POST request (e.g. after submitting a form), their browser will try to protect them from sending the POST again, breaking the back button, causing browser warnings and pop-ups, and sometimes reposting the form. Instead, when receiving a POST, we should store the data in a session container and redirect the user to a GET request.

This plugin can be invoked with two arguments:

- `$redirect`, a string containing the redirect location which can either be a named route or a URL, based on the contents of the second parameter.
- `$redirectToUrl`, a boolean that when set to TRUE, causes the first parameter to be treated as a URL instead of a route name (this is required when redirecting to a URL instead of a route). This argument defaults to false.

When no arguments are provided, the current matched route is used.

Example Usage

```
// Pass in the route/url you want to redirect to after the POST
$prg = $this->prg('/user/register', true);

if ($prg instanceof \Zend\Http\PhpEnvironment\Response) {
    // returned a response to redirect us
    return $prg;
} elseif ($prg === false) {
    // this wasn't a POST request, but there were no params in the flash messenger
    // probably this is the first time the form was loaded
    return array('form' => $myForm);
}

// $prg is an array containing the POST params from the previous request
$form->setData($prg);

// ... your form processing code here
```

185.8 File Post/Redirect/Get Plugin

While similar to the standard *Post/Redirect/Get Plugin*, the File PRG Plugin will work for forms with file inputs. The difference is in the behavior: The File PRG Plugin will interact directly with your form instance and the file inputs, rather than *only* returning the POST params from the previous request.

By interacting directly with the form, the File PRG Plugin will turn off any file inputs' `required` flags for already uploaded files (for a partially valid form state), as well as run the file input filters to move the uploaded files into a new location (configured by the user).

> **Warning:** You **must** attach a Filter for moving the uploaded files to a new location, such as the *RenameUpload Filter*, or else your files will be removed upon the redirect.

This plugin can be invoked with three arguments:

- `$form`: the form instance.
- `$redirect`: (Optional) a string containing the redirect location which can either be a named route or a URL, based on the contents of the third parameter. If this argument is not provided, it will default to the current matched route.
- `$redirectToUrl`: (Optional) a boolean that when set to TRUE, causes the second parameter to be treated as a URL instead of a route name (this is required when redirecting to a URL instead of a route). This argument defaults to false.

Example Usage

```
$myForm = new Zend\Form\Form('my-form');
$myForm->add(array(
    'type' => 'Zend\Form\Element\File',
    'name' => 'file',
));
// NOTE: Without a filter to move the file,
//       our files will disappear between the requests
$myForm->getInputFilter()->getFilterChain()->attach(
    new Zend\Filter\File\RenameUpload(array(
        'target'    => './data/tmpuploads/file',
        'randomize' => true,
    ))
);

// Pass in the route/url you want to redirect to after the POST
$prg = $this->prg($myForm, '/user/profile-pic', true);

if ($prg instanceof \Zend\Http\PhpEnvironment\Response) {
    // Returned a response to redirect us
    return $prg;
} elseif ($prg === false) {
    // First time the form was loaded
    return array('form' => $myForm);
}

// Form was submitted.
// $prg is now an array containing the POST params from the previous request,
// but we don't have to apply it to the form since that has already been done.

// Process the form
if ($form->isValid()) {
    // ...Save the form...
    return $this->redirect()->toRoute('/user/profile-pic/success');
} else {
    // Form not valid, but file uploads might be valid and uploaded
    $fileErrors = $form->get('file')->getMessages();
    if (empty($fileErrors)) {
        $tempFile = $form->get('file')->getValue();
    }
}
```

185.9 Redirect Plugin

Redirections are quite common operations within applications. If done manually, you will need to do the following steps:

- Assemble a url using the router
- Create and inject a "Location" header into the `Response` object, pointing to the assembled URL
- Set the status code of the `Response` object to one of the 3xx HTTP statuses.

The `Redirect` plugin does this work for you. It offers three methods:

toRoute(*string $route = null*, *array $params = array()*, *array $options = array()*, *boolean $reuseMatchedParams = false*)
Redirects to a named route, using the provided `$params` and `$options` to assembled the URL.

Return type `Zend\Http\Response`

toUrl(*string $url*)
Simply redirects to the given URL.

Return type `Zend\Http\Response`

refresh()
Refresh to current route

Return type `Zend\Http\Response`

In each case, the `Response` object is returned. If you return this immediately, you can effectively short-circuit execution of the request.

Note: This plugin requires that the controller invoking it implements `InjectApplicationEventInterface`, and thus has an `MvcEvent` composed, as it retrieves the router from the event object.

As an example:

```
return $this->redirect()->toRoute('login-success');
```

185.10 Url Plugin

Often you may want to generate URLs from route definitions within your controllers – in order to seed the view, generate headers, etc. While the `MvcEvent` object composes the router, doing so manually would require this workflow:

```
$router = $this->getEvent()->getRouter();
$url    = $router->assemble($params, array('name' => 'route-name'));
```

The `Url` helper makes this slightly more convenient:

```
$url = $this->url()->fromRoute('route-name', $params);
```

The `fromRoute()` method is the only public method defined, and has the following signature:

```
public function fromRoute($route, array $params = array(), array $options = array())
```

Note: This plugin requires that the controller invoking it implements `InjectApplicationEventInterface`, and thus has an `MvcEvent` composed, as it retrieves the router from the event object.

CHAPTER

SIX

EXAMPLES

186.1 Controllers

186.1.1 Accessing the Request and Response

When using `AbstractActionController` or `AbstractRestfulController`, the request and response object are composed directly into the controller as soon as `dispatch()` is called. You may access them in the following ways:

```
// Using explicit accessor methods
$request  = $this->getRequest();
$response = $this->getResponse();

// Using direct property access
$request  = $this->request;
$response = $this->response;
```

Additionally, if your controller implements `InjectApplicationEventInterface` (as both `AbstractActionController` and `AbstractRestfulController` do), you can access these objects from the attached `MvcEvent`:

```
$event    = $this->getEvent();
$request  = $event->getRequest();
$response = $event->getResponse();
```

The above can be useful when composing event listeners into your controller.

186.1.2 Accessing routing parameters

The parameters returned when routing completes are wrapped in a `Zend\Mvc\Router\RouteMatch` object. This object is detailed in the section on *Routing*.

Within your controller, if you implement `InjectApplicationEventInterface` (as both `AbstractActionController` and `AbstractRestfulController` do), you can access this object from the attached `MvcEvent`:

```
$event   = $this->getEvent();
$matches = $event->getRouteMatch();
```

Once you have the `RouteMatch` object, you can pull parameters from it.

The same can be done using the *Params plugin*.

186.1.3 Returning early

You can effectively short-circuit execution of the application at any point by returning a `Response` from your controller or any event. When such a value is discovered, it halts further execution of the event manager, bubbling up to the `Application` instance, where it is immediately returned.

As an example, the `Redirect` plugin returns a `Response`, which can be returned immediately so as to complete the request as quickly as possible. Other use cases might be for returning JSON or XML results from web service endpoints, returning "401 Forbidden" results, etc.

186.2 Bootstrapping

186.2.1 Registering module-specific listeners

Often you may want module-specific listeners. As an example, this would be a simple and effective way to introduce authorization, logging, or caching into your application.

Each `Module` class can have an optional `onBootstrap()` method. Typically, you'll do module-specific configuration here, or setup event listeners for you module here. The `onBootstrap()` method is called for **every** module on **every** page request and should **only** be used for performing **lightweight** tasks such as registering event listeners.

The base `Application` class shipped with the framework has an `EventManager` associated with it, and once the modules are initialized, it triggers the *bootstrap* event, with a `getApplication()` method on the event.

So, one way to accomplish module-specific listeners is to listen to that event, and register listeners at that time. As an example:

```
namespace SomeCustomModule;

class Module
{
    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function onBootstrap($e)
    {
        $application = $e->getApplication();
        $config      = $application->getConfig();
        $view        = $application->getServiceManager()->get('ViewHelperManager');
        // You must have these keys in you application config
        $view->headTitle($config['view']['base_title']);

        // This is your custom listener
        $listener   = new Listeners\ViewListener();
        $listener->setView($view);
        $application->getEventManager()->attachAggregate($listener);
    }
}
```

The above demonstrates several things. First, it demonstrates a listener on the application's *bootstrap* event (the `onBootstrap()` method). Second, it demonstrates that listener, and how it can be used to register listeners with the application. It grabs the `Application` instance; from the `Application`, it is able to grab the attached service manager and configuration. These are then used to retrieve the view, configure some helpers, and then register a listener aggregate with the application event manager.

CHAPTER

SEVEN

INTRODUCTION TO ZEND\NAVIGATION

`Zend\Navigation` is a component for managing trees of pointers to web pages. Simply put: It can be used for creating menus, breadcrumbs, links, and sitemaps, or serve as a model for other navigation related purposes.

187.1 Pages and Containers

There are two main concepts in `Zend\Navigation`:

187.1.1 Pages

A page (`Zend\Navigation\AbstractPage`) in `Zend\Navigation` – in its most basic form – is an object that holds a pointer to a web page. In addition to the pointer itself, the page object contains a number of other properties that are typically relevant for navigation, such as `label`, `title`, etc.

Read more about pages in the *pages* section.

187.1.2 Containers

A navigation container (`Zend\Navigation\AbstractContainer`) is a container class for pages. It has methods for adding, retrieving, deleting and iterating pages. It implements the SPL interfaces `RecursiveIterator` and `Countable`, and can thus be iterated with SPL iterators such as `RecursiveIteratorIterator`.

Read more about containers in the *containers* section.

Note: `Zend\Navigation\AbstractPage` extends `Zend\Navigation\AbstractContainer`, which means that a page can have sub pages.

187.2 View Helpers

187.2.1 Separation of data (model) and rendering (view)

Classes in the `Zend\Navigation` namespace do not deal with rendering of navigational elements. Rendering is done with navigational view helpers. However, pages contain information that is used by view helpers when rendering, such as; `label`, `class` (*CSS*), `title`, `lastmod` and `priority` properties for sitemaps, etc.

Read more about rendering navigational elements in the *view helpers* section.

CHAPTER

EIGHT

QUICK START

The fastest way to get up and running with `Zend\Navigation` is by the **navigation** key in your service manager configuration and the navigation factory will handle the rest for you. After setting up the configuration simply use the key name with the `Zend\Navigation` view helper to output the container.

```php
<?php
// your configuration file, e.g., config/autoload/global.php
return array(
    // ...

    'navigation' => array(
        'default' => array(
            array(
                'label' => 'Home',
                'route' => 'home',
            ),
            array(
                'label' => 'Page #1',
                'route' => 'page-1',
                'pages' => array(
                    array(
                        'label' => 'Child #1',
                        'route' => 'page-1-child',
                    ),
                ),
            ),
            array(
                'label' => 'Page #2',
                'route' => 'page-2',
            ),
        ),
    ),
    'service_manager' => array(
        'factories' => array(
            'navigation' => 'Zend\Navigation\Service\DefaultNavigationFactory',
        ),
    ),
    // ...
);
```

```php
<!-- in your layout -->
<!-- ... -->

<body>
```

```
5       <?php echo $this->navigation('navigation')->menu(); ?>
6   </body>
7   <!-- ... -->
```

CHAPTER

NINE

PAGES

`Zend\Navigation` ships with two page types:

- *MVC pages* – using the class `Zend\Navigation\Page\Mvc`
- *URI pages* – using the class `Zend\Navigation\Page\Uri`

MVC pages are link to on-site web pages, and are defined using MVC parameters (*action*, *controller*, *route*, *params*). URI pages are defined by a single property *uri*, which give you the full flexibility to link off-site pages or do other things with the generated links (e.g. an URI that turns into `<a href="#">foo<a>`).

CHAPTER

COMMON PAGE FEATURES

All page classes must extend `Zend\Navigation\Page\AbstractPage`, and will thus share a common set of features and properties. Most notably they share the options in the table below and the same initialization process.

Option keys are mapped to *set* methods. This means that the option *order* maps to the method `setOrder()`, and *reset_params* maps to the method `setResetParams()`. If there is no setter method for the option, it will be set as a custom property of the page.

Read more on extending `Zend\Navigation\Page\AbstractPage` in *Creating custom page types*.

Key	Type	Default	Description
label	String	NULL	A page label, such as 'Home' or 'Blog'.
fragment	String | NULL	NULL	A fragment identifier (anchor identifier) pointing to an anchor within a resource that is subordinate to another, primary resource. The fragment identifier introduced by a hash mark "#". Example: `http://www.example.org/foo.html#bar` (*bar* is the fragment identifier)
id	String | Integer	NULL	An *id* tag/attribute that may be used when rendering the page, typically in an anchor element.
class	String	NULL	A *CSS* class that may be used when rendering the page, typically in an anchor element.
title	String	NULL	A short page description, typically for using as the title attribute in an anchor.
target	String	NULL	Specifies a target that may be used for the page, typically in an anchor element.
rel	Array	array()	Specifies forward relations for the page. Each element in the array is a key-value pair, where the key designates the relation/link type, and the value is a pointer to the linked page. An example of a key-value pair is `'alternate' => 'format/plain.html'`. To allow full flexibility, there are no restrictions on relation values. The value does not have to be a string. Read more about `rel` and `rev` in the section on the Links helper.
rev	Array	array()	Specifies reverse relations for the page. Works exactly like rel.
order	String | Integer | NULL	NULL	Works like order for elements in `Zend\Form`. If specified, the page will be iterated in a specific order, meaning you can force a page to be iterated before others by setting the order attribute to a low number, e.g. -100. If a String is given, it must parse to a valid int. If `NULL` is given, it will be reset, meaning the order in which the page was added to the container will be used.
resource	String | `Zend\Permissions\Acl\Resource\ResourceInterface` | NULL	NULL	ACL resource to associate with the page. Read more in the section on ACL integration in view helpers.
privilege	String | NULL	NULL	ACL privilege to associate with the page. Read more in the section on ACL integration in view helpers.
active	Boolean	FALSE	Whether the page should be considered active for the current request. If active is FALSE or not given, MVC pages will check its properties against the request object upon calling `$page->isActive()`.
visible	Boolean	TRUE	Whether page should be visible for the user, or just be a part of the structure. Invi
pages	Array	`Zend\Config` | NULL	NULL | Child pages of the page. This could be an Array or `Zer`

Note: Custom properties

All pages support setting and getting of custom properties by use of the magic methods `__set($name, $value)`, `__get($name)`, `__isset($name)` and `__unset($name)`. Custom properties may have any value, and will be included in the array that is returned from `$page->toArray()`, which means that pages can be serialized/deserialized successfully even if the pages contains properties that are not native in the page class.

Both native and custom properties can be set using `$page->set($name, $value)` and retrieved using

`$page->get($name)`, or by using magic methods.

Custom page properties

This example shows how custom properties can be used.

```
$page = new Zend\Navigation\Page\Mvc();
$page->foo     = 'bar';
$page->meaning = 42;

echo $page->foo;

if ($page->meaning != 42) {
    // action should be taken
}
```

CHAPTER

ONE

ZEND\NAVIGATION\PAGE\MVC

MVC pages are defined using *MVC* parameters known from the `Zend\Mvc` component. An *MVC* page will use `Zend\Mvc\Router\RouteStackInterface` internally in the `getHref()` method to generate hrefs, and the `isActive()` method will compare the `Zend\Mvc\Router\RouteMatch` params with the page's params to determine if the page is active.

Table 191.1: MVC page options

Key	Type	Default	Description
action	String	NULL	Action name to use when generating href to the page.
controller	String	NULL	Controller name to use when generating href to the page.
params	Array	array()	User params to use when generating href to the page.
route	String	NULL	Route name to use when generating href to the page.
routeMatch	`Zend\Mvc\Router\RouteMatch`	NULL	RouteInterface matches used for routing parameters and testing validity.
router	`Zend\Mvc\Router\RouteStackInterface`	NULL	Router for assembling URLs

Note: The *URI* returned is relative to the *baseUrl* in `Zend\Mvc\Router\Http\TreeRouteStack`. In the examples, the baseUrl is '/' for simplicity.

getHref() generates the page URI

This example show that *MVC* pages use `Zend\Mvc\Router\RouteStackInterface` internally to generate *URI*s when calling *$page->getHref()*.

```
// Create route
$route = Zend\Mvc\Router\Http\Segment::factory(array(
   'route'       => '/[:controller[/:action][/:id]]',
   'constraints' => array(
      'controller' => '[a-zA-Z][a-zA-Z0-9_-]+',
      'action'     => '[a-zA-Z][a-zA-Z0-9_-]+',
      'id'         => '[0-9]+',
   ),
   array(
      'controller' => 'Album\Controller\Album',
      'action'     => 'index',
```

```
    )
));
$router = new Zend\Mvc\Router\Http\TreeRouteStack();
$router->addRoute('default', $route);

// getHref() returns /album/add
$page = new Zend\Navigation\Page\Mvc(array(
    'action'     => 'add',
    'controller' => 'album',
));

// getHref() returns /album/edit/1337
$page = new Zend\Navigation\Page\Mvc(array(
    'action'     => 'edit',
    'controller' => 'album',
    'params'     => array('id' => 1337),
));
```

isActive() determines if page is active

This example show that *MVC* pages determine whether they are active by using the params found in the route match object.

```
/**
 * Dispatched request:
 * - controller: album
 * - action:     index
 */
$page1 = new Zend\Navigation\Page\Mvc(array(
    'action'     => 'index',
    'controller' => 'album',
));

$page2 = new Zend\Navigation\Page\Mvc(array(
    'action'     => 'edit',
    'controller' => 'album',
));

$page1->isActive(); // returns true
$page2->isActive(); // returns false

/**
 * Dispatched request:
 * - controller: album
 * - action:     edit
 * - id:         1337
 */
$page = new Zend\Navigation\Page\Mvc(array(
    'action'     => 'edit',
    'controller' => 'album',
    'params'     => array('id' => 1337),
));

// returns true, because request has the same controller and action
$page->isActive();

/**
```

```
 * Dispatched request:
 * - controller: album
 * - action:     edit
 */
$page = new Zend\Navigation\Page\Mvc(array(
    'action'     => 'edit',
    'controller' => 'album',
    'params'     => array('id' => null),
));

// returns false, because page requires the id param to be set in the request
$page->isActive(); // returns false
```

Using routes

Routes can be used with *MVC* pages. If a page has a route, this route will be used in `getHref()` to generate the *URL* for the page.

Note: Note that when using the *route* property in a page, you do not need to specify the default params that the route defines (controller, action, etc.).

```
// the following route is added to the ZF router
$route = Zend\Mvc\Router\Http\Segment::factory(array(
   'route'       => '/a/:id',
   'constraints' => array(
      'id' => '[0-9]+',
   ),
   array(
      'controller' => 'Album\Controller\Album',
      'action'     => 'show',
   )
));
$router = new Zend\Mvc\Router\Http\TreeRouteStack();
$router->addRoute('albumShow', $route);

// a page is created with a 'route' option
$page = new Zend\Navigation\Page\Mvc(array(
    'label'      => 'Show album',
    'route'      => 'albumShow',
    'params'     => array('id' => 42)
));

// returns: /a/42
$page->getHref();
```

CHAPTER

TWO

ZEND\NAVIGATION\PAGE\URI

Pages of type `Zend\Navigation\Page\Uri` can be used to link to pages on other domains or sites, or to implement custom logic for the page. *URI* pages are simple; in addition to the common page options, a *URI* page takes only one option — *uri*. The *uri* will be returned when calling `$page->getHref()`, and may be a `String` or `NULL`.

Note: `Zend\Navigation\Page\Uri` will not try to determine whether it should be active when calling `$page->isActive()`. It merely returns what currently is set, so to make a *URI* page active you have to manually call `$page->setActive()` or specifying *active* as a page option when constructing.

Table 192.1: URI page options

Key	Type	Default	Description
uri	String	NULL	URI to page. This can be any string or NULL.

CHAPTER

THREE

CREATING CUSTOM PAGE TYPES

When extending `Zend\Navigation\Page\AbstractPage`, there is usually no need to override the constructor or the methods `setOptions()` or `setConfig()`. The page constructor takes a single parameter, an `Array` or a `Zend\Config` object, which is passed to `setOptions()` or `setConfig()` respectively. Those methods will in turn call `set()` method, which will map options to native or custom properties. If the option `internal_id` is given, the method will first look for a method named `setInternalId()`, and pass the option to this method if it exists. If the method does not exist, the option will be set as a custom property of the page, and be accessible via `$internalId = $page->internal_id;` or `$internalId = $page->get('internal_id');`.

The most simple custom page

The only thing a custom page class needs to implement is the `getHref()` method.

```
class My\Simple\Page extends Zend\Navigation\Page\AbstractPage
{
    public function getHref()
    {
        return 'something-completely-different';
    }
}
```

A custom page with properties

When adding properties to an extended page, there is no need to override/modify `setOptions()` or `setConfig()`.

```
class My\Navigation\Page extends Zend\Navigation\Page\AbstractPage
{
    protected $foo;
    protected $fooBar;

    public function setFoo($foo)
    {
        $this->foo = $foo;
    }

    public function getFoo()
    {
        return $this->foo;
    }

```

```
    public function setFooBar($fooBar)
    {
        $this->fooBar = $fooBar;
    }

    public function getFooBar()
    {
        return $this->fooBar;
    }

    public function getHref()
    {
        return $this->foo . '/' . $this->fooBar;
    }
}

// can now construct using
$page = new My\Navigation\Page(array(
    'label'   => 'Property names are mapped to setters',
    'foo'     => 'bar',
    'foo_bar' => 'baz'
));

// ...or
$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'type'    => 'My\Navigation\Page',
    'label'   => 'Property names are mapped to setters',
    'foo'     => 'bar',
    'foo_bar' => 'baz'
));
```

CHAPTER

FOUR

CREATING PAGES USING THE PAGE FACTORY

All pages (also custom classes), can be created using the page factory, `Zend\Navigation\Page\AbstractPage::factory()`. The factory can take an array with options, or a `Zend\Config` object. Each key in the array/config corresponds to a page option, as seen in the section on *Pages*. If the option *uri* is given and no *MVC* options are given (*action, controller, route*), an *URI* page will be created. If any of the *MVC* options are given, an *MVC* page will be created.

If *type* is given, the factory will assume the value to be the name of the class that should be created. If the value is *mvc* or *uri* and *MVC*/URI page will be created.

Creating an MVC page using the page factory

```
$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'label'  => 'My MVC page',
    'action' => 'index',
));

$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'label'      => 'Search blog',
    'action'     => 'index',
    'controller' => 'search',
));

$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'label' => 'Home',
    'route' => 'home',
));

$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'type'   => 'mvc',
    'label'  => 'My MVC page',
));
```

Creating a URI page using the page factory

```
$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'label' => 'My URI page',
    'uri'   => 'http://www.example.com/',
```

```
));

$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'label'  => 'Search',
    'uri'    => 'http://www.example.com/search',
    'active' => true,
));

$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'label' => 'My URI page',
    'uri'   => '#',
));

$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'type'  => 'uri',
    'label' => 'My URI page',
));
```

Creating a custom page type using the page factory

To create a custom page type using the factory, use the option *type* to specify a class name to instantiate.

```
class My\Navigation\Page extends Zend\Navigation\Page\AbstractPage
{
    protected $_fooBar = 'ok';

    public function setFooBar($fooBar)
    {
        $this->_fooBar = $fooBar;
    }
}

$page = Zend\Navigation\Page\AbstractPage::factory(array(
    'type'    => 'My\Navigation\Page',
    'label'   => 'My custom page',
    'foo_bar' => 'foo bar',
));
```

CHAPTER

FIVE

CONTAINERS

Containers have methods for adding, retrieving, deleting and iterating pages. Containers implement the SPL interfaces `RecursiveIterator` and `Countable`, meaning that a container can be iterated using the SPL `RecursiveIteratorIterator` class.

195.1 Creating containers

`Zend\Navigation\AbstractContainer` can not be instantiated directly. Use `Zend\Navigation\Navigation` if you want to instantiate a container.

`Zend\Navigation\Navigation` can be constructed entirely empty, or take an array or a `Zend\Config\Config` object with pages to put in the container. Each page in the given array/config will eventually be passed to the `addPage()` method of the container class, which means that each element in the array/config can be an array, a config object or a `Zend\Navigation\Page\AbstractPage` instance.

Creating a container using an array

```
/*
 * Create a container from an array
 *
 * Each element in the array will be passed to
 * Zend\Navigation\Page\AbstractPage::factory() when constructing.
 */
$container = new Zend\Navigation\Navigation(array(
    array(
        'label' => 'Page 1',
        'id' => 'home-link',
        'uri' => '/',
    ),
    array(
        'label' => 'Zend',
        'uri' => 'http://www.zend-project.com/',
        'order' => 100,
    ),
    array(
        'label' => 'Page 2',
        'controller' => 'page2',
        'pages' => array(
            array(
                'label' => 'Page 2.1',
                'action' => 'page2_1',
```

```
                'controller' => 'page2',
                'class' => 'special-one',
                'title' => 'This element has a special class',
                'active' => true,
            ),
            array(
                'label' => 'Page 2.2',
                'action' => 'page2_2',
                'controller' => 'page2',
                'class' => 'special-two',
                'title' => 'This element has a special class too',
            ),
        ),
    ),
    array(
        'label' => 'Page 2 with params',
        'action' => 'index',
        'controller' => 'page2',
        // specify a param or two,
        'params' => array(
            'format' => 'json',
            'foo' => 'bar',
        )
    ),
    array(
        'label' => 'Page 2 with params and a route',
        'action' => 'index',
        'controller' => 'page2',
        // specify a route name and a param for the route
        'route' => 'nav-route-example',
        'params' => array(
            'format' => 'json',
        ),
    ),
    array(
        'label' => 'Page 3',
        'action' => 'index',
        'controller' => 'index',
        'module' => 'mymodule',
        'reset_params' => false,
    ),
    array(
        'label' => 'Page 4',
        'uri' => '#',
        'pages' => array(
            array(
                'label' => 'Page 4.1',
                'uri' => '/page4',
                'title' => 'Page 4 using uri',
                'pages' => array(
                    array(
                        'label' => 'Page 4.1.1',
                        'title' => 'Page 4 using mvc params',
                        'action' => 'index',
                        'controller' => 'page4',
                        // let's say this page is active
                        'active' => '1',
                    )
```

```
                ),
            ),
        ),
    ),
    array(
        'label' => 'Page 0?',
        'uri' => '/setting/the/order/option',
        // setting order to -1 should make it appear first
        'order' => -1,
    ),
    array(
        'label' => 'Page 5',
        'uri' => '/',
        // this page should not be visible
        'visible' => false,
        'pages' => array(
            array(
                'label' => 'Page 5.1',
                'uri' => '#',
                'pages' => array(
                    array(
                        'label' => 'Page 5.1.1',
                        'uri' => '#',
                        'pages' => array(
                            array(
                                'label' => 'Page 5.1.2',
                                'uri' => '#',
                                // let's say this page is active
                                'active' => true,
                            ),
                        ),
                    ),
                ),
            ),
        ),
    ),
    array(
        'label' => 'ACL page 1 (guest)',
        'uri' => '#acl-guest',
        'resource' => 'nav-guest',
        'pages' => array(
            array(
                'label' => 'ACL page 1.1 (foo)',
                'uri' => '#acl-foo',
                'resource' => 'nav-foo',
            ),
            array(
                'label' => 'ACL page 1.2 (bar)',
                'uri' => '#acl-bar',
                'resource' => 'nav-bar',
            ),
            array(
                'label' => 'ACL page 1.3 (baz)',
                'uri' => '#acl-baz',
                'resource' => 'nav-baz',
            ),
            array(
                'label' => 'ACL page 1.4 (bat)',
```

```
                'uri' => '#acl-bat',
                'resource' => 'nav-bat',
            ),
        ),
    ),
    array(
        'label' => 'ACL page 2 (member)',
        'uri' => '#acl-member',
        'resource' => 'nav-member',
    ),
    array(
        'label' => 'ACL page 3 (admin',
        'uri' => '#acl-admin',
        'resource' => 'nav-admin',
        'pages' => array(
            array(
                'label' => 'ACL page 3.1 (nothing)',
                'uri' => '#acl-nada',
            ),
        ),
    ),
    array(
        'label' => 'Zend Framework',
        'route' => 'zf-route',
    ),
));
```

Creating a container using a config object

```
/* CONTENTS OF /path/to/navigation.xml:
<?xml version="1.0" encoding="UTF-8"?>
<nav>

  <zend>
      <label>Zend</label>
      <uri>http://www.zend-project.com/</uri>
      <order>100</order>
  </zend>

  <page1>
      <label>Page 1</label>
      <uri>page1</uri>
      <pages>

          <page1_1>
              <label>Page 1.1</label>
              <uri>page1/page1_1</uri>
          </page1_1>

      </pages>
  </page1>

  <page2>
      <label>Page 2</label>
      <uri>page2</uri>
      <pages>
```

```
28
29          <page2_1>
30              <label>Page 2.1</label>
31              <uri>page2/page2_1</uri>
32          </page2_1>
33
34          <page2_2>
35              <label>Page 2.2</label>
36              <uri>page2/page2_2</uri>
37              <pages>
38
39                  <page2_2_1>
40                      <label>Page 2.2.1</label>
41                      <uri>page2/page2_2/page2_2_1</uri>
42                  </page2_2_1>
43
44                  <page2_2_2>
45                      <label>Page 2.2.2</label>
46                      <uri>page2/page2_2/page2_2_2</uri>
47                      <active>1</active>
48                  </page2_2_2>
49
50              </pages>
51          </page2_2>
52
53          <page2_3>
54              <label>Page 2.3</label>
55              <uri>page2/page2_3</uri>
56              <pages>
57
58                  <page2_3_1>
59                      <label>Page 2.3.1</label>
60                      <uri>page2/page2_3/page2_3_1</uri>
61                  </page2_3_1>
62
63                  <page2_3_2>
64                      <label>Page 2.3.2</label>
65                      <uri>page2/page2_3/page2_3_2</uri>
66                      <visible>0</visible>
67                      <pages>
68
69                              <page2_3_2_1>
70                                  <label>Page 2.3.2.1</label>
71                                  <uri>page2/page2_3/page2_3_2/1</uri>
72                                  <active>1</active>
73                              </page2_3_2_1>
74
75                              <page2_3_2_2>
76                                  <label>Page 2.3.2.2</label>
77                                  <uri>page2/page2_3/page2_3_2/2</uri>
78                                  <active>1</active>
79
80                                  <pages>
81                                      <page_2_3_2_2_1>
82                                          <label>Ignore</label>
83                                          <uri>#</uri>
84                                          <active>1</active>
85                                      </page_2_3_2_2_1>
```

```
                                    </pages>
                                </page2_3_2_2>

                        </pages>
                    </page2_3_2>

                    <page2_3_3>
                        <label>Page 2.3.3</label>
                        <uri>page2/page2_3/page2_3_3</uri>
                        <resource>admin</resource>
                        <pages>

                                <page2_3_3_1>
                                    <label>Page 2.3.3.1</label>
                                    <uri>page2/page2_3/page2_3_3/1</uri>
                                    <active>1</active>
                                </page2_3_3_1>

                                <page2_3_3_2>
                                    <label>Page 2.3.3.2</label>
                                    <uri>page2/page2_3/page2_3_3/2</uri>
                                    <resource>guest</resource>
                                    <active>1</active>
                                </page2_3_3_2>

                        </pages>
                    </page2_3_3>

                </pages>
            </page2_3>

        </pages>
    </page2>

    <page3>
        <label>Page 3</label>
        <uri>page3</uri>
        <pages>

            <page3_1>
                <label>Page 3.1</label>
                <uri>page3/page3_1</uri>
                <resource>guest</resource>
            </page3_1>

            <page3_2>
                <label>Page 3.2</label>
                <uri>page3/page3_2</uri>
                <resource>member</resource>
                <pages>

                    <page3_2_1>
                        <label>Page 3.2.1</label>
                        <uri>page3/page3_2/page3_2_1</uri>
                    </page3_2_1>

                    <page3_2_2>
                        <label>Page 3.2.2</label>
```

```
                        <uri>page3/page3_2/page3_2_2</uri>
                        <resource>admin</resource>
                    </page3_2_2>

                </pages>
            </page3_2>

            <page3_3>
                <label>Page 3.3</label>
                <uri>page3/page3_3</uri>
                <resource>special</resource>
                <pages>

                    <page3_3_1>
                        <label>Page 3.3.1</label>
                        <uri>page3/page3_3/page3_3_1</uri>
                        <visible>0</visible>
                    </page3_3_1>

                    <page3_3_2>
                        <label>Page 3.3.2</label>
                        <uri>page3/page3_3/page3_3_2</uri>
                        <resource>admin</resource>
                    </page3_3_2>

                </pages>
            </page3_3>

        </pages>
    </page3>

    <home>
        <label>Home</label>
        <order>-100</order>
        <module>default</module>
        <controller>index</controller>
        <action>index</action>
    </home>

</nav>
 */

$reader = new Zend\Config\Reader\Xml();
$config = $reader->fromFile('/path/to/navigation.xml');
$container = new Zend\Navigation\Navigation($config);
```

195.2 Adding pages

Adding pages to a container can be done with the methods `addPage()`, `addPages()`, or `setPages()`. See examples below for explanation.

Adding pages to a container

```
// create container
$container = new Zend\Navigation\Navigation();

// add page by giving a page instance
$container->addPage(
    Zend\Navigation\Page\AbstractPage::factory(
        array(
            'uri' => 'http://www.example.com/',
        )
    )
);

// add page by giving an array
$container->addPage(
    array(
        'uri' => 'http://www.example.com/',
    )
);

// add page by giving a config object
$container->addPage(
    new Zend\Config\Config(
        array(
            'uri' => 'http://www.example.com/',
        )
    )
);

$pages = array(
    array(
        'label'  => 'Save',
        'action' => 'save',
    ),
    array(
        'label' =>  'Delete',
        'action' => 'delete',
    )
);

// add two pages
$container->addPages($pages);

// remove existing pages and add the given pages
$container->setPages($pages);
```

195.3 Removing pages

Removing pages can be done with `removePage()` or `removePages()`. The first method accepts a an instance of a page, or an integer. The integer corresponds to the `order` a page has. The latter method will remove all pages in the container.

Removing pages from a container

```
$container = new Zend\Navigation\Navigation(array(
    array(
        'label'  => 'Page 1',
        'action' => 'page1',
    ),
    array(
        'label'  => 'Page 2',
        'action' => 'page2',
        'order'  => 200,
    ),
    array(
        'label'  => 'Page 3',
        'action' => 'page3',
    )
));

// remove page by implicit page order
$container->removePage(0);      // removes Page 1

// remove page by instance
$page3 = $container->findOneByAction('page3');
$container->removePage($page3); // removes Page 3

// remove page by explicit page order
$container->removePage(200);    // removes Page 2

// remove all pages
$container->removePages();      // removes all pages
```

195.4 Finding pages

Containers have finder methods for retrieving pages. They are `findOneBy($property, $value)`, `findAllBy($property, $value)`, and `findBy($property, $value, $all = false)`. Those methods will recursively search the container for pages matching the given `$page->$property == $value`. The first method, `findOneBy()`, will return a single page matching the property with the given value, or `NULL` if it cannot be found. The second method will return all pages with a property matching the given value. The third method will call one of the two former methods depending on the `$all` flag.

The finder methods can also be used magically by appending the property name to `findBy`, `findOneBy`, or `findAllBy`, e.g. `findOneByLabel('Home')` to return the first matching page with label 'Home'. Other combinations are `findByLabel(...)`, `findOneByTitle(...)`, `findAllByController(...)`, etc. Finder methods also work on custom properties, such as `findByFoo('bar')`.

Finding pages in a container

```
$container = new Zend\Navigation\Navigation(array(
    array(
        'label' => 'Page 1',
        'uri'   => 'page-1',
        'foo'   => 'bar',
        'pages' => array(
            array(
```

```
                'label' => 'Page 1.1',
                'uri'   => 'page-1.1',
                'foo'   => 'bar',
            ),
            array(
                'label' => 'Page 1.2',
                'uri'   => 'page-1.2',
                'class' => 'my-class',
            ),
            array(
                'type'   => 'uri',
                'label'  => 'Page 1.3',
                'uri'    => 'page-1.3',
                'action' => 'about',
            )
        )
    ),
    array(
        'label'      => 'Page 2',
        'id'         => 'page_2_and_3',
        'class'      => 'my-class',
        'module'     => 'page2',
        'controller' => 'index',
        'action'     => 'page1',
    ),
    array(
        'label'      => 'Page 3',
        'id'         => 'page_2_and_3',
        'module'     => 'page3',
        'controller' => 'index',
    ),
));

// The 'id' is not required to be unique, but be aware that
// having two pages with the same id will render the same id attribute
// in menus and breadcrumbs.
$found = $container->findBy('id',
                            'page_2_and_3');      // returns Page 2
$found = $container->findOneBy('id',
                               'page_2_and_3');   // returns Page 2
$found = $container->findBy('id',
                            'page_2_and_3',
                            true);                // returns Page 2 and Page 3
$found = $container->findById('page_2_and_3');    // returns Page 2
$found = $container->findOneById('page_2_and_3'); // returns Page 2
$found = $container->findAllById('page_2_and_3'); // returns Page 2 and Page 3

// Find all matching CSS class my-class
$found = $container->findAllBy('class',
                               'my-class');       // returns Page 1.2 and Page 2
$found = $container->findAllByClass('my-class');  // returns Page 1.2 and Page 2

// Find first matching CSS class my-class
$found = $container->findOneByClass('my-class');  // returns Page 1.2

// Find all matching CSS class non-existant
$found = $container->findAllByClass('non-existant'); // returns array()

```

```
// Find first matching CSS class non-existant
$found = $container->findOneByClass('non-existant'); // returns null

// Find all pages with custom property 'foo' = 'bar'
$found = $container->findAllBy('foo', 'bar'); // returns Page 1 and Page 1.1

// To achieve the same magically, 'foo' must be in lowercase.
// This is because 'foo' is a custom property, and thus the
// property name is not normalized to 'Foo'
$found = $container->findAllByfoo('bar');

// Find all with controller = 'index'
$found = $container->findAllByController('index'); // returns Page 2 and Page 3
```

195.5 Iterating containers

`Zend\Navigation\AbstractContainer` implements `RecursiveIterator`, and can be iterated using any `Iterator` class. To iterate a container recursively, use the `RecursiveIteratorIterator` class.

Iterating a container

```
/*
 * Create a container from an array
 */
$container = new Zend\Navigation\Navigation(array(
    array(
        'label' => 'Page 1',
        'uri'   => '#',
    ),
    array(
        'label' => 'Page 2',
        'uri'   => '#',
        'pages' => array(
            array(
                'label' => 'Page 2.1',
                'uri'   => '#',
            ),
            array(
                'label' => 'Page 2.2',
                'uri'   => '#',
            )
        )
    )
    array(
        'label' => 'Page 3',
        'uri'   => '#',
    ),
));

// Iterate flat using regular foreach:
// Output: Page 1, Page 2, Page 3
foreach ($container as $page) {
    echo $page->label;
}
```

```

// Iterate recursively using RecursiveIteratorIterator
$it = new RecursiveIteratorIterator(
        $container, RecursiveIteratorIterator::SELF_FIRST);

// Output: Page 1, Page 2, Page 2.1, Page 2.2, Page 3
foreach ($it as $page) {
    echo $page->label;
}
```

195.6 Other operations

The method `hasPage(Zend\Navigation\Page\AbstractPage $page)` checks if the container has the given page. The method `hasPages()` checks if there are any pages in the container, and is equivalent to `count($container) > 0`.

The `toArray()` method converts the container and the pages in it to an array. This can be useful for serializing and debugging.

Converting a container to an array

```
$container = new Zend\Navigation\Navigation(array(
    array(
        'label' => 'Page 1',
        'uri'   => '#',
    ),
    array(
        'label' => 'Page 2',
        'uri'   => '#',
        'pages' => array(
            array(
                'label' => 'Page 2.1',
                'uri'   => '#',
            ),
            array(
                'label' => 'Page 2.2',
               'uri'   => '#',
            ),
        ),
    ),
));

var_dump($container->toArray());

/* Output:
array(2) {
  [0]=> array(15) {
    ["label"]=> string(6) "Page 1"
    ["id"]=> NULL
    ["class"]=> NULL
    ["title"]=> NULL
    ["target"]=> NULL
    ["rel"]=> array(0) {
    }
    ["rev"]=> array(0) {
```

```
    }
    ["order"]=> NULL
    ["resource"]=> NULL
    ["privilege"]=> NULL
    ["active"]=> bool(false)
    ["visible"]=> bool(true)
    ["type"]=> string(23) "Zend\Navigation\Page\Uri"
    ["pages"]=> array(0) {
    }
    ["uri"]=> string(1) "#"
  }
  [1]=> array(15) {
    ["label"]=> string(6) "Page 2"
    ["id"]=> NULL
    ["class"]=> NULL
    ["title"]=> NULL
    ["target"]=> NULL
    ["rel"]=> array(0) {
    }
    ["rev"]=> array(0) {
    }
    ["order"]=> NULL
    ["resource"]=> NULL
    ["privilege"]=> NULL
    ["active"]=> bool(false)
    ["visible"]=> bool(true)
    ["type"]=> string(23) "Zend\Navigation\Page\Uri"
    ["pages"]=> array(2) {
      [0]=> array(15) {
        ["label"]=> string(8) "Page 2.1"
        ["id"]=> NULL
        ["class"]=> NULL
        ["title"]=> NULL
        ["target"]=> NULL
        ["rel"]=> array(0) {
        }
        ["rev"]=> array(0) {
        }
        ["order"]=> NULL
        ["resource"]=> NULL
        ["privilege"]=> NULL
        ["active"]=> bool(false)
        ["visible"]=> bool(true)
        ["type"]=> string(23) "Zend\Navigation\Page\Uri"
        ["pages"]=> array(0) {
        }
        ["uri"]=> string(1) "#"
      }
      [1]=>
      array(15) {
        ["label"]=> string(8) "Page 2.2"
        ["id"]=> NULL
        ["class"]=> NULL
        ["title"]=> NULL
        ["target"]=> NULL
        ["rel"]=> array(0) {
        }
        ["rev"]=> array(0) {
```

```
          }
          ["order"]=> NULL
          ["resource"]=> NULL
          ["privilege"]=> NULL
          ["active"]=> bool(false)
          ["visible"]=> bool(true)
          ["type"]=> string(23) "Zend\Navigation\Page\Uri"
          ["pages"]=> array(0) {
          }
          ["uri"]=> string(1) "#"
        }
      }
      ["uri"]=> string(1) "#"
    }
  }
*/
```

CHAPTER

SIX

VIEW HELPERS

The navigation helpers are used for rendering navigational elements from *Zend\Navigation\Navigation* instances.

There are 5 built-in helpers:

- *Breadcrumbs*, used for rendering the path to the currently active page.
- *Links*, used for rendering navigational head links (e.g. `<link rel="next" href="..." />`)
- *Menu*, used for rendering menus.
- *Sitemap*, used for rendering sitemaps conforming to the **'Sitemaps XML format'_**.
- *Navigation*, used for proxying calls to other navigational helpers.

All built-in helpers extend `Zend\View\Helper\Navigation\AbstractHelper`, which adds integration with *ACL* and *translation*. The abstract class implements the interface `Zend\View\Helper\Navigation\HelperInterface`, which defines the following methods:

- `getContainer()` and `setContainer()` gets and sets the navigation container the helper should operate on by default, and `hasContainer()` checks if the helper has container registered.
- `getTranslator()` and `setTranslator()` gets and sets the translator used for translating labels and titles. `getUseTranslator()` and `setUseTranslator()` controls whether the translator should be enabled. The method `hasTranslator()` checks if the helper has a translator registered.
- `getAcl()`, `setAcl()`, `getRole()` and `setRole()`, gets and sets *ACL* (`Zend\Permissions\Acl\AclInterface`) instance and role (`String` or `Zend\Permissions\Acl\Role\RoleInterface`) used for filtering out pages when rendering. `getUseAcl()` and `setUseAcl()` controls whether *ACL* should be enabled. The methods `hasAcl()` and `hasRole()` checks if the helper has an *ACL* instance or a role registered.
- `__toString()`, magic method to ensure that helpers can be rendered by echoing the helper instance directly.
- `render()`, must be implemented by concrete helpers to do the actual rendering.

In addition to the method stubs from the interface, the abstract class also implements the following methods:

- `getIndent()` and `setIndent()` gets and sets indentation. The setter accepts a `String` or an `Integer`. In the case of an `Integer`, the helper will use the given number of spaces for indentation. I.e., `setIndent(4)` means 4 initial spaces of indentation. Indentation can be specified for all helpers except the Sitemap helper.
- `getMinDepth()` and `setMinDepth()` gets and sets the minimum depth a page must have to be included by the helper. Setting `NULL` means no minimum depth.
- `getMaxDepth()` and `setMaxDepth()` gets and sets the maximum depth a page can have to be included by the helper. Setting `NULL` means no maximum depth.

- `getRenderInvisible()` and `setRenderInvisible()` gets and sets whether to render items that have been marked as invisible or not.
- `__call()` is used for proxying calls to the container registered in the helper, which means you can call methods on a helper as if it was a container. See *example* below.
- `findActive($container, $minDepth, $maxDepth)` is used for finding the deepest active page in the given container. If depths are not given, the method will use the values retrieved from `getMinDepth()` and `getMaxDepth()`. The deepest active page must be between `$minDepth` and `$maxDepth` inclusively. Returns an array containing a reference to the found page instance and the depth at which the page was found.
- `htmlify()` renders an **'a'** *HTML* element from a `Zend\Navigation\Page\AbstractPage` instance.
- `accept()` is used for determining if a page should be accepted when iterating containers. This method checks for page visibility and verifies that the helper's role is allowed access to the page's resource and privilege.
- The static method `setDefaultAcl()` is used for setting a default *ACL* object that will be used by helpers.
- The static method `setDefaultRole()` is used for setting a default *ACL* that will be used by helpers

If a container is not explicitly set, the helper will create an empty `Zend\Navigation\Navigation` container when calling `$helper->getContainer()`.

Proxying calls to the navigation container

Navigation view helpers use the magic method `__call()` to proxy method calls to the navigation container that is registered in the view helper.

```
$this->navigation()->addPage(array(
    'type' => 'uri',
    'label' => 'New page'));
```

The call above will add a page to the container in the `Navigation` helper.

196.1 Translation of labels and titles

The navigation helpers support translation of page labels and titles. You can set a translator of type `Zend\I18n\Translator` in the helper using `$helper->setTranslator($translator)`.

If you want to disable translation, use `$helper->setUseTranslator(false)`.

The *proxy helper* will inject its own translator to the helper it proxies to if the proxied helper doesn't already have a translator.

Note: There is no translation in the sitemap helper, since there are no page labels or titles involved in an *XML* sitemap.

196.2 Integration with ACL

All navigational view helpers support *ACL* inherently from the class `Zend\View\Helper\Navigation\AbstractHelper`. An object implementing `Zend\Permissions\Acl\AclInterface` can be assigned to a helper instance with *$helper->setAcl($acl)*, and role with *$helper->setRole('member')* or *$helper->setRole(new Zend\Permissions\Acl\Role\GenericRole('member'))*. If *ACL* is used in the helper, the role in the helper must be allowed by the *ACL* to access a page's *resource* and/or have the page's *privilege* for the page to be included when rendering.

If a page is not accepted by *ACL*, any descendant page will also be excluded from rendering.

The *proxy helper* will inject its own *ACL* and role to the helper it proxies to if the proxied helper doesn't already have any.

The examples below all show how *ACL* affects rendering.

196.3 Navigation setup used in examples

This example shows the setup of a navigation container for a fictional software company.

Notes on the setup:

- The domain for the site is *www.example.com*.
- Interesting page properties are marked with a comment.
- Unless otherwise is stated in other examples, the user is requesting the *URL http://www.example.com/products/server/faq/*, which translates to the page labeled `FAQ` under *Foo Server*.
- The assumed *ACL* and router setup is shown below the container setup.

```
/*
 * Navigation container (config/array)

 * Each element in the array will be passed to
 * Zend\Navigation\Page\AbstractPage::factory() when constructing
 * the navigation container below.
 */
$pages = array(
    array(
        'label'      => 'Home',
        'title'      => 'Go Home',
        'module'     => 'default',
        'controller' => 'index',
        'action'     => 'index',
        'order'      => -100 // make sure home is the first page
    ),
    array(
        'label'      => 'Special offer this week only!',
        'module'     => 'store',
        'controller' => 'offer',
        'action'     => 'amazing',
        'visible'    => false // not visible
    ),
    array(
        'label'      => 'Products',
        'module'     => 'products',
        'controller' => 'index',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'Foo Server',
                'module'     => 'products',
                'controller' => 'server',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'FAQ',
```

```
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'faq',
                        'rel'        => array(
                            'canonical' => 'http://www.example.com/?page=faq',
                            'alternate' => array(
                                'module'     => 'products',
                                'controller' => 'server',
                                'action'     => 'faq',
                                'params'     => array('format' => 'xml')
                            )
                        )
                    ),
                    array(
                        'label'      => 'Editions',
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'editions'
                    ),
                    array(
                        'label'      => 'System Requirements',
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'requirements'
                    )
                )
            ),
            array(
                'label'      => 'Foo Studio',
                'module'     => 'products',
                'controller' => 'studio',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'Customer Stories',
                        'module'     => 'products',
                        'controller' => 'studio',
                        'action'     => 'customers'
                    ),
                    array(
                        'label'      => 'Support',
                        'module'     => 'products',
                        'controller' => 'studio',
                        'action'     => 'support'
                    )
                )
            )
        )
    ),
    array(
        'label'      => 'Company',
        'title'      => 'About us',
        'module'     => 'company',
        'controller' => 'about',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'Investor Relations',
```

```
                'module'     => 'company',
                'controller' => 'about',
                'action'     => 'investors'
            ),
            array(
                'label'      => 'News',
                'class'      => 'rss', // class
                'module'     => 'company',
                'controller' => 'news',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'Press Releases',
                        'module'     => 'company',
                        'controller' => 'news',
                        'action'     => 'press'
                    ),
                    array(
                        'label'      => 'Archive',
                        'route'      => 'archive', // route
                        'module'     => 'company',
                        'controller' => 'news',
                        'action'     => 'archive'
                    )
                )
            )
        )
    ),
    array(
        'label'      => 'Community',
        'module'     => 'community',
        'controller' => 'index',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'My Account',
                'module'     => 'community',
                'controller' => 'account',
                'action'     => 'index',
                'resource'   => 'mvc:community.account' // resource
            ),
            array(
                'label' => 'Forums',
                'uri'   => 'http://forums.example.com/',
                'class' => 'external' // class
            )
        )
    ),
    array(
        'label'      => 'Administration',
        'module'     => 'admin',
        'controller' => 'index',
        'action'     => 'index',
        'resource'   => 'mvc:admin', // resource
        'pages'      => array(
            array(
                'label'      => 'Write new article',
                'module'     => 'admin',
```

```
                'controller' => 'post',
                'action'     => 'write'
            )
        )
    )
);

// Create container from array
$container = new Zend\Navigation\Navigation($pages);

// Store the container in the proxy helper:
$view->plugin('navigation')->setContainer($container);

// ...or simply:
$view->navigation($container);
```

In addition to the container above, the following setup is assumed:

```
<?php
// module/MyModule/config/module.config.php

return array(
    /* ... */
    'router' array(
        'routes' => array(
            'archive' => array(
                'type'    => 'Segment',
                'options' => array(
                    'route'    => '/archive/:year',
                    'defaults' => array(
                        'module'     => 'company',
                        'controller' => 'news',
                        'action'     => 'archive',
                        'year'       => (int) date('Y') - 1,
                    ),
                    'constraints' => array(
                        'year' => '\d+',
                    ),
                ),
            ),
            /* You can have other routes here... */
        ),
    ),
    /* ... */
);
```

```
<?php
// module/MyModule/Module.php

namespace MyModule;

use Zend\View\HelperPluginManager;
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole;
use Zend\Permissions\Acl\Resource\GenericResource;

class Module
{
```

```
    /* ... */
    public function getViewHelperConfig()
    {
        return array(
            'factories' => array(
                // This will overwrite the native navigation helper
                'navigation' => function(HelperPluginManager $pm) {
                    // Setup ACL:
                    $acl = new Acl();
                    $acl->addRole(new GenericRole('member'));
                    $acl->addRole(new GenericRole('admin'));
                    $acl->addResource(new GenericResource('mvc:admin'));
                    $acl->addResource(new GenericResource('mvc:community.account'));
                    $acl->allow('member', 'mvc:community.account');
                    $acl->allow('admin', null);

                    // Get an instance of the proxy helper
                    $navigation = $pm->get('Zend\View\Helper\Navigation');

                    // Store ACL and role in the proxy helper:
                    $navigation->setAcl($acl)
                               ->setRole('member');

                    // Return the new navigation helper instance
                    return $navigation;
                }
            )
        );
    }
    /* ... */
}
```

CHAPTER

SEVEN

VIEW HELPER - BREADCRUMBS

Breadcrumbs are used for indicating where in a sitemap a user is currently browsing, and are typically rendered like this: "You are here: Home > Products > FantasticProduct 1.0". The breadcrumbs helper follows the guidelines from Breadcrumbs Pattern - Yahoo! Design Pattern Library, and allows simple customization (minimum/maximum depth, indentation, separator, and whether the last element should be linked), or rendering using a partial view script.

The Breadcrumbs helper works like this; it finds the deepest active page in a navigation container, and renders an upwards path to the root. For *MVC* pages, the "activeness" of a page is determined by inspecting the request object, as stated in the section on *Zend\Navigation\Page\Mvc*.

The helper sets the *minDepth* property to 1 by default, meaning breadcrumbs will not be rendered if the deepest active page is a root page. If *maxDepth* is specified, the helper will stop rendering when at the specified depth (e.g. stop at level 2 even if the deepest active page is on level 3).

Methods in the breadcrumbs helper:

- *{get|set}Separator()* gets/sets separator string that is used between breadcrumbs. Default is *' > '*.
- *{get|set}LinkLast()* gets/sets whether the last breadcrumb should be rendered as an anchor or not. Default is `FALSE`.
- *{get|set}Partial()* gets/sets a partial view script that should be used for rendering breadcrumbs. If a partial view script is set, the helper's `render()` method will use the `renderPartial()` method. If no partial is set, the `renderStraight()` method is used. The helper expects the partial to be a `String` or an `Array` with two elements. If the partial is a `String`, it denotes the name of the partial script to use. If it is an `Array`, the first element will be used as the name of the partial view script, and the second element is the module where the script is found.
- `renderStraight()` is the default render method.
- `renderPartial()` is used for rendering using a partial view script.

Rendering breadcrumbs

This example shows how to render breadcrumbs with default settings.

```
In a view script or layout:
<?php echo $this->navigation()->breadcrumbs(); ?>

The two calls above take advantage of the magic __toString() method,
and are equivalent to:
<?php echo $this->navigation()->breadcrumbs()->render(); ?>

Output:
<a href="/products">Products</a> &gt; <a href="/products/server">Foo Server</a> &gt; FAQ
```

Specifying indentation

This example shows how to render breadcrumbs with initial indentation.

```
Rendering with 8 spaces indentation:
<?php echo $this->navigation()->breadcrumbs()->setIndent(8);?>

Output:
        <a href="/products">Products</a> &gt; <a href="/products/server">Foo Server</a> &gt; FAQ
```

Customize breadcrumbs output

This example shows how to customize breadcrumbs output by specifying various options.

```
In a view script or layout:

<?php
echo $this->navigation()
          ->breadcrumbs()
          ->setLinkLast(true)                   // link last page
          ->setMaxDepth(1)                      // stop at level 1
          ->setSeparator(' ' . PHP_EOL); // cool separator with newline
?>

Output:
<a href="/products">Products</a>
<a href="/products/server">Foo Server</a>

//////////////////////////////////////////////////////

Setting minimum depth required to render breadcrumbs:

<?php
$this->navigation()->breadcrumbs()->setMinDepth(10);
echo $this->navigation()->breadcrumbs();
?>

Output:
Nothing, because the deepest active page is not at level 10 or deeper.
```

Rendering breadcrumbs using a partial view script

This example shows how to render customized breadcrumbs using a partial vew script. By calling `setPartial()`, you can specify a partial view script that will be used when calling `render()`. When a partial is specified, the `renderPartial()` method will be called. This method will find the deepest active page and pass an array of pages that leads to the active page to the partial view script.

In a layout:

```
echo $this->navigation()->breadcrumbs()
                        ->setPartial('my-module/partials/breadcrumbs');
```

Contents of *module/MyModule/view/my-module/partials/breadcrumbs.phtml*:

```
echo implode(', ', array_map(
        function ($a) { return $a->getLabel(); },
        $this->pages));
```

Output:

```
Products, Foo Server, FAQ
```

CHAPTER

EIGHT

VIEW HELPER - LINKS

The links helper is used for rendering *HTML* `LINK` elements. Links are used for describing document relationships of the currently active page. Read more about links and link types at Document relationships: the LINK element (HTML4 W3C Rec.) and Link types (HTML4 W3C Rec.) in the *HTML*4 W3C Recommendation.

<<<<<<< HEAD There are two types of relations; forward and reverse, indicated by the kewyords *'rel'* and *'rev'*. Most methods ======= There are two types of relations; forward and reverse, indicated by the keywords *'rel'* and *'rev'*. Most methods >>>>>>> c6880ba87e0376b0608f2bfcd9d4c93d018e6bcf in the helper will take a `$rel` param, which must be either *'rel'* or *'rev'*. Most methods also take a `$type` param, which is used for specifying the link type (e.g. alternate, start, next, prev, chapter, etc).

Relationships can be added to page objects manually, or found by traversing the container registered in the helper. The method `findRelation($page, $rel, $type)` will first try to find the given `$rel` of `$type` from the `$page` by calling *$page->findRel($type)* or *$page->findRel($type)*. If the `$page` has a relation that can be converted to a page instance, that relation will be used. If the `$page` instance doesn't have the specified `$type`, the helper will look for a method in the helper named *searchreltype* (e.g. `searchRelNext()` or `searchRevAlternate()`). If such a method exists, it will be used for determining the `$page`'s relation by traversing the container.

Not all relations can be determined by traversing the container. These are the relations that will be found by searching:

- `searchRelStart()`, forward 'start' relation: the first page in the container.
- `searchRelNext()`, forward 'next' relation; finds the next page in the container, i.e. the page after the active page.
- `searchRelPrev()`, forward 'prev' relation; finds the previous page, i.e. the page before the active page.
- `searchRelChapter()`, forward 'chapter' relations; finds all pages on level 0 except the 'start' relation or the active page if it's on level 0.
- `searchRelSection()`, forward 'section' relations; finds all child pages of the active page if the active page is on level 0 (a 'chapter').
- `searchRelSubsection()`, forward 'subsection' relations; finds all child pages of the active page if the active pages is on level 1 (a 'section').
- `searchRevSection()`, reverse 'section' relation; finds the parent of the active page if the active page is on level 1 (a 'section').
- `searchRevSubsection()`, reverse 'subsection' relation; finds the parent of the active page if the active page is on level 2 (a 'subsection').

Note: When looking for relations in the page instance (*$page->getRel($type)* or *$page->getRev($type)*), the helper accepts the values of type `String`, `Array`, `Zend\Config`, or `Zend\Navigation\Page\AbstractPage`. If a string is found, it will be converted to a `Zend\Navigation\Page\Uri`. If an array or a config is found, it will be converted to one or several page instances. If the first key of the array/config is numeric, it will be considered

to contain several pages, and each element will be passed to the *page factory*. If the first key is not numeric, the array/config will be passed to the page factory directly, and a single page will be returned.

The helper also supports magic methods for finding relations. E.g. to find forward alternate relations, call *$helper->findRelAlternate($page)*, and to find reverse section relations, call *$helper->findRevSection($page)*. Those calls correspond to *$helper->findRelation($page, 'rel', 'alternate');* and *$helper->findRelation($page, 'rev', 'section');* respectively.

To customize which relations should be rendered, the helper uses a render flag. The render flag is an integer value, and will be used in a bitwise and (&) operation against the helper's render constants to determine if the relation that belongs to the render constant should be rendered.

See the *example below* for more information.

- `Zend\View\Helper\Navigation\Links::RENDER_ALTERNATE`
- `Zend\View\Helper\Navigation\Links::RENDER_STYLESHEET`
- `Zend\View\Helper\Navigation\Links::RENDER_START`
- `Zend\View\Helper\Navigation\Links::RENDER_NEXT`
- `Zend\View\Helper\Navigation\Links::RENDER_PREV`
- `Zend\View\Helper\Navigation\Links::RENDER_CONTENTS`
- `Zend\View\Helper\Navigation\Links::RENDER_INDEX`
- `Zend\View\Helper\Navigation\Links::RENDER_GLOSSARY`
- `Zend\View\Helper\Navigation\Links::RENDER_COPYRIGHT`
- `Zend\View\Helper\Navigation\Links::RENDER_CHAPTER`
- `Zend\View\Helper\Navigation\Links::RENDER_SECTION`
- `Zend\View\Helper\Navigation\Links::RENDER_SUBSECTION`
- `Zend\View\Helper\Navigation\Links::RENDER_APPENDIX`
- `Zend\View\Helper\Navigation\Links::RENDER_HELP`
- `Zend\View\Helper\Navigation\Links::RENDER_BOOKMARK`
- `Zend\View\Helper\Navigation\Links::RENDER_CUSTOM`
- `Zend\View\Helper\Navigation\Links::RENDER_ALL`

The constants from `RENDER_ALTERNATE` to `RENDER_BOOKMARK` denote standard *HTML* link types. `RENDER_CUSTOM` denotes non-standard relations that specified in pages. `RENDER_ALL` denotes standard and non-standard relations.

Methods in the links helper:

- *{get|set}RenderFlag()* gets/sets the render flag. Default is `RENDER_ALL`. See examples below on how to set the render flag.
- `findAllRelations()` finds all relations of all types for a given page.
- `findRelation()` finds all relations of a given type from a given page.
- *searchRel{Start|Next|Prev|Chapter|Section|Subsection}()* traverses a container to find forward relations to the start page, the next page, the previous page, chapters, sections, and subsections.
- *searchRev{Section|Subsection}()* traverses a container to find reverse relations to sections or subsections.
- `renderLink()` renders a single *link* element.

Specify relations in pages

This example shows how to specify relations in pages.

```
$container = new Zend\Navigation\Navigation(array(
    array(
        'label' => 'Relations using strings',
        'rel'   => array(
            'alternate' => 'http://www.example.org/'
        ),
        'rev'   => array(
            'alternate' => 'http://www.example.net/'
        )
    ),
    array(
        'label' => 'Relations using arrays',
        'rel'   => array(
            'alternate' => array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            )
        )
    ),
    array(
        'label' => 'Relations using configs',
        'rel'   => array(
            'alternate' => new Zend\Config\Config(array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            ))
        )
    ),
    array(
        'label' => 'Relations using pages instance',
        'rel'   => array(
            'alternate' => Zend\Navigation\Page\AbstractPage::factory(array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            ))
        )
    )
));
```

Default rendering of links

This example shows how to render a menu from a container registered/found in the view helper.

```
In a view script or layout:
<?php echo $this->view->navigation()->links(); ?>

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="chapter" href="/products" title="Products">
<link rel="chapter" href="/company/about" title="Company">
```

```
<link rel="chapter" href="/community" title="Community">
<link rel="canonical" href="http://www.example.com/?page=server-faq">
<link rev="subsection" href="/products/server" title="Foo Server">
```

Specify which relations to render

This example shows how to specify which relations to find and render.

```
Render only start, next, and prev:
$helper->setRenderFlag(Zend\View\Helper\Navigation\Links::RENDER_START |
                       Zend\View\Helper\Navigation\Links::RENDER_NEXT |
                       Zend\View\Helper\Navigation\Links::RENDER_PREV);

Output:
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
```

```
Render only native link types:
$helper->setRenderFlag(Zend\View\Helper\Navigation\Links::RENDER_ALL ^
                       Zend\View\Helper\Navigation\Links::RENDER_CUSTOM);

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="chapter" href="/products" title="Products">
<link rel="chapter" href="/company/about" title="Company">
<link rel="chapter" href="/community" title="Community">
<link rev="subsection" href="/products/server" title="Foo Server">
```

```
Render all but chapter:
$helper->setRenderFlag(Zend\View\Helper\Navigation\Links::RENDER_ALL ^
                       Zend\View\Helper\Navigation\Links::RENDER_CHAPTER);

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="canonical" href="http://www.example.com/?page=server-faq">
<link rev="subsection" href="/products/server" title="Foo Server">
```

CHAPTER

NINE

VIEW HELPER - MENU

The Menu helper is used for rendering menus from navigation containers. By default, the menu will be rendered using *HTML UL* and *LI* tags, but the helper also allows using a partial view script.

Methods in the Menu helper:

- *{get|set}UlClass()* gets/sets the *CSS* class used in `renderMenu()`.
- *{get|set}OnlyActiveBranch()* gets/sets a flag specifying whether only the active branch of a container should be rendered.
- *{get|set}RenderParents()* gets/sets a flag specifying whether parents should be rendered when only rendering active branch of a container. If set to `FALSE`, only the deepest active menu will be rendered.
- *{get|set}Partial()* gets/sets a partial view script that should be used for rendering menu. If a partial view script is set, the helper's `render()` method will use the `renderPartial()` method. If no partial is set, the `renderMenu()` method is used. The helper expects the partial to be a `String` or an `Array` with two elements. If the partial is a `String`, it denotes the name of the partial script to use. If it is an `Array`, the first element will be used as the name of the partial view script, and the second element is the module where the script is found.
- `htmlify()` overrides the method from the abstract class to return *span* elements if the page has no *href*.
- `renderMenu($container = null, $options = array())` is the default render method, and will render a container as a *HTML UL* list.

  If `$container` is not given, the container registered in the helper will be rendered.

  `$options` is used for overriding options specified temporarily without resetting the values in the helper instance. It is an associative array where each key corresponds to an option in the helper.

  Recognized options:

  - *indent*; indentation. Expects a `String` or an *int* value.
  - *minDepth*; minimum depth. Expects an *int* or `NULL` (no minimum depth).
  - *maxDepth*; maximum depth. Expects an *int* or `NULL` (no maximum depth).
  - *ulClass*; *CSS* class for *ul* element. Expects a `String`.
  - *onlyActiveBranch*; whether only active branch should be rendered. Expects a `Boolean` value.
  - *renderParents*; whether parents should be rendered if only rendering active branch. Expects a `Boolean` value.

  If an option is not given, the value set in the helper will be used.
- `renderPartial()` is used for rendering the menu using a partial view script.

- `renderSubMenu()` renders the deepest menu level of a container's active branch.

Rendering a menu

This example shows how to render a menu from a container registered/found in the view helper. Notice how pages are filtered out based on visibility and *ACL*.

```
In a view script or layout:
<?php echo $this->navigation()->menu()->render() ?>

Or simply:
<?php echo $this->navigation()->menu() ?>

Output:
<ul class="navigation">
    <li>
        <a title="Go Home" href="/">Home</a>
    </li>
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
                <ul>
                    <li class="active">
                        <a href="/products/server/faq">FAQ</a>
                    </li>
                    <li>
                        <a href="/products/server/editions">Editions</a>
                    </li>
                    <li>
                        <a href="/products/server/requirements">System Requirements</a>
                    </li>
                </ul>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
                <ul>
                    <li>
                        <a href="/products/studio/customers">Customer Stories</a>
                    </li>
                    <li>
                        <a href="/products/studio/support">Support</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
    <li>
        <a title="About us" href="/company/about">Company</a>
        <ul>
            <li>
                <a href="/company/about/investors">Investor Relations</a>
            </li>
            <li>
                <a class="rss" href="/company/news">News</a>
                <ul>
```

```
                    <li>
                        <a href="/company/news/press">Press Releases</a>
                    </li>
                    <li>
                        <a href="/archive">Archive</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community">Community</a>
        <ul>
            <li>
                <a href="/community/account">My Account</a>
            </li>
            <li>
                <a class="external" href="http://forums.example.com/">Forums</a>
            </li>
        </ul>
    </li>
</ul>
```

Calling renderMenu() directly

This example shows how to render a menu that is not registered in the view helper by calling the `renderMenu()` directly and specifying a few options.

```
<?php
// render only the 'Community' menu
$community = $this->navigation()->findOneByLabel('Community');
$options = array(
    'indent'  => 16,
    'ulClass' => 'community'
);
echo $this->navigation()
          ->menu()
          ->renderMenu($community, $options);
?>
Output:
                <ul class="community">
                    <li>
                        <a href="/community/account">My Account</a>
                    </li>
                    <li>
                        <a class="external" href="http://forums.example.com/">Forums</a>
                    </li>
                </ul>
```

Rendering the deepest active menu

This example shows how the `renderSubMenu()` will render the deepest sub menu of the active branch.

Calling `renderSubMenu($container, $ulClass, $indent)` is equivalent to calling `renderMenu($container, $options)` with the following options:

```
array(
    'ulClass'          => $ulClass,
    'indent'           => $indent,
    'minDepth'         => null,
    'maxDepth'         => null,
    'onlyActiveBranch' => true,
    'renderParents'    => false
);
```

```
<?php
echo $this->navigation()
          ->menu()
          ->renderSubMenu(null, 'sidebar', 4);
?>

The output will be the same if 'FAQ' or 'Foo Server' is active:
    <ul class="sidebar">
        <li class="active">
            <a href="/products/server/faq">FAQ</a>
        </li>
        <li>
            <a href="/products/server/editions">Editions</a>
        </li>
        <li>
            <a href="/products/server/requirements">System Requirements</a>
        </li>
    </ul>
```

Rendering a menu with maximum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setMaxDepth(1);
?>

Output:
<ul class="navigation">
    <li>
        <a title="Go Home" href="/">Home</a>
    </li>
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
            </li>
        </ul>
    </li>
    <li>
        <a title="About us" href="/company/about">Company</a>
        <ul>
            <li>
                <a href="/company/about/investors">Investor Relations</a>
```

```
            </li>
            <li>
                <a class="rss" href="/company/news">News</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community">Community</a>
        <ul>
            <li>
                <a href="/community/account">My Account</a>
            </li>
            <li>
                <a class="external" href="http://forums.example.com/">Forums</a>
            </li>
        </ul>
    </li>
</ul>
```

Rendering a menu with minimum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setMinDepth(1);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
        <ul>
            <li class="active">
                <a href="/products/server/faq">FAQ</a>
            </li>
            <li>
                <a href="/products/server/editions">Editions</a>
            </li>
            <li>
                <a href="/products/server/requirements">System Requirements</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/products/studio">Foo Studio</a>
        <ul>
            <li>
                <a href="/products/studio/customers">Customer Stories</a>
            </li>
            <li>
                <a href="/products/studio/support">Support</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/company/about/investors">Investor Relations</a>
    </li>
```

```
    <li>
        <a class="rss" href="/company/news">News</a>
        <ul>
            <li>
                <a href="/company/news/press">Press Releases</a>
            </li>
            <li>
                <a href="/archive">Archive</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community/account">My Account</a>
    </li>
    <li>
        <a class="external" href="http://forums.example.com/">Forums</a>
    </li>
</ul>
```

Rendering only the active branch of a menu

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
                <ul>
                    <li class="active">
                        <a href="/products/server/faq">FAQ</a>
                    </li>
                    <li>
                        <a href="/products/server/editions">Editions</a>
                    </li>
                    <li>
                        <a href="/products/server/requirements">System Requirements</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
</ul>
```

Rendering only the active branch of a menu with minimum depth

```
<?php
echo $this->navigation()
          ->menu()
```

```
          ->setOnlyActiveBranch(true)
          ->setMinDepth(1);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
        <ul>
            <li class="active">
                <a href="/products/server/faq">FAQ</a>
            </li>
            <li>
                <a href="/products/server/editions">Editions</a>
            </li>
            <li>
                <a href="/products/server/requirements">System Requirements</a>
            </li>
        </ul>
    </li>
</ul>
```

Rendering only the active branch of a menu with maximum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setMaxDepth(1);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
            </li>
        </ul>
    </li>
</ul>
```

Rendering only the active branch of a menu with maximum depth and no parents

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setRenderParents(false)
          ->setMaxDepth(1);
?>
```

```

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
    </li>
    <li>
        <a href="/products/studio">Foo Studio</a>
    </li>
</ul>
```

Rendering a custom menu using a partial view script

This example shows how to render a custom menu using a partial view script. By calling `setPartial()`, you can specify a partial view script that will be used when calling `render()`. When a partial is specified, the `renderPartial()` method will be called. This method will assign the container to the view with the key *container*.

In a layout:

```
$this->navigation()->menu()->setPartial('my-module/partials/menu');
echo $this->navigation()->menu()->render();
```

In *module/MyModule/view/my-module/partials/menu.phtml*:

```
foreach ($this->container as $page) {
    echo $this->navigation()->menu()->htmlify($page) . PHP_EOL;
}
```

Output:

```
<a title="Go Home" href="/">Home</a>
<a href="/products">Products</a>
<a title="About us" href="/company/about">Company</a>
<a href="/community">Community</a>
```

CHAPTER

VIEW HELPER - SITEMAP

The Sitemap helper is used for generating *XML* sitemaps, as defined by the Sitemaps XML format. Read more about Sitemaps on Wikipedia.

By default, the sitemap helper uses *sitemap validators* to validate each element that is rendered. This can be disabled by calling *$helper->setUseSitemapValidators(false)*.

Note: If you disable sitemap validators, the custom properties (see table) are not validated at all.

The sitemap helper also supports Sitemap XSD Schema validation of the generated sitemap. This is disabled by default, since it will require a request to the Schema file. It can be enabled with *$helper->setUseSchemaValidation(true)*.

Table 200.1: Sitemap XML elements

Element	Description
loc	Absolute URL to page. An absolute URL will be generated by the helper.
lastmod	The date of last modification of the file, in W3C Datetime format. This time portion can be omitted if desired, and only use YYYY-MM-DD. The helper will try to retrieve the lastmod value from the page's custom property lastmod if it is set in the page. If the value is not a valid date, it is ignored.
changefreq	How frequently the page is likely to change. This value provides general information to search engines and may not correlate exactly to how often they crawl the page. Valid values are: alwayshourlydailyweeklymonthlyyearlynever The helper will try to retrieve the changefreq value from the page's custom property changefreq if it is set in the page. If the value is not valid, it is ignored.
priority	The priority of this URL relative to other URLs on your site. Valid values range from 0.0 to 1.0. The helper will try to retrieve the priority value from the page's custom property priority if it is set in the page. If the value is not valid, it is ignored.

Methods in the sitemap helper:

- *{get\|set}FormatOutput()* gets/sets a flag indicating whether *XML* output should be formatted. This corresponds to the *formatOutput* property of the native `DOMDocument` class. Read more at PHP: DOMDocument - Manual. Default is `FALSE`.
- *{get\|set}UseXmlDeclaration()* gets/sets a flag indicating whether the *XML* declaration should be included when rendering. Default is `TRUE`.
- *{get\|set}UseSitemapValidators()* gets/sets a flag indicating whether sitemap validators should be used when generating the DOM sitemap. Default is `TRUE`.
- *{get\|set}UseSchemaValidation()* gets/sets a flag indicating whether the helper should use *XML* Schema validation when generating the DOM sitemap. Default is `FALSE`. If `TRUE`.
- *{get\|set}ServerUrl()* gets/sets server *URL* that will be prepended to non-absolute *URL*s in the `url()` method. If no server *URL* is specified, it will be determined by the helper.

- `url()` is used to generate absolute *URL*s to pages.
- `getDomSitemap()` generates a DOMDocument from a given container.

Rendering an XML sitemap

This example shows how to render an *XML* sitemap based on the setup we did further up.

```
// In a view script or layout:

// format output
$this->navigation()
      ->sitemap()
      ->setFormatOutput(true); // default is false

// other possible methods:
// ->setUseXmlDeclaration(false); // default is true
// ->setServerUrl('http://my.otherhost.com');
// default is to detect automatically

// print sitemap
echo $this->navigation()->sitemap();
```

Notice how pages that are invisible or pages with *ACL* roles incompatible with the view helper are filtered out:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/faq</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/editions</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/requirements</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio/customers</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio/support</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about</loc>
  </url>
  <url>
```

```
    <loc>http://www.example.com/company/about/investors</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news/press</loc>
  </url>
  <url>
    <loc>http://www.example.com/archive</loc>
  </url>
  <url>
    <loc>http://www.example.com/community</loc>
  </url>
  <url>
    <loc>http://www.example.com/community/account</loc>
  </url>
  <url>
    <loc>http://forums.example.com/</loc>
  </url>
</urlset>
```

Render the sitemap using no *ACL* role (should filter out /community/account):

```
echo $this->navigation()
          ->sitemap()
          ->setFormatOutput(true)
          ->setRole();
```

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/faq</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/editions</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/requirements</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio/customers</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio/support</loc>
  </url>
  <url>
```

```
    <loc>http://www.example.com/company/about</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about/investors</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news/press</loc>
  </url>
  <url>
    <loc>http://www.example.com/archive</loc>
  </url>
  <url>
    <loc>http://www.example.com/community</loc>
  </url>
  <url>
    <loc>http://forums.example.com/</loc>
  </url>
</urlset>
```

Render the sitemap using a maximum depth of 1.

```
echo $this->navigation()
          ->sitemap()
          ->setFormatOutput(true)
          ->setMaxDepth(1);
```

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about/investors</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news</loc>
  </url>
  <url>
    <loc>http://www.example.com/community</loc>
  </url>
  <url>
    <loc>http://www.example.com/community/account</loc>
  </url>
  <url>
```

```
    <loc>http://forums.example.com/</loc>
  </url>
</urlset>
```

Note: UTF-8 encoding used by default

By default, Zend Framework uses *UTF-8* as its default encoding, and, specific to this case, `Zend\View` does as well. Character encoding can be set differently on the view object itself using the `setEncoding()` method (or the the `encoding` instantiation parameter). However, since `Zend\View\Interface` does not define accessors for encoding, it's possible that if you are using a custom view implementation with the Dojo view helper, you will not have a `getEncoding()` method, which is what the view helper uses internally for determining the character set in which to encode.

If you do not want to utilize *UTF-8* in such a situation, you will need to implement a `getEncoding()` method in your custom view implementation.

CHAPTER

ONE

VIEW HELPER - NAVIGATION PROXY

The Navigation helper is a proxy helper that relays calls to other navigational helpers. It can be considered an entry point to all navigation-related view tasks. The aforementioned navigational helpers are in the namespace `Zend\View\Helper\Navigation`, and would thus require the path *Zend/View/Helper/Navigation* to be added as a helper path to the view. With the proxy helper residing in the `Zend\View\Helper` namespace, it will always be available, without the need to add any helper paths to the view.

The Navigation helper finds other helpers that implement the `Zend\View\Helper\Navigation\HelperInterface`, which means custom view helpers can also be proxied. This would, however, require that the custom helper path is added to the view.

When proxying to other helpers, the Navigation helper can inject its container, *ACL*/role, and translator. This means that you won't have to explicitly set all three in all navigational helpers, nor resort to injecting by means of static methods.

- `findHelper()` finds the given helper, verifies that it is a navigational helper, and injects container, *ACL*/role and translator.
- *{get|set}InjectContainer()* gets/sets a flag indicating whether the container should be injected to proxied helpers. Default is `TRUE`.
- *{get|set}InjectAcl()* gets/sets a flag indicating whether the *ACL*/role should be injected to proxied helpers. Default is `TRUE`.
- *{get|set}InjectTranslator()* gets/sets a flag indicating whether the translator should be injected to proxied helpers. Default is `TRUE`.
- *{get|set}DefaultProxy()* gets/sets the default proxy. Default is *'menu'*.
- `render()` proxies to the render method of the default proxy.

CHAPTER

TWO

INTRODUCTION TO ZEND\PAGINATOR

`Zend\Paginator` is a flexible component for paginating collections of data and presenting that data to users.

The primary design goals of `Zend\Paginator` are as follows:

- Paginate arbitrary data, not just relational databases
- Fetch only the results that need to be displayed
- Do not force users to adhere to only one way of displaying data or rendering pagination controls
- Loosely couple `Zend\Paginator` to other Zend Framework components so that users who wish to use it independently of `Zend\View`, `Zend\Db`, etc. can do so

CHAPTER

THREE

USAGE

203.1 Paginating data collections

In order to paginate items into pages, `Zend\Paginator` must have a generic way of accessing that data. For that reason, all data access takes place through data source adapters. Several adapters ship with Zend Framework by default:

Table 203.1: Adapters for Zend\Paginator

Adapter	Description
ArrayAdapter	Accepts a PHP array
DbSelect	Accepts a Zend\Db\Sql\Select plus either a Zend\Db\Adapter\Adapter or Zend\Db\Sql\Sql to paginate rows from a database
Iterator	Accepts an Iterator instance
Null	Does not use Zend\Paginator to manage data pagination. You can still take advantage of the pagination control feature.

Note: Instead of selecting every matching row of a given query, the DbSelect adapter retrieves only the smallest amount of data necessary for displaying the current page. Because of this, a second query is dynamically generated to determine the total number of matching rows.

To create an instance of `Zend\Paginator`, you must supply an adapter to the constructor:

```
$paginator = new \Zend\Paginator\Paginator(new \Zend\Paginator\Adapter\ArrayAdapter($array));
```

In the case of the `Null` adapter, in lieu of a data collection you must supply an item count to its constructor.

Although the instance is technically usable in this state, in your controller action you'll need to tell the paginator what page number the user requested. This allows advancing through the paginated data.

```
$paginator->setCurrentPageNumber($page);
```

The simplest way to keep track of this value is through a *URL* parameter. The following is an example route you might use in an *Array* configuration file:

```
return array(
    'routes' => array(
        'paginator' => array(
            'type' => 'segment',
            'options' => array(
                'route' => '/list/[page/:page]',
                'defaults' => array(
```

```
                        'page' => 1,
                    ),
                ),
            ),
        ),
);
```

With the above route (and using Zend Framework *MVC* components), you might set the current page number in your controller action like so:

```
$paginator->setCurrentPageNumber($this->params()->fromRoute('page'));
```

There are other options available; see *Configuration* for more on them.

Finally, you'll need to assign the paginator instance to your view. If you're using Zend Framework *MVC* component, you can assign the paginator object to your view model:

```
$vm = new ViewModel();
$vm->setVariable('paginator', $paginator);
return $vm;
```

203.2 The DbSelect adapter

The usage of most adapters is pretty straight-forward. However, the database adapter requires a more detailed explanation regarding the retrieval and count of the data from the database.

To use the DbSelect adapter you don't have to retrieve the data upfront from the database. The adapter will do the retrieval for you, as well as the counting of the total pages. If additional work has to be done on the database results which cannot be expressed via the provided `Zend\Db\Sql\Select` object you must extend the adapter and override the `getItems()` method.

Additionally this adapter does **not** fetch all records from the database in order to count them. Instead, the adapter manipulates the original query to produce a corresponding COUNT query. Paginator then executes that COUNT query to get the number of rows. This does require an extra round-trip to the database, but this is many times faster than fetching an entire result set and using `count()`, especially with large collections of data.

The database adapter will try and build the most efficient query that will execute on pretty much any modern database. However, depending on your database or even your own schema setup, there might be more efficient ways to get a rowcount. For this scenario, you can extend the provided DbSelect adapter and implement a custom `getRowCount` method. For example, if you keep track of the count of blog posts in a separate table, you could achieve a faster count query with the following setup:

```
class MyDbSelect extends Zend\Paginator\Adapter\DbSelect
{
    public function count()
    {
        $select = new Zend\Db\Sql\Select();
        $select->from('item_counts')->columns(array('c'=>'post_count'));

        $statement = $this->sql->prepareStatementForSqlObject($select);
        $result    = $statement->execute();
        $row       = $result->current();
        $this->rowCount = $row['c'];

        return $this->rowCount;
    }
}
```

```

$adapter = new MyDbSelect($query, $adapter);
$paginator = new Zend\Paginator\Paginator($adapter);
```

This approach will probably not give you a huge performance gain on small collections and/or simple select queries. However, with complex queries and large collections, a similar approach could give you a significant performance boost.

The DbSelect adapter also supports returning of fetched records using the `Zend\Db\ResultSet` component of `Zend\Db`. You can override the concrete RowSet implementation by passing an object implementing `Zend\Db\ResultSet\ResultSetInterface` as the third constructor argument to the DbSelect adapter:

```
// $objectPrototype is an instance of our custom entity
// $hydrator is a custom hydrator for our entity (implementing Zend\Stdlib\Hydrator\HydratorInterfac
$resultSet = new Zend\Db\ResultSet\HydratingResultSet($hydrator, $objectPrototype);

$adapter = new Zend\Paginator\Adapter\DbSelect($query, $dbAdapter, $resultSet)
$paginator = new Zend\Paginator\Paginator($adapter);
```

Now when we iterate over `$paginator` we will get instances of our custom entity instead of key-value-pair arrays.

203.3 Rendering pages with view scripts

The view script is used to render the page items (if you're using `Zend\Paginator` to do so) and display the pagination control.

Because `Zend\Paginator` implements the *SPL* interface IteratorAggregate, looping over your items and displaying them is simple.

```
<html>
<body>
<h1>Example</h1>
<?php if (count($this->paginator)): ?>
<ul>
<?php foreach ($this->paginator as $item): ?>
  <li><?php echo $item; ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>

<?php echo $this->paginationControl($this->paginator,
                                    'Sliding',
                                    'my_pagination_control', array('route' => 'application/paginator'
</body>
</html>
```

Notice the view helper call near the end. PaginationControl accepts up to four parameters: the paginator instance, a scrolling style, a view script name, and an array of additional parameters.

The second and third parameters are very important. Whereas the view script name is used to determine how the pagination control should **look**, the scrolling style is used to control how it should **behave**. Say the view script is in the style of a search pagination control, like the one below:

What happens when the user clicks the "next" link a few times? Well, any number of things could happen. The current page number could stay in the middle as you click through (as it does on Yahoo!), or it could advance to the end of

the page range and then appear again on the left when the user clicks "next" one more time. The page numbers might even expand and contract as the user advances (or "scrolls") through them (as they do on Google).

There are four scrolling styles packaged with Zend Framework:

Table 203.2: Scrolling styles for Zend\Paginator

Scrolling style	Description
All	Returns every page. This is useful for dropdown menu pagination controls with relatively few pages. In these cases, you want all pages available to the user at once.
Elastic	A Google-like scrolling style that expands and contracts as a user scrolls through the pages.
Jumping	As users scroll through, the page number advances to the end of a given range, then starts again at the beginning of the new range.
Sliding	A Yahoo!-like scrolling style that positions the current page number in the center of the page range, or as close as possible. This is the default style.

The fourth and final parameter is reserved for an optional associative array of additional variables that you want available in your view (available via `$this`). For instance, these values could include extra *URL* parameters for pagination links.

By setting the default view script name, default scrolling style, and view instance, you can eliminate the calls to PaginationControl completely:

```
Zend\Paginator\Paginator::setDefaultScrollingStyle('Sliding');
Zend\View\Helper\PaginationControl::setDefaultViewPartial(
    'my_pagination_control'
);
```

When all of these values are set, you can render the pagination control inside your view script with a simple echo statement:

```
<?php echo $this->paginator; ?>
```

Note: Of course, it's possible to use `Zend\Paginator` with other template engines. For example, with Smarty you might do the following:

```
$smarty->assign('pages', $paginator->getPages());
```

You could then access paginator values from a template like so:

```
{$pages->pageCount}
```

203.3.1 Example pagination controls

The following example pagination controls will hopefully help you get started:

Search pagination:

```
<!--
See http://developer.yahoo.com/ypatterns/pattern.php?pattern=searchpagination
-->

<?php if ($this->pageCount): ?>
<div class="paginationControl">
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
```

```
  <a href="<?php echo $this->url($this->route, array('page' => $this->previous)); ?>">
    &lt; Previous
  </a> |
<?php else: ?>
  <span class="disabled">&lt; Previous</span> |
<?php endif; ?>

<!-- Numbered page links -->
<?php foreach ($this->pagesInRange as $page): ?>
  <?php if ($page != $this->current): ?>
    <a href="<?php echo $this->url($this->route, array('page' => $page)); ?>">
        <?php echo $page; ?>
    </a> |
  <?php else: ?>
    <?php echo $page; ?> |
  <?php endif; ?>
<?php endforeach; ?>

<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url($this->route, array('page' => $this->next)); ?>">
    Next &gt;
  </a>
<?php else: ?>
  <span class="disabled">Next &gt;</span>
<?php endif; ?>
</div>
<?php endif; ?>
```

Item pagination:

```
<!--
See http://developer.yahoo.com/ypatterns/pattern.php?pattern=itempagination
-->

<?php if ($this->pageCount): ?>
<div class="paginationControl">
<?php echo $this->firstItemNumber; ?> - <?php echo $this->lastItemNumber; ?>
of <?php echo $this->totalItemCount; ?>

<!-- First page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url($this->route, array('page' => $this->first)); ?>">
    First
  </a> |
<?php else: ?>
  <span class="disabled">First</span> |
<?php endif; ?>

<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url($this->route, array('page' => $this->previous)); ?>">
    &lt; Previous
  </a> |
<?php else: ?>
  <span class="disabled">&lt; Previous</span> |
<?php endif; ?>

<!-- Next page link -->
```

```
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url($this->route, array('page' => $this->next)); ?>">
    Next &gt;
  </a> |
<?php else: ?>
  <span class="disabled">Next &gt;</span> |
<?php endif; ?>

<!-- Last page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url($this->route, array('page' => $this->last)); ?>">
    Last
  </a>
<?php else: ?>
  <span class="disabled">Last</span>
<?php endif; ?>

</div>
<?php endif; ?>
```

Dropdown pagination:

```
<?php if ($this->pageCount): ?>
<select id="paginationControl" size="1">
<?php foreach ($this->pagesInRange as $page): ?>
  <?php $selected = ($page == $this->current) ? ' selected="selected"' : ''; ?>
  <option value="<?php
        echo $this->url($this->route, array('page' => $page));?>"<?php echo $selected ?>>
    <?php echo $page; ?>
  </option>
<?php endforeach; ?>
</select>
<?php endif; ?>

<script type="text/javascript"
     src="http://ajax.googleapis.com/ajax/libs/prototype/1.6.0.2/prototype.js">
</script>
<script type="text/javascript">
$('paginationControl').observe('change', function() {
    window.location = this.options[this.selectedIndex].value;
})
</script>
```

203.3.2 Listing of properties

The following options are available to pagination control view scripts:

Table 203.3: Properties available to view partials

Property	Type	Description
first	integer	First page number (i.e., 1)
firstItemNumber	integer	Absolute number of the first item on this page
firstPageInRange	integer	First page in the range returned by the scrolling style
current	integer	Current page number
currentItemCount	integer	Number of items on this page
itemCountPerPage	integer	Maximum number of items available to each page
last	integer	Last page number
lastItemNumber	integer	Absolute number of the last item on this page
lastPageInRange	integer	Last page in the range returned by the scrolling style
next	integer	Next page number
pageCount	integer	Number of pages
pagesInRange	array	Array of pages returned by the scrolling style
previous	integer	Previous page number
totalItemCount	integer	Total number of items

CHAPTER

FOUR

CONFIGURATION

`Zend\Paginator` has several configuration methods that can be called:

Table 204.1: Configuration methods for Zend\Paginator\Paginator

Method	Description
setCurrentPageNumber	Sets the current page number (default 1).
setItemCountPerPage	Sets the maximum number of items to display on a page (default 10).
setPageRange	Sets the number of items to display in the pagination control (default 10). Note: Most of the time this number will be adhered to exactly, but scrolling styles do have the option of only using it as a guideline or starting value (e.g., Elastic).
setView	Sets the view instance, for rendering convenience.

CHAPTER

FIVE

ADVANCED USAGE

205.1 Custom data source adapters

At some point you may run across a data type that is not covered by the packaged adapters. In this case, you will need to write your own.

To do so, you must implement `Zend\Paginator\Adapter\AdapterInterface`. There are two methods required to do this:

- count()
- getItems($offset, $itemCountPerPage)

Additionally, you'll want to implement a constructor that takes your data source as a parameter and stores it as a protected or private property. How you wish to go about doing this specifically is up to you.

If you've ever used the SPL interface Countable, you're familiar with `count()`. As used with `Zend\Paginator`, this is the total number of items in the data collection. Additionally, the `Zend\Paginator\Paginator` instance provides a method `countAllItems()` that proxies to the adapter `count()` method.

The `getItems()` method is only slightly more complicated. For this, your adapter is supplied with an offset and the number of items to display per page. You must return the appropriate slice of data. For an array, that would be:

```
return array_slice($this->_array, $offset, $itemCountPerPage);
```

Take a look at the packaged adapters (all of which implement the `Zend\Paginator\Adapter\AdapterInterface`) for ideas of how you might go about implementing your own.

205.2 Custom scrolling styles

Creating your own scrolling style requires that you implement `Zend\Paginator\ScrollingStyle\ScrollingStyleInterf` which defines a single method, `getPages()`. Specifically,

```
public function getPages(Zend\Paginator\Paginator $paginator, $pageRange = null);
```

This method should calculate a lower and upper bound for page numbers within the range of so-called "local" pages (that is, pages that are nearby the current page).

Unless it extends another scrolling style (see `Zend\Paginator\ScrollingStyle\Elastic` for an example), your custom scrolling style will inevitably end with something similar to the following line of code:

```
return $paginator->getPagesInRange($lowerBound, $upperBound);
```

There's nothing special about this call; it's merely a convenience method to check the validity of the lower and upper bound and return an array of the range to the paginator.

When you're ready to use your new scrolling style, you'll need to tell `Zend\Paginator\Paginator` what directory to look in. To do that, do the following:

```
$manager = Zend\Paginator\Paginator::getScrollingStyleManager();
$manager->setInvokableClass('my-style', 'My\Paginator\ScrollingStyle');
```

205.3 Caching features

`Zend\Paginator\Paginator` can be told to cache the data it has already passed on, preventing the adapter from fetching them each time they are used. To tell paginator to automatically cache the adapter's data, just pass to its `setCache()` method a pre-configured cache object implementing the `Zend\Cache\Storage\StorageInterface` interface.

```
$cache = StorageFactory::adapterFactory('filesystem', array(
    'cache_dir' => '/tmp',
    'ttl'       => 3600,
    'plugins'   => array( 'serializer' ),
));
Zend\Paginator\Paginator::setCache($cache);
```

As long as `Zend\Paginator\Paginator` has been seeded with a cache storage object the data it generates will be cached. Sometimes you would like not to cache data even if you already passed a cache instance. You should then use `setCacheEnable()` for that.

```
// $cache is a Zend\Cache\Storage\StorageInterface instance
Zend\Paginator\Paginator::setCache($cache);
// ... later on the script
$paginator->setCacheEnable(false);
// cache is now disabled
```

When a cache is set, data are automatically stored in it and pulled out from it. It then can be useful to empty the cache manually. You can get this done by calling `clearPageItemCache($pageNumber)`. If you don't pass any parameter, the whole cache will be empty. You can optionally pass a parameter representing the page number to empty in the cache:

```
// $cache is a Zend\Cache\Storage\StorageInterface instance
Zend\Paginator\Paginator::setCache($cache);
// $paginator is a fully configured Zend\Paginator\Paginator instance
$items = $paginator->getCurrentItems();
// page 1 is now in cache
$page3Items = $paginator->getItemsByPage(3);
// page 3 is now in cache

// clear the cache of the results for page 3
$paginator->clearPageItemCache(3);

// clear all the cache data
$paginator->clearPageItemCache();
```

Changing the item count per page will empty the whole cache as it would have become invalid:

```
// $cache is a Zend\Cache\Storage\StorageInterface instance
Zend\Paginator\Paginator::setCache($cache);
// fetch some items
```

```
// $paginator is a fully configured Zend\Paginator\Paginator instance
$items = $paginator->getCurrentItems();

// all the cache data will be flushed:
$paginator->setItemCountPerPage(2);
```

It is also possible to see the data in cache and ask for them directly. `getPageItemCache()` can be used for that:

```
// $cache is a Zend\Cache\Storage\StorageInterface instance
Zend\Paginator\Paginator::setCache($cache);
// $paginator is a fully configured Zend\Paginator\Paginator instance
$paginator->setItemCountPerPage(3);
// fetch some items
$items = $paginator->getCurrentItems();
$otherItems = $paginator->getItemsPerPage(4);

// see the cached items as a two-dimension array:
var_dump($paginator->getPageItemCache());
```

CHAPTER

SIX

INTRODUCTION TO ZEND\PERMISSIONS\ACL

The `Zend\Permissions\Acl` component provides a lightweight and flexible access control list (*ACL*) implementation for privileges management. In general, an application may utilize such *ACL*'s to control access to certain protected objects by other requesting objects.

For the purposes of this documentation:

- a **resource** is an object to which access is controlled.
- a **role** is an object that may request access to a Resource.

Put simply, **roles request access to resources**. For example, if a parking attendant requests access to a car, then the parking attendant is the requesting role, and the car is the resource, since access to the car may not be granted to everyone.

Through the specification and use of an *ACL*, an application may control how roles are granted access to resources.

206.1 Resources

Creating a resource using `Zend\Permissions\Acl\Acl` is very simple. A resource interface `Zend\Permissions\Acl\Resource\ResourceInterface` is provided to facilitate creating resources in an application. A class need only implement this interface, which consists of a single method, `getResourceId()`, for `Zend\Permissions\Acl\Acl` to recognize the object as a resource. Additionally, `Zend\Permissions\Acl\Resource\GenericResource` is provided as a basic resource implementation for developers to extend as needed.

`Zend\Permissions\Acl\Acl` provides a tree structure to which multiple resources can be added. Since resources are stored in such a tree structure, they can be organized from the general (toward the tree root) to the specific (toward the tree leaves). Queries on a specific resource will automatically search the resource's hierarchy for rules assigned to ancestor resources, allowing for simple inheritance of rules. For example, if a default rule is to be applied to each building in a city, one would simply assign the rule to the city, instead of assigning the same rule to each building. Some buildings may require exceptions to such a rule, however, and this can be achieved in `Zend\Permissions\Acl\Acl` by assigning such exception rules to each building that requires such an exception. A resource may inherit from only one parent resource, though this parent resource can have its own parent resource, etc.

`Zend\Permissions\Acl\Acl` also supports privileges on resources (e.g., "create", "read", "update", "delete"), so the developer can assign rules that affect all privileges or specific privileges on one or more resources.

206.2 Roles

As with resources, creating a role is also very simple. All roles must implement `Zend\Permissions\Acl\Role\RoleInterface`. This interface consists of a single method, `getRoleId()`, Additionally, `Zend\Permissions\Acl\Role\GenericRole` is provided by the `Zend\Permissions\Acl` component as a basic role implementation for developers to extend as needed.

In `Zend\Permissions\Acl\Acl`, a role may inherit from one or more roles. This is to support inheritance of rules among roles. For example, a user role, such as "sally", may belong to one or more parent roles, such as "editor" and "administrator". The developer can assign rules to "editor" and "administrator" separately, and "sally" would inherit such rules from both, without having to assign rules directly to "sally".

Though the ability to inherit from multiple roles is very useful, multiple inheritance also introduces some degree of complexity. The following example illustrates the ambiguity condition and how `Zend\Permissions\Acl\Acl` solves it.

Multiple Inheritance among Roles

The following code defines three base roles - "guest", "member", and "admin" - from which other roles may inherit. Then, a role identified by "someUser" is established and inherits from the three other roles. The order in which these roles appear in the `$parents` array is important. When necessary, `Zend\Permissions\Acl\Acl` searches for access rules defined not only for the queried role (herein, "someUser"), but also upon the roles from which the queried role inherits (herein, "guest", "member", and "admin"):

```
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;
use Zend\Permissions\Acl\Resource\GenericResource as Resource;

$acl = new Acl();

$acl->addRole(new Role('guest'))
    ->addRole(new Role('member'))
    ->addRole(new Role('admin'));

$parents = array('guest', 'member', 'admin');
$acl->addRole(new Role('someUser'), $parents);

$acl->addResource(new Resource('someResource'));

$acl->deny('guest', 'someResource');
$acl->allow('member', 'someResource');

echo $acl->isAllowed('someUser', 'someResource') ? 'allowed' : 'denied';
```

Since there is no rule specifically defined for the "someUser" role and "someResource", `Zend\Permissions\Acl\Acl` must search for rules that may be defined for roles that "someUser" inherits. First, the "admin" role is visited, and there is no access rule defined for it. Next, the "member" role is visited, and `Zend\Permissions\Acl\Acl` finds that there is a rule specifying that "member" is allowed access to "someResource".

If `Zend\Permissions\Acl\Acl` were to continue examining the rules defined for other parent roles, however, it would find that "guest" is denied access to "someResource". This fact introduces an ambiguity because now "someUser" is both denied and allowed access to "someResource", by reason of having inherited conflicting rules from different parent roles.

`Zend\Permissions\Acl\Acl` resolves this ambiguity by completing a query when it finds the first rule that is

directly applicable to the query. In this case, since the "member" role is examined before the "guest" role, the example code would print "allowed".

Note: When specifying multiple parents for a role, keep in mind that the last parent listed is the first one searched for rules applicable to an authorization query.

206.3 Creating the Access Control List

An Access Control List (*ACL*) can represent any set of physical or virtual objects that you wish. For the purposes of demonstration, however, we will create a basic Content Management System (*CMS*) *ACL* that maintains several tiers of groups over a wide variety of areas. To create a new *ACL* object, we instantiate the *ACL* with no parameters:

```
use Zend\Permissions\Acl\Acl;
$acl = new Acl();
```

Note: Until a developer specifies an "allow" rule, `Zend\Permissions\Acl\Acl` denies access to every privilege upon every resource by every role.

206.4 Registering Roles

CMS's will nearly always require a hierarchy of permissions to determine the authoring capabilities of its users. There may be a 'Guest' group to allow limited access for demonstrations, a 'Staff' group for the majority of *CMS* users who perform most of the day-to-day operations, an 'Editor' group for those responsible for publishing, reviewing, archiving and deleting content, and finally an 'Administrator' group whose tasks may include all of those of the other groups as well as maintenance of sensitive information, user management, back-end configuration data, backup and export. This set of permissions can be represented in a role registry, allowing each group to inherit privileges from 'parent' groups, as well as providing distinct privileges for their unique group only. The permissions may be expressed as follows:

Table 206.1: Access Controls for an Example CMS

Name	Unique Permissions	Inherit Permissions From
Guest	View	N/A
Staff	Edit, Submit, Revise	Guest
Editor	Publish, Archive, Delete	Staff
Administrator	(Granted all access)	N/A

For this example, `Zend\Permissions\Acl\Role\GenericRole` is used, but any object that implements `Zend\Permissions\Acl\Role\RoleInterface` is acceptable. These groups can be added to the role registry as follows:

```
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;

$acl = new Acl();

// Add groups to the Role registry using Zend\Permissions\Acl\Role\GenericRole
// Guest does not inherit access controls
$roleGuest = new Role('guest');
$acl->addRole($roleGuest);

```

```
// Staff inherits from guest
$acl->addRole(new Role('staff'), $roleGuest);

/*
Alternatively, the above could be written:
$acl->addRole(new Role('staff'), 'guest');
*/

// Editor inherits from staff
$acl->addRole(new Role('editor'), 'staff');

// Administrator does not inherit access controls
$acl->addRole(new Role('administrator'));
```

206.5 Defining Access Controls

Now that the *ACL* contains the relevant roles, rules can be established that define how resources may be accessed by roles. You may have noticed that we have not defined any particular resources for this example, which is simplified to illustrate that the rules apply to all resources. `Zend\Permissions\Acl\Acl` provides an implementation whereby rules need only be assigned from general to specific, minimizing the number of rules needed, because resources and roles inherit rules that are defined upon their ancestors.

Note: In general, `Zend\Permissions\Acl\Acl` obeys a given rule if and only if a more specific rule does not apply.

Consequently, we can define a reasonably complex set of rules with a minimum amount of code. To apply the base permissions as defined above:

```
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;

$acl = new Acl();

$roleGuest = new Role('guest');
$acl->addRole($roleGuest);
$acl->addRole(new Role('staff'), $roleGuest);
$acl->addRole(new Role('editor'), 'staff');
$acl->addRole(new Role('administrator'));

// Guest may only view content
$acl->allow($roleGuest, null, 'view');

/*
Alternatively, the above could be written:
$acl->allow('guest', null, 'view');
//*/

// Staff inherits view privilege from guest, but also needs additional
// privileges
$acl->allow('staff', null, array('edit', 'submit', 'revise'));

// Editor inherits view, edit, submit, and revise privileges from
// staff, but also needs additional privileges
$acl->allow('editor', null, array('publish', 'archive', 'delete'));
```

```

// Administrator inherits nothing, but is allowed all privileges
$acl->allow('administrator');
```

The `NULL` values in the above `allow()` calls are used to indicate that the allow rules apply to all resources.

206.6 Querying an ACL

We now have a flexible *ACL* that can be used to determine whether requesters have permission to perform functions throughout the web application. Performing queries is quite simple using the `isAllowed()` method:

```
echo $acl->isAllowed('guest', null, 'view') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', null, 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('staff', null, 'revise') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('editor', null, 'view') ?
     "allowed" : "denied";
// allowed because of inheritance from guest

echo $acl->isAllowed('editor', null, 'update') ?
     "allowed" : "denied";
// denied because no allow rule for 'update'

echo $acl->isAllowed('administrator', null, 'view') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator', null, 'update') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges
```

CHAPTER

SEVEN

REFINING ACCESS CONTROLS

207.1 Precise Access Controls

The basic *ACL* as defined in the *previous section* shows how various privileges may be allowed upon the entire *ACL* (all resources). In practice, however, access controls tend to have exceptions and varying degrees of complexity. `Zend\Permissions\Acl\Acl` allows to you accomplish these refinements in a straightforward and flexible manner.

For the example *CMS*, it has been determined that whilst the 'staff' group covers the needs of the vast majority of users, there is a need for a new 'marketing' group that requires access to the newsletter and latest news in the *CMS*. The group is fairly self-sufficient and will have the ability to publish and archive both newsletters and the latest news.

In addition, it has also been requested that the 'staff' group be allowed to view news stories but not to revise the latest news. Finally, it should be impossible for anyone (administrators included) to archive any 'announcement' news stories since they only have a lifespan of 1-2 days.

First we revise the role registry to reflect these changes. We have determined that the 'marketing' group has the same basic permissions as 'staff', so we define 'marketing' in such a way that it inherits permissions from 'staff':

```
// The new marketing group inherits permissions from staff
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;
use Zend\Permissions\Acl\Resource\GenericResource as Resource;

$acl = new Acl();

$acl->addRole(new Role('marketing'), 'staff');
```

Next, note that the above access controls refer to specific resources (e.g., "newsletter", "latest news", "announcement news"). Now we add these resources:

```
// Create Resources for the rules

// newsletter
$acl->addResource(new Resource('newsletter'));

// news
$acl->addResource(new Resource('news'));

// latest news
$acl->addResource(new Resource('latest'), 'news');

// announcement news
$acl->addResource(new Resource('announcement'), 'news');
```

Then it is simply a matter of defining these more specific rules on the target areas of the *ACL*:

```
// Marketing must be able to publish and archive newsletters and the
// latest news
$acl->allow('marketing',
            array('newsletter', 'latest'),
            array('publish', 'archive'));

// Staff (and marketing, by inheritance), are denied permission to
// revise the latest news
$acl->deny('staff', 'latest', 'revise');

// Everyone (including administrators) are denied permission to
// archive news announcements
$acl->deny(null, 'announcement', 'archive');
```

We can now query the *ACL* with respect to the latest changes:

```
echo $acl->isAllowed('staff', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', 'latest', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('editor', 'announcement', 'archive') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('administrator', 'announcement', 'archive') ?
     "allowed" : "denied";
// denied
```

207.2 Removing Access Controls

To remove one or more access rules from the *ACL*, simply use the available `removeAllow()` or `removeDeny()` methods. As with `allow()` and `deny()`, you may provide a `NULL` value to indicate application to all roles, resources, and/or privileges:

```
// Remove the denial of revising latest news to staff (and marketing,
// by inheritance)
$acl->removeDeny('staff', 'latest', 'revise');

echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// allowed

// Remove the allowance of publishing and archiving newsletters to
// marketing
$acl->removeAllow('marketing',
                  'newsletter',
                  array('publish', 'archive'));

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'archive') ?
     "allowed" : "denied";
// denied
```

Privileges may be modified incrementally as indicated above, but a `NULL` value for the privileges overrides such incremental changes:

```
// Allow marketing all permissions upon the latest news
$acl->allow('marketing', 'latest');

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'anything') ?
     "allowed" : "denied";
// allowed
```

CHAPTER

EIGHT

ADVANCED USAGE

208.1 Storing ACL Data for Persistence

The `Zend\Permissions\Acl` component was designed in such a way that it does not require any particular backend technology such as a database or cache server for storage of the *ACL* data. Its complete *PHP* implementation enables customized administration tools to be built upon `Zend\Permissions\Acl\Acl` with relative ease and flexibility. Many situations require some form of interactive maintenance of the *ACL*, and `Zend\Permissions\Acl\Acl` provides methods for setting up, and querying against, the access controls of an application.

Storage of *ACL* data is therefore left as a task for the developer, since use cases are expected to vary widely for various situations. Because `Zend\Permissions\Acl\Acl` is serializable, *ACL* objects may be serialized with *PHP*'s serialize() function, and the results may be stored anywhere the developer should desire, such as a file, database, or caching mechanism.

208.2 Writing Conditional ACL Rules with Assertions

Sometimes a rule for allowing or denying a role access to a resource should not be absolute but dependent upon various criteria. For example, suppose that certain access should be allowed, but only between the hours of 8:00am and 5:00pm. Another example would be denying access because a request comes from an IP address that has been flagged as a source of abuse. `Zend\Permissions\Acl\Acl` has built-in support for implementing rules based on whatever conditions the developer needs.

`Zend\Permissions\Acl\Acl` provides support for conditional rules with `Zend\Permissions\Acl\Assertion\AssertionInterface`. In order to use the rule assertion interface, a developer writes a class that implements the `assert()` method of the interface:

```
class CleanIPAssertion implements Zend\Permissions\Acl\Assertion\AssertionInterface
{
    public function assert(Zend\Permissions\Acl $acl,
                           Zend\Permissions\Acl\Role\RoleInterface $role = null,
                           Zend\Permissions\Acl\Resource\ResourceInterface $resource = null,
                           $privilege = null)
    {
        return $this->_isCleanIP($_SERVER['REMOTE_ADDR']);
    }

    protected function _isCleanIP($ip)
    {
        // ...
```

```
    }
}
```

Once an assertion class is available, the developer must supply an instance of the assertion class when assigning conditional rules. A rule that is created with an assertion only applies when the assertion method returns `TRUE`.

```
use Zend\Permissions\Acl\Acl;

$acl = new Acl();
$acl->allow(null, null, null, new CleanIPAssertion());
```

The above code creates a conditional allow rule that allows access to all privileges on everything by everyone, except when the requesting IP is "blacklisted." If a request comes in from an IP that is not considered "clean," then the allow rule does not apply. Since the rule applies to all roles, all resources, and all privileges, an "unclean" IP would result in a denial of access. This is a special case, however, and it should be understood that in all other cases (i.e., where a specific role, resource, or privilege is specified for the rule), a failed assertion results in the rule not applying, and other rules would be used to determine whether access is allowed or denied.

The `assert()` method of an assertion object is passed the *ACL*, role, resource, and privilege to which the authorization query (i.e., `isAllowed()`) applies, in order to provide a context for the assertion class to determine its conditions where needed.

CHAPTER

NINE

INTRODUCTION TO ZEND\PERMISSIONS\RBAC

The `Zend\Permissions\Rbac` component provides a lightweight role-based access control implementation based around PHP 5.3's SPL RecursiveIterator and RecursiveIteratorIterator. RBAC differs from access control lists (ACL) by putting the emphasis on roles and their permissions rather than objects (resources).

For the purposes of this documentation:

- an **identity** has one or more roles.
- a **role** requests access to a permission.
- a **permission** is given to a role.

Thus, RBAC has the following model:

- many to many relationship between **identities** and **roles**.
- many to many relationship between **roles** and **permissions**.
- **roles** can have a parent role.

209.1 Roles

The easiest way to create a role is by extending the abstract class `Zend\Permission\Rbac\AbstractRole` or simply using the default class provided in `Zend\Permission\Rbac\Role`. You can instantiate a role and add it to the RBAC container or add a role directly using the RBAC container `addRole()` method.

209.2 Permissions

Each role can have zero or more permissions and can be set directly to the role or by first retrieving the role from the RBAC container. Any parent role will inherit the permissions of their children.

209.3 Dynamic Assertions

In certain situations simply checking a permission key for access may not be enough. For example, assume two users, Foo and Bar, both have *article.edit* permission. What's to stop Bar from editing Foo's articles? The answer is dynamic assertions which allow you to specify extra runtime credentials that must pass for access to be granted.

CHAPTER

METHODS

Zend\Permissions\Rbac\AbstractIterator

- current
- getChildren
- hasChildren
- key
- next
- rewind
- valid

Zend\Permissions\Rbac\AbstractRole

- addChild
- addPermission
- getName
- hasPermission
- setParent
- getParent

Zend\Permissions\Rbac\AssertionInterface

- assert

Zend\Permissions\Rbac\Rbac

- addRole
- getCreateMissingRoles
- getRole
- hasRole
- isGranted
- setCreateMissingRoles

Zend\Permissions\Rbac\Role

- __construct

CHAPTER

ONE

EXAMPLES

The following is a list of common use-case examples for `Zend\Permission\Rbac`.

211.1 Roles

Extending and adding roles via instantiation.

```
<?php
use Zend\Permissions\Rbac\Rbac;
use Zend\Permissions\Rbac\AbstractRole;

class MyRole extends AbstractRole
{
    // .. implementation
}

// Creating roles manually
$foo  = new MyRole('foo');

$rbac = new Rbac();
$rbac->addRole($foo);

var_dump($rbac->hasRole('foo')); // true
```

Adding roles directly to RBAC with the default `Zend\Permission\Rbac\Role`.

```
<?php
use Zend\Permissions\Rbac\Rbac;

$rbac = new Rbac();
$rbac->addRole('foo');

var_dump($rbac->hasRole('foo')); // true
```

Handling roles with children.

```
<?php
use Zend\Permissions\Rbac\Rbac;
use Zend\Permissions\Rbac\Role;

$rbac = new Rbac();
$foo  = new Role('foo');
$bar  = new Role('bar');
```

```

// 1 - Add a role with child role directly with instantiated classes.
$foo->addChild($bar);
$rbac->addRole($foo);

// 2 - Same as one, only via rbac container.
$rbac->addRole('boo', 'baz'); // baz is a parent of boo
$rbac->addRole('baz', array('out', 'of', 'roles')); // create several parents of baz
```

211.2 Permissions

```
<?php
use Zend\Permissions\Rbac\Rbac;
use Zend\Permissions\Rbac\Role;

$rbac = new Rbac();
$foo  = new Role('foo');
$foo->addPermission('bar');

var_dump($foo->hasPermission('bar')); // true

$rbac->addRole($foo);
$rbac->isGranted('foo', 'bar'); // true
$rbac->isGranted('foo', 'baz'); // false

$rbac->getRole('foo')->addPermission('baz');
$rbac->isGranted('foo', 'baz'); // true
```

211.3 Dynamic Assertions

Checking permission using `isGranted()` with a class implementing `Zend\Permissions\Rbac\AssertionInterface`.

```
<?php
use Zend\Permissions\Rbac\AssertionInterface;
use Zend\Permissions\Rbac\Rbac;

class AssertUserIdMatches implements AssertionInterface
{
    protected $userId;
    protected $article;

    public function __construct($userId)
    {
        $this->userId = $userId;
    }

    public function setArticle($article)
    {
        $this->article = $article;
    }

    public function assert(Rbac $rbac)
    {
        if (!$this->article) {
```

```
            return false;
        }
        return $this->userId == $article->getUserId();
    }
}

// User is assigned the foo role with id 5
// News article belongs to userId 5
// Jazz article belongs to userId 6

$rbac = new Rbac();
$user = $mySessionObject->getUser();
$news = $articleService->getArticle(5);
$jazz = $articleService->getArticle(6);

$rbac->addRole($user->getRole());
$rbac->getRole($user->getRole())->addPermission('edit.article');

$assertion = new AssertUserIdMatches($user->getId());
$assertion->setArticle($news);

// true always - bad!
if ($rbac->isGranted($user->getRole(), 'edit.article')) {
    // hacks another user's article
}

// true for user id 5, because he belongs to write group and user id matches
if ($rbac->isGranted($user->getRole(), 'edit.article', $assertion)) {
    // edits his own article
}

$assertion->setArticle($jazz);

// false for user id 5
if ($rbac->isGranted($user->getRole(), 'edit.article', $assertion)) {
    // can not edit another user's article
}
```

Performing the same as above with a Closure.

```
<?php
// assume same variables from previous example

$assertion = function($rbac) use ($user, $news) {
    return $user->getId() == $news->getUserId();
};

// true
if ($rbac->isGranted($user->getRole(), 'edit.article', $assertion)) {
    // edits his own article
}
```

CHAPTER

TWO

PROGRESS BARS

212.1 Introduction

`Zend\ProgressBar` is a component to create and update progress bars in different environments. It consists of a single backend, which outputs the progress through one of the multiple adapters. On every update, it takes an absolute value and optionally a status message, and then calls the adapter with some precalculated values like percentage and estimated time left.

212.2 Basic Usage

`Zend\ProgressBar` is quite easy in its usage. You simply create a new instance of `Zend\Progressbar`, defining a min- and a max-value, and choose an adapter to output the data. If you want to process a file, you would do something like:

```
$progressBar = new Zend\ProgressBar\ProgressBar($adapter, 0, $fileSize);

while (!feof($fp)) {
    // Do something

    $progressBar->update($currentByteCount);
}

$progressBar->finish();
```

In the first step, an instance of `Zend\ProgressBar` is created, with a specific adapter, a min-value of 0 and a max-value of the total filesize. Then a file is processed and in every loop the progressbar is updated with the current byte count. At the end of the loop, the progressbar status is set to finished.

You can also call the `update()` method of `Zend\ProgressBar` without arguments, which just recalculates ETA and notifies the adapter. This is useful when there is no data update but you want the progressbar to be updated.

212.3 Persistent Progress

If you want the progressbar to be persistent over multiple requests, you can give the name of a session namespace as fourth argument to the constructor. In that case, the progressbar will not notify the adapter within the constructor, but only when you call `update()` or `finish()`. Also the current value, the status text and the start time for ETA calculation will be fetched in the next request run again.

212.4 Standard Adapters

`Zend\ProgressBar` comes with the following three adapters:

- *Zend\Progressbar\Adapter\Console*
- *Zend\Progressbar\Adapter\JsPush*
- *Zend\ProgressBar\Adapter\JsPull*

212.4.1 Console Adapter

`Zend\ProgressBar\Adapter\Console` is a text-based adapter for terminals. It can automatically detect terminal widths but supports custom widths as well. You can define which elements are displayed with the progressbar and as well customize the order of them. You can also define the style of the progressbar itself.

Note: Automatic console width recognition

shell_exec is required for this feature to work on *nix based systems. On windows, there is always a fixed terminal width of 80 character, so no recognition is required there.

You can set the adapter options either via the *set** methods or give an array or a `Zend\Config\Config` instance with options as first parameter to the constructor. The available options are:

- *outputStream*: A different output-stream, if you don't want to stream to STDOUT. Can be any other stream like *php://stderr* or a path to a file.
- *width*: Either an integer or the `AUTO` constant of `Zend\Console\ProgressBar`.
- *elements*: Either `NULL` for default or an array with at least one of the following constants of `Zend\Console\ProgressBar` as value:
  - `ELEMENT_PERCENT`: The current value in percent.
  - `ELEMENT_BAR`: The visual bar which display the percentage.
  - `ELEMENT_ETA`: The automatic calculated ETA. This element is firstly displayed after five seconds, because in this time, it is not able to calculate accurate results.
  - `ELEMENT_TEXT`: An optional status message about the current process.
- *textWidth*: Width in characters of the `ELEMENT_TEXT` element. Default is 20.
- *charset*: Charset of the `ELEMENT_TEXT` element. Default is utf-8.
- *barLeftChar*: A string which is used left-hand of the indicator in the progressbar.
- *barRightChar*: A string which is used right-hand of the indicator in the progressbar.
- *barIndicatorChar*: A string which is used for the indicator in the progressbar. This one can be empty.

212.4.2 JsPush Adapter

`Zend\ProgressBar\Adapter\JsPush` is an adapter which let's you update a progressbar in a browser via Javascript Push. This means that no second connection is required to gather the status about a running process, but that the process itself sends its status directly to the browser.

You can set the adapter options either via the *set** methods or give an array or a `Zend\Config\Config` instance with options as first parameter to the constructor. The available options are:

- *updateMethodName*: The JavaScript method which should be called on every update. Default value is `Zend\ProgressBar\Update`.
- *finishMethodName*: The JavaScript method which should be called after finish status was set. Default value is `NULL`, which means nothing is done.

The usage of this adapter is quite simple. First you create a progressbar in your browser, either with JavaScript or previously created with plain *HTML*. Then you define the update method and optionally the finish method in JavaScript, both taking a json object as single argument. Then you call a webpage with the long-running process in a hidden *iframe* or *object* tag. While the process is running, the adapter will call the update method on every update with a json object, containing the following parameters:

- *current*: The current absolute value
- *max*: The max absolute value
- *percent*: The calculated percentage
- *timeTaken*: The time how long the process ran yet
- *timeRemaining*: The expected time for the process to finish
- *text*: The optional status message, if given

Basic example for the client-side stuff

This example illustrates a basic setup of *HTML*, *CSS* and JavaScript for the JsPush adapter

```
<div id="zend-progressbar-container">
    <div id="zend-progressbar-done"></div>
</div>

<iframe src="long-running-process.php" id="long-running-process"></iframe>
```

```
#long-running-process {
    position: absolute;
    left: -100px;
    top: -100px;

    width: 1px;
    height: 1px;
}

#zend-progressbar-container {
    width: 100px;
    height: 30px;

    border: 1px solid #000000;
    background-color: #ffffff;
}

#zend-progressbar-done {
    width: 0;
    height: 30px;

    background-color: #000000;
}
```

```
function Zend\ProgressBar\Update(data)
{
    document.getElementById('zend-progressbar-done').style.width =
         data.percent + '%';
}
```

This will create a simple container with a black border and a block which indicates the current process. You should not hide the *iframe* or *object* by *display: none;*, as some browsers like Safari 2 will not load the actual content then.

Instead of creating your custom progressbar, you may want to use one of the available JavaScript libraries like Dojo, jQuery etc. For example, there are:

- Dojo: http://dojotoolkit.org/reference-guide/dijit/ProgressBar.html
- jQuery: http://t.wits.sg/2008/06/20/jquery-progress-bar-11/
- MooTools: http://davidwalsh.name/dw-content/progress-bar.php
- Prototype: http://livepipe.net/control/progressbar

Note: Interval of updates

You should take care of not sending too many updates, as every update has a min-size of 1kb. This is a requirement for the Safari browser to actually render and execute the function call. Internet Explorer has a similar limitation of 256 bytes.

212.4.3 JsPull Adapter

`Zend\ProgressBar\Adapter\JsPull` is the opposite of jsPush, as it requires to pull for new updates, instead of pushing updates out to the browsers. Generally you should use the adapter with the persistence option of the `Zend\ProgressBar`. On notify, the adapter sends a *JSON* string to the browser, which looks exactly like the *JSON* string which is send by the jsPush adapter. The only difference is, that it contains an additional parameter, `finished`, which is either `FALSE` when `update()` is called or `TRUE`, when `finish()` is called.

You can set the adapter options either via the `set*()` methods or give an array or a `Zend\Config\Config` instance with options as first parameter to the constructor. The available options are:

- `exitAfterSend`: Exits the current request after the data were send to the browser. Default is `TRUE`.

CHAPTER

THREE

FILE UPLOAD HANDLERS

213.1 Introduction

`Zend\ProgressBar\Upload` provides handlers that can give you the actual state of a file upload in progress. To use this feature you need to choose one of the upload progress handlers (APC, uploadprogress, or Session) and ensure that your server setup has the appropriate extension or feature enabled. All of the progress handlers use the same interface.

When uploading a file with a form POST, you must also include the progress identifier in a hidden input. The *File Upload Progress View Helpers* provide a convenient way to add the hidden input based on your handler type.

213.2 Methods of Reporting Progress

There are two methods for reporting the current upload progress status. By either using a ProgressBar Adapter, or by using the returned status array manually.

213.2.1 Using a ProgressBar Adapter

A `Zend\ProgressBar` adapter can be used to display upload progress to your users.

```
$adapter  = new \Zend\ProgressBar\Adapter\JsPush();
$progress = new \Zend\ProgressBar\Upload\SessionProgress();

$filter   = new \Zend\I18n\Filter\Alnum(false, 'en_US');
$id       = $filter->filter($_GET['id']);

$status   = null;
while (empty($status['done'])) {
    $status = $progress->getProgress($id);
}
```

Each time the `getProgress()` method is called, the ProgressBar adapter will be updated.

213.2.2 Using the Status Array

You can also work manually with `getProgress()` without using a `Zend\ProgressBar` adapter.

The `getProgress()` will return you an array with several keys. They will sometimes differ based on the specific Upload handler used, but the following keys are always standard:

- `total`: The total file size of the uploaded file(s) in bytes as integer.
- `current`: The current uploaded file size in bytes as integer.
- `rate`: The average upload speed in bytes per second as integer.
- `done`: Returns `TRUE` when the upload is finished and `FALSE` otherwise.
- `message`: A status message. Either the progress as text in the form "10kB / 200kB", or a helpful error message in the case of a problem. Problems such as: no upload in progress, failure while retrieving the data for the progress, or that the upload has been canceled.

All other returned keys are provided directly from the specific handler.

An example of using the status array manually:

```
// In a Controller...

public function sessionProgressAction()
{
    $id = $this->params()->fromQuery('id', null);
    $progress = new \Zend\ProgressBar\Upload\SessionProgress();
    return new \Zend\View\Model\JsonModel($progress->getProgress($id));
}

// Returns JSON
//{
//    "total"    : 204800,
//    "current"  : 10240,
//    "rate"     : 1024,
//    "message"  : "10kB / 200kB",
//    "done"     : false
//}
```

213.3 Standard Handlers

`Zend\ProgressBar\Upload` comes with the following three upload handlers:

- *Zend\Progressbar\Upload\ApcProgress*
- *Zend\ProgressBar\Upload\SessionProgress*
- *Zend\Progressbar\Upload\UploadProgress*

213.3.1 APC Progress Handler

The `Zend\ProgressBar\Upload\ApcProgress` handler uses the APC extension for tracking upload progress.

Note: The APC extension is required.

This handler is best used with the *FormFileApcProgress* view helper, to provide a hidden element with the upload progress identifier.

213.3.2 Session Progress Handler

The `Zend\ProgressBar\Upload\SessionProgress` handler uses the PHP 5.4 Session Progress feature for tracking upload progress.

Note: PHP 5.4 is required.

This handler is best used with the *FormFileSessionProgress* view helper, to provide a hidden element with the upload progress identifier.

213.3.3 Upload Progress Handler

The `Zend\ProgressBar\Upload\UploadProgress` handler uses the PECL Uploadprogress extension for tracking upload progress.

Note: The PECL Uploadprogress extension is required.

This handler is best used with the *FormFileUploadProgress* view helper, to provide a hidden element with the upload progress identifier.

CHAPTER

FOUR

INTRODUCTION TO ZEND\SERIALIZER

The `Zend\Serializer` component provides an adapter based interface to simply generate storable representation of PHP types by different facilities, and recover.

For more information what a serializer is read the wikipedia page of Serialization.

214.1 Quick Start

Serializing adapters can either be created from the provided `Zend\Serializer\Serializer::factory` method, or by simply instantiating one of the `Zend\Serializer\Adapter\*` classes.

```
use Zend\Serializer\Serializer;

// Via factory:
$serializer = Zend\Serializer\Serializer::factory('PhpSerialize');

// Alternately:
$serializer = new Zend\Serializer\Adapter\PhpSerialize();

// Now $serializer is an instance of Zend\Serializer\Adapter\AdapterInterface,
// specifically Zend\Serializer\Adapter\PhpSerialize

try {
    $serialized = $serializer->serialize($data);
    // now $serialized is a string

    $unserialized = $serializer->unserialize($serialized);
    // now $data == $unserialized
} catch (Zend\Serializer\Exception\ExceptionInterface $e) {
    echo $e;
}
```

The method `serialize()` generates a storable string. To regenerate this serialized data you can simply call the method `unserialize()`.

Any time an error is encountered serializing or unserializing, `Zend\Serializer` will throw a `Zend\Serializer\Exception\ExceptionInterface`.

Because of an application often uses internally only one serializer it is possible to define and use a default serializer. That serializer will be used by default by other components like `Zend\Cache\Storage\Plugin\Serializer`.

To use the default serializer you can simply use the static serialization methods of the basic `Zend\Serializer\Serializer`:

```
use Zend\Serializer\Serializer;

try {
    $serialized = Serializer::serialize($data);
    // now $serialized is a string

    $unserialized = Serializer::unserialize($serialized);
    // now $data == $unserialized
} catch (Zend\Serializer\Exception\ExceptionInterface $e) {
    echo $e;
}
```

214.2 Basic configuration Options

To configure a serializer adapter, you can optionally use an instance of `Zend\Serializer\Adapter\AdapterOptions`, an instance of one of the adapter specific options class, an `array` or an instance of `Traversable`. The adapter will convert it into the adapter specific options class instance (if present) or into the basic `Zend\Serializer\Adapter\AdapterOptions` class instance.

Options can be passed as second argument to the provided `Zend\Serializer\Serializer::factory` method, using the method `setOptions` or set as constructor argument.

214.3 Available Methods

Each serializer implements the interface `Zend\Serializer\Adapter\AdapterInterface`.

This interface defines the following methods:

serialize(*mixed $value*)
Generates a storable representation of a value.

Return type string

unserialize(*string $value*)
Creates a PHP value from a stored representation.

Return type mixed

The basic class `Zend\Serializer\Serializer` will be used to instantiate the adapters, to configure the factory and to handle static serializing.

It defines the following **static** methods:

factory(*string|Zend\Serializer\Adapter\AdapterInterface $adapterName*, *Zend\Serializer\Adapter\AdapterOptions|array|Traversabl*
$adapterOptions = null)
Create a serializer adapter instance.

Return type Zend\Serializer\Adapter\AdapterInterface

setAdapterPluginManager(*Zend\Serializer\AdapterPluginManager $adapters*)
Change the adapter plugin manager.

Return type void

getAdapterPluginManager()
Get the adapter plugin manager.

Return type Zend\Serializer\AdapterPluginManager

resetAdapterPluginManager()
Resets the internal adapter plugin manager.

Return type void

setDefaultAdapter(*string|Zend\Serializer\Adapter\AdapterInterface $adapter, Zend\Serializer\Adapter\AdapterOptions|array|Traversable|null $adapterOptions = null*)
Change the default adapter.

Return type void

getDefaultAdapter()
Get the default adapter.

Return type Zend\Serializer\Adapter\AdapterInterface

serialize(*mixed $value, string|Zend\Serializer\Adapter\AdapterInterface|null $adapter = null, Zend\Serializer\Adapter\AdapterOptions|array|Traversable|null $adapterOptions = null*)
Generates a storable representation of a value using the default adapter. Optionally different adapter could be provided as second argument.

Return type string

unserialize(*string $value, string|Zend\Serializer\Adapter\AdapterInterface|null $adapter = null, Zend\Serializer\Adapter\AdapterOptions|array|Traversable|null $adapterOptions = null*)
Creates a PHP value from a stored representation using the default adapter. Optionally different adapter could be provided as second argument.

Return type mixed

CHAPTER

FIVE

ZEND\SERIALIZER\ADAPTER

`Zend\Serializer` adapters create a bridge for different methods of serializing with very little effort.

Every adapter has different pros and cons. In some cases, not every PHP datatype (e.g., objects) can be converted to a string representation. In most such cases, the type will be converted to a similar type that is serializable.

As an example, PHP objects will often be cast to arrays. If this fails, a `Zend\Serializer\Exception\ExceptionInterface` will be thrown.

215.1 The PhpSerialize Adapter

The `Zend\Serializer\Adapter\PhpSerialize` adapter uses the built-in `un/serialize` PHP functions, and is a good default adapter choice.

There are no configurable options for this adapter.

215.2 The IgBinary Adapter

Igbinary is Open Source Software released by Sulake Dynamoid Oy and since 2011-03-14 moved to PECL maintained by Pierre Joye. It's a drop-in replacement for the standard PHP serializer. Instead of time and space consuming textual representation, igbinary stores PHP data structures in a compact binary form. Savings are significant when using memcached or similar memory based storages for serialized data.

You need the igbinary PHP extension installed on your system in order to use this adapter.

There are no configurable options for this adapter.

215.3 The Wddx Adapter

WDDX (Web Distributed Data eXchange) is a programming-language-, platform-, and transport-neutral data interchange mechanism for passing data between different environments and different computers.

The adapter simply uses the wddx_*() PHP functions. Please read the PHP manual to determine how you may enable them in your PHP installation.

Additionally, the SimpleXML PHP extension is used to check if a returned `NULL` value from `wddx_unserialize()` is based on a serialized `NULL` or on invalid data.

Available options include:

Table 215.1: ZendSerializerAdapterWddx Options

Option	Data Type	Default Value	Description
comment	`string`		An optional comment that appears in the packet header.

215.4 The Json Adapter

The JSON adapter provides a bridge to the `Zend\Json` component. Please read the *ZendJson documentation* for further information.

Available options include:

Table 215.2: ZendSerializerAdapterJson Options

Option	Data Type	Default Value
cycle_check	`boolean`	`false`
object_decode_type	`Zend\Json\Json::TYPE_*`	`Zend\Json\Json::TYPE_ARRAY`
enable_json_expr_finder	`boolean`	`false`

215.5 The PythonPickle Adapter

This adapter converts PHP types to a Python Pickle string representation. With it, you can read the serialized data with Python and read Pickled data of Python with PHP.

Available options include:

Table 215.3: ZendSerializerAdapterPythonPickle Options

Option	Data Type	Default Value	Description
protocol	`integer` (0|1|2|3)	0	The Pickle protocol version used on serialize

Table 215.4: Datatype merging (PHP to Python Pickle)

PHP Type	Python Pickle Type
`NULL`	`None`
`boolean`	`boolean`
`integer`	`integer`
`float`	`float`
`string`	`string`
`array` list	`list`
`array` map	`dictionary`
`object`	`dictionary`

Table 215.5: Datatype merging (Python Pickle to PHP)

Python Pickle Type	PHP Type
`None`	`NULL`
"boolean	`boolean`
"integer	`integer`
"long	`integer` or `float` or `string` or `Zend\Serializer\Exception\ExceptionInterface`
"float	`float`
"string	`string`
"bytes	`string`
`unicode string`	`string` UTF-8
`list`	`array` list
`tuple`	`array` list
`dictionary`	`array` map
All other types	`Zend\Serializer\Exception\ExceptionInterface`

215.6 The PhpCode Adapter

The `Zend\Serializer\Adapter\PhpCode` adapter generates a parsable PHP code representation using var_export(). On restoring, the data will be executed using eval.

There are no configuration options for this adapter.

Warning: Unserializing objects
Objects will be serialized using the __set_state magic method. If the class doesn't implement this method, a fatal error will occur during execution.

Warning: Uses eval()
The `PhpCode` adapter utilizes `eval()` to unserialize. This introduces both a performance and potential security issue as a new process will be executed. Typically, you should use the `PhpSerialize` adapter unless you require human-readability of the serialized data.

CHAPTER

SIX

INTRODUCTION TO ZEND\SERVER

The `Zend\Server` family of classes provides functionality for the various server classes, including `Zend\XmlRpc\Server` and `Zend\Json\Server`. `Zend\Server\Server` provides an interface that mimics *PHP* 5's `SoapServer` class; all server classes should implement this interface in order to provide a standard server *API*.

The `Zend\Server\Reflection` tree provides a standard mechanism for performing function and class introspection for use as callbacks with the server classes, and provides data suitable for use with `Zend\Server\Server`'s `getFunctions()` and `loadFunctions()` methods.

CHAPTER

SEVEN

ZEND\SERVER\REFLECTION

217.1 Introduction

`Zend\Server\Reflection` provides a standard mechanism for performing function and class introspection for use with server classes. It is based on *PHP* 5's Reflection *API*, augmenting it with methods for retrieving parameter and return value types and descriptions, a full list of function and method prototypes (i.e., all possible valid calling combinations), and function or method descriptions.

Typically, this functionality will only be used by developers of server classes for the framework.

217.2 Usage

Basic usage is simple:

```
$class    = Zend\Server\Reflection::reflectClass('My\Class');
$function = Zend\Server\Reflection::reflectFunction('my_function');

// Get prototypes
$prototypes = $function->getPrototypes();

// Loop through each prototype for the function
foreach ($prototypes as $prototype) {

    // Get prototype return type
    echo "Return type: ", $prototype->getReturnType(), "\n";

    // Get prototype parameters
    $parameters = $prototype->getParameters();

    echo "Parameters: \n";
    foreach ($parameters as $parameter) {
        // Get parameter type
        echo "    ", $parameter->getType(), "\n";
    }
}

// Get namespace for a class, function, or method.
// Namespaces may be set at instantiation time (second argument), or using
// setNamespace()
$class->getNamespace();
```

`reflectFunction()` returns a `Zend\Server\Reflection\Function` object; `reflectClass()` returns a `Zend\Server\Reflection\Class` object. Please refer to the *API* documentation to see what methods are available to each.

CHAPTER

EIGHT

ZEND\SERVICEMANAGER

The Service Locator design pattern is implemented by the `Zend\ServiceManager` component. The Service Locator is a service/object locator, tasked with retrieving other objects. Following is the `Zend\ServiceManager\ServiceLocatorInterface` API:

```
namespace Zend\ServiceManager;

interface ServiceLocatorInterface
{
    public function get($name);
    public function has($name);
}
```

- `has($name)`, tests whether the `ServiceManager` has a named service;
- `get($name)`, retrieves a service by the given name.

A `Zend\ServiceManager\ServiceManager` is an implementation of the `ServiceLocatorInterface`. In addition to the above described methods, the `ServiceManager` provides additional API:

- **Service registration**. `ServiceManager::setService` allows you to register an object as a service:

  ```
  $serviceManager->setService('my-foo', new stdClass());
  $serviceManager->setService('my-settings', array('password' => 'super-secret'));

  var_dump($serviceManager->get('my-foo')); // an instance of stdClass
  var_dump($serviceManager->get('my-settings')); // array('password' => 'super-secret')
  ```

- **Lazy-loaded service objects**. `ServiceManager::setInvokableClass` allows you to tell the `ServiceManager` what class to instantiate when a particular service is requested:

  ```
  $serviceManager->setInvokableClass('foo-service-name', 'Fully\Qualified\Classname');

  var_dump($serviceManager->get('foo-service-name')); // an instance of Fully\Qualified\Classname
  ```

- **Service factories**. Instead of an actual object instance or a class name, you can tell the `ServiceManager` to invoke a provided factory in order to get the object instance. Factories may be either a PHP callback, an object implementing `Zend\ServiceManager\FactoryInterface`, or the name of a class implementing that interface:

  ```
  use Zend\ServiceManager\FactoryInterface;
  use Zend\ServiceManager\ServiceLocatorInterface;

  class MyFactory implements FactoryInterface
  {
      public function createService()
  ```

```
    {
        return new \stdClass();
    }
}

// registering a factory instance
$serviceManager->setFactory('foo-service-name', new MyFactory());

// registering a factory by factory class name
$serviceManager->setFactory('bar-service-name', 'MyFactory');

// registering a callback as a factory
$serviceManager->setFactory('baz-service-name', function () { return new \stdClass(); });

var_dump($serviceManager->get('foo-service-name')); // stdClass(1)
var_dump($serviceManager->get('bar-service-name')); // stdClass(2)
var_dump($serviceManager->get('baz-service-name')); // stdClass(3)
```

- **Service aliasing**. With `ServiceManager::setAlias` you can create aliases of any registered service, factory or invokable, or even other aliases:

```
$foo      = new \stdClass();
$foo->bar = 'baz!';

$serviceManager->setService('my-foo', $foo);
$serviceManager->setAlias('my-bar', 'my-foo');
$serviceManager->setAlias('my-baz', 'my-bar');

var_dump($serviceManager->get('my-foo')->bar); // baz!
var_dump($serviceManager->get('my-bar')->bar); // baz!
var_dump($serviceManager->get('my-baz')->bar); // baz!
```

- **Abstract factories**. An abstract factory can be considered as a "fallback" factory. If the service manager was not able to find a service for the requested name, it will check the registered abstract factories:

```
use Zend\ServiceManager\ServiceLocatorInterface;
use Zend\ServiceManager\AbstractFactoryInterface;

class MyAbstractFactory implements AbstractFactoryInterface
{
    public function canCreateServiceWithName(ServiceLocatorInterface $serviceLocator, $name, $re
    {
        // this abstract factory only knows about 'foo' and 'bar'
        return $requestedName === 'foo' || $requestedName === 'bar';
    }

    public function createServiceWithName(ServiceLocatorInterface $serviceLocator, $name, $reque
    {
        $service = new \stdClass();

        $service->name = $requestedName;

        return $service;
    }
}

$serviceManager->addAbstractFactory('MyAbstractFactory');

var_dump($serviceManager->get('foo')->name); // foo
```

```
var_dump($serviceManager->get('bar')->name); // bar
var_dump($serviceManager->get('bar')->name); // exception! Zend\ServiceManager\Exception\Service
```

- **Initializers**. You may want certain injection points to be always called. As an example, any object you load via the service manager that implements `Zend\EventManager\EventManagerAwareInterface` should likely receive an `EventManager` instance. **Initializers** are PHP callbacks or classes implementing `Zend\ServiceManager\InitializerInterface`. They receive the new instance, and can then manipulate it:

```
use Zend\ServiceManager\ServiceLocatorInterface;
use Zend\ServiceManager\InitializerInterface;

class MyInitializer implements InitializerInterface
{
    public function initialize($instance, ServiceLocatorInterface $serviceLocator)
    {
        if ($instance instanceof \stdClass) {
            $instance->initialized = 'initialized!';
        }
    }
}

$serviceManager->setInvokableClass('my-service', 'stdClass');

var_dump($serviceManager->get('my-service')->initialized); // initialized!
```

In addition to the above, the `ServiceManager` also provides optional ties to `Zend\Di`, allowing `Di` to act as an initializer or an abstract factory for the service manager.

CHAPTER

NINE

ZEND\SERVICEMANAGER QUICK START

By default, Zend Framework utilizes `Zend\ServiceManager` within the MVC layer and in various other components. As such, in most cases you'll be providing services, invokable classes, aliases, and factories either via configuration or via your module classes.

By default, the module manager listener `Zend\ModuleManager\Listener\ServiceListener` will do the following:

- For modules implementing `Zend\ModuleManager\Feature\ServiceProviderInterface`, or the `getServiceConfig()` method, it will call that method and merge the retrieved configuration.
- After all modules have been processed, it will grab the configuration from the registered `Zend\ModuleManager\Listener\ConfigListener`, and merge any configuration under the `service_manager` key.
- Finally, it will use the merged configuration to configure the `ServiceManager` instance.

In most cases, you won't interact with the `ServiceManager`, other than to providing services to it; your application will typically rely on the configuration of the `ServiceManager` to ensure that services are configured correctly with their dependencies. When creating factories, however, you may want to interact with the `ServiceManager` to retrieve other services to inject as dependencies. Additionally, there are some cases where you may want to receive the `ServiceManager` to lazy-retrieve dependencies; as such, you may want to implement `ServiceLocatorAwareInterface` and know more details about the API of the `ServiceManager`.

219.1 Using Configuration

Configuration requires a `service_manager` key at the top level of your configuration, with any of the following sub-keys:

- **abstract_factories**, which should be an array of abstract factory class names.
- **aliases**, which should be an associative array of alias name/target name pairs (where the target name may also be an alias).
- **factories**, an array of service name/factory class name pairs. The factories should be either classes implementing `Zend\ServiceManager\FactoryInterface` or invokable classes. If you are using PHP configuration files, you may provide any PHP callable as the factory.
- **invokables**, an array of service name/class name pairs. The class name should be class that may be directly instantiated without any constructor arguments.
- **services**, an array of service name/object pairs. Clearly, this will only work with PHP configuration.

- **shared**, an array of service name/boolean pairs, indicating whether or not a service should be shared. By default, the `ServiceManager` assumes all services are shared, but you may specify a boolean false value here to indicate a new instance should be returned.

219.2 Modules as Service Providers

Modules may act as service configuration providers. To do so, the Module class must either implement `Zend\ModuleManager\Feature\ServiceProviderInterface` or simply the method `getServiceConfig()` (which is also defined in the interface). This method must return one of the following:

- An array (or `Traversable` object) of configuration compatible with `Zend\ServiceManager\Config`. (Basically, it should have the keys for configuration as discussed in *the previous section*.
- A string providing the name of a class implementing `Zend\ServiceManager\ConfigInterface`.
- An instance of either `Zend\ServiceManager\Config`, or an object implementing `Zend\ServiceManager\ConfigInterface`.

As noted previously, this configuration will be merged with the configuration returned from other modules as well as configuration files, prior to being passed to the `ServiceManager`; this allows overriding configuration from modules easily.

219.3 Examples

219.3.1 Sample Configuration

The following is valid configuration for any configuration being merged in your application, and demonstrates each of the possible configuration keys. Configuration is merged in the following order:

- Configuration returned from Module classes via the `getServiceConfig()` method, in the order in which modules are processed.
- Module configuration under the `service_manager` key, in the order in which modules are processed.
- Application configuration under the `config/autoload/` directory, in the order in which they are processed.

As such, you have a variety of ways to override service manager configuration settings.

```
<?php
// a module configuration, "module/SomeModule/config/module.config.php"
return array(
    'service_manager' => array(
        'abstract_factories' => array(
            // Valid values include names of classes implementing
            // AbstractFactoryInterface, instances of classes implementing
            // AbstractFactoryInterface, or any PHP callbacks
            'SomeModule\Service\FallbackFactory',
        ),
        'aliases' => array(
            // Aliasing a FQCN to a service name
            'SomeModule\Model\User' => 'User',
            // Aliasing a name to a known service name
            'AdminUser' => 'User',
            // Aliasing to an alias
            'SuperUser' => 'AdminUser',
        ),
```

```
        'factories' => array(
            // Keys are the service names.
            // Valid values include names of classes implementing
            // FactoryInterface, instances of classes implementing
            // FactoryInterface, or any PHP callbacks
            'User'     => 'SomeModule\Service\UserFactory',
            'UserForm' => function ($serviceManager) {
                $form = new SomeModule\Form\User();

                // Retrieve a dependency from the service manager and inject it!
                $form->setInputFilter($serviceManager->get('UserInputFilter'));
                return $form;
            },
        ),
        'invokables' => array(
            // Keys are the service names
            // Values are valid class names to instantiate.
            'UserInputFiler' => 'SomeModule\InputFilter\User',
        ),
        'services' => array(
            // Keys are the service names
            // Values are objects
            'Auth' => new SomeModule\Authentication\AuthenticationService(),
        ),
        'shared' => array(
            // Usually, you'll only indicate services that should **NOT** be
            // shared -- i.e., ones where you want a different instance
            // every time.
            'UserForm' => false,
        ),
    ),
);
```

Note: Configuration and PHP

Typically, you should not have your configuration files create new instances of objects or even closures for factories; at the time of configuration, not all autoloading may be in place, and if another configuration overwrites this one later, you're now spending CPU and memory performing work that is ultimately lost.

For instances that require factories, write a factory. If you'd like to inject specific, configured instances, use the Module class to do so, or a listener.

Additionally you will lose the ability to use the caching feature of the configuration files when you use closures within them. This is a limitation of PHP which can't (de)serialize closures.

Note: Service names good practices

When defining a new service, it is usually a good idea to use the fully qualified class name of the produced instance or of one of the interfaces it implements as service name.

Using a FQCN as service name makes collisions with other services very hard if the class is part of your own code base, and also aids the developer that consumes that service to have a clear overview on what the API of the service looks like.

If the service is not an instance of a class/interface of your own code base, you should always consider using a prefix for it, so that collisions with other services are avoided.

219.3.2 Module Returning an Array

The following demonstrates returning an array of configuration from a module class. It can be substantively the same as the array configuration from the previous example.

```
namespace SomeModule;

// you may eventually want to implement Zend\ModuleManager\Feature\ServiceProviderInterface
class Module
{
    public function getServiceConfig()
    {
        return array(
            'abstract_factories' => array(),
            'aliases' => array(),
            'factories' => array(),
            'invokables' => array(),
            'services' => array(),
            'shared' => array(),
        );
    }
}
```

Creating a ServiceLocator-aware class

By default, the Zend Framework MVC registers an initializer that will inject the `ServiceManager` instance, which is an implementation of `Zend\ServiceManager\ServiceLocatorInterface`, into any class implementing `Zend\ServiceManager\ServiceLocatorAwareInterface`.

A simple implementation looks like following:

```
namespace SomeModule\Controller;

use Zend\ServiceManager\ServiceLocatorAwareInterface;
use Zend\ServiceManager\ServiceLocatorInterface;
use Zend\Stdlib\DispatchableInterface as Dispatchable;
use Zend\Stdlib\RequestInterface as Request;
use Zend\Stdlib\ResponseInterface as Response;

class BareController implements
    Dispatchable,
    ServiceLocatorAwareInterface
{
    protected $services;

    public function setServiceLocator(ServiceLocatorInterface $serviceLocator)
    {
        $this->services = $serviceLocator;
    }

    public function getServiceLocator()
    {
        return $this->services;
    }

    public function dispatch(Request $request, Response $response = null)
    {
```

```
        // ...

        // Retrieve something from the service manager
        $router = $this->getServiceLocator()->get('Router');

        // ...
    }
}
```

CHAPTER

SESSION CONFIG

Zend Framework comes with a standard set of config classes which are ready for you to use. Config handles setting various configuration such as where a cookie lives, lifetime, including several bits to configure ext/session when using `Zend\Session\Config\SessionConfig`.

CHAPTER

ONE

STANDARD CONFIG

`Zend\Session\Config\StandardConfig` provides you a basic interface for implementing sessions when *not* leveraging ext/session. This is utilized more for specialized cases such as when you might have session management done by another system.

221.1 Basic Configuration Options

The following configuration options are defined by `Zend\Session\Config\StandardConfig`.

Option	Data Type	Description
cache_expire	`integer`	Specifies time-to-live for cached session pages in minutes.
cookie_domain	`string`	Specifies the domain to set in the session cookie.
cookie_httponly	`boolean`	Marks the cookie as accessible only through the HTTP protocol.
cookie_lifetime	`integer`	Specifies the lifetime of the cookie in seconds which is sent to the browser.
cookie_path	`string`	Specifies path to set in the session cookie.
cookie_secure	`boolean`	Specifies whether cookies should only be sent over secure connections.
entropy_length	`integer`	Specifies the number of bytes which will be read from the file specified in entropy_file.
entropy_file	`string`	Defines a path to an external resource (file) which will be used as an additional entropy.
gc_maxlifetime	`integer`	Specifies the number of seconds after which data will be seen as 'garbage'.
gc_divisor	`integer`	Defines the probability that the gc process is started on every session initialization.
gc_probability	`integer`	Defines the probability that the gc process is started on every session initialization.
hash_bits_per_character	`integer`	Defines how many bits are stored in each character when converting the binary hash data.
name	`string`	Specifies the name of the session which is used as cookie name.
remem-ber_me_seconds	`integer`	Specifies how long to remember the session before clearing data.
save_path	`string`	Defines the argument which is passed to the save handler.
use_cookies	`boolean`	Specifies whether the module will use cookies to store the session id.

CHAPTER

TWO

BASIC USAGE

A basic example is one like the following:

```
use Zend\Session\Config\StandardConfig;
use Zend\Session\SessionManager;

$config = new StandardConfig();
$config->setOptions(array(
    'remember_me_seconds' => 1800,
    'name'                => 'zf2',
));
$manager = new SessionManager($config);
```

CHAPTER

THREE

SESSION CONFIG

`Zend\Session\Config\SessionConfig` provides you a basic interface for implementing sessions when that leverage PHP's ext/session. Most configuration options configure either the `Zend\Session\Storage` OR configure ext/session directly.

223.1 Basic Configuration Options

The following configuration options are defined by `Zend\Session\Config\SessionConfig`, note that it inherits all configuration from `Zend\Session\Config\StandardConfig`.

Option	Data Type	Description
cache_limiter	`string`	Specifies the cache control method used for session pages.
hash_function	`string`	Allows you to specify the hash algorithm used to generate the session IDs.
php_save_handler	`string`	Defines the name of a PHP save_handler embedded into PHP.
serialize_handler	`string`	Defines the name of the handler which is used to serialize/deserialize data.
url_rewriter_tags	`string`	Specifies which HTML tags are rewritten to include session id if transparent sid enabled.
use_trans_sid	`boolean`	Whether transparent sid support is enabled or not.

CHAPTER

FOUR

BASIC USAGE

A basic example is one like the following:

```
use Zend\Session\Config\SessionConfig;
use Zend\Session\SessionManager;

$config = new SessionConfig();
$config->setOptions(array(
    'phpSaveHandler' => 'redis',
    'savePath' => 'tcp://127.0.0.1:6379?weight=1&timeout=1',
));
$manager = new SessionManager($config);
```

224.1 Custom Configuration

In the event that you prefer to create your own session configuration; you *must* implement `Zend\Session\Config\ConfigInterface` which contains the basic interface for items needed when implementing a session. This includes cookie configuration, lifetime, session name, save path and an interface for getting and setting options.

CHAPTER

FIVE

SESSION CONTAINER

`Zend\Session\Container` instances provide the primary API for manipulating session data in the Zend Framework. Containers are used to segregate all session data, although a default namespace exists for those who only want one namespace for all their session data.

Each instance of `Zend\Session\Container` corresponds to an entry of the `Zend\Session\Storage`, where the namespace is used as the key. `Zend\Session\Container` itself is an instance of an ArrayObject.

225.1 Basic Usage

```
use Zend\Session\Container;

$container = new Container('namespace');
$container->item = 'foo';
```

225.2 Setting the Default Session Manager

In the event you are using multiple session managers or prefer to be explicit, the default session manager that is utilized can be explicitly set.

```
use Zend\Session\Container;
use Zend\Session\SessionManager;

$manager = new SessionManager();
Container::setDefaultManager($manager);
```

CHAPTER

SIX

SESSION MANAGER

The session manager, `Zend\Session\SessionManager`, is a class that is responsible for all aspects of session management. It initializes and configures configuration, storage and save handling. Additionally the session manager can be injected into the session container to provide a wrapper or namespace around your session data.

The session manager is responsible for session start, session exists, session write, regenerate id, time to live and session destroy. The session manager can validate sessions from a validator chain to ensure that the session data is indeed correct.

226.1 Initializing the Session Manager

Generally speaking you will always want to initialize the session manager and ensure that you had initialized it on your end; this puts in place a simple solution to prevent against session fixation. Generally you will setup configuration and then inside of your Application module bootstrap the session manager.

Additionally you will likely want to supply validators to prevent against session hijacking.

The following illustrates how you may configure session manager by setting options in your local or global config:

```
return array(
    'session' => array(
        'config' => array(
            'class' => 'Zend\Session\Config\SessionConfig',
            'options' => array(
                'name' => 'myapp',
            ),
        ),
        'storage' => 'Zend\Session\Storage\SessionArrayStorage',
        'validators' => array(
            array(
                'Zend\Session\Validator\RemoteAddr',
                'Zend\Session\Validator\HttpUserAgent',
            ),
        ),
    ),
);
```

The following illustrates how you might utilize the above configuration to create the session manager:

```
use Zend\Session\SessionManager;
use Zend\Session\Container;

class Module
```

```
5  {
6      public function onBootstrap($e)
7      {
8          $eventManager        = $e->getApplication()->getEventManager();
9          $serviceManager      = $e->getApplication()->getServiceManager();
10         $moduleRouteListener = new ModuleRouteListener();
11         $moduleRouteListener->attach($eventManager);
12         $this->bootstrapSession($e);
13     }
14
15     public function bootstrapSession($e)
16     {
17         $session = $e->getApplication()
18                      ->getServiceManager()
19                      ->get('Zend\Session\SessionManager');
20         $session->start();
21
22         $container = new Container('initialized');
23         if (!isset($container->init)) {
24              $session->regenerateId(true);
25              $container->init = 1;
26         }
27     }
28
29     public function getServiceConfig()
30     {
31         return array(
32             'factories' => array(
33                 'Zend\Session\SessionManager' => function ($sm) {
34                     $config = $sm->get('config');
35                     if (isset($config['session'])) {
36                         $session = $config['session'];
37
38                         $sessionConfig = null;
39                         if (isset($session['config'])) {
40                             $class = isset($session['config']['class'])  ? $session['config']['class'
41                             $options = isset($session['config']['options']) ? $session['config']['opt
42                             $sessionConfig = new $class();
43                             $sessionConfig->setOptions($options);
44                         }
45
46                         $sessionStorage = null;
47                         if (isset($session['storage'])) {
48                             $class = $session['storage'];
49                             $sessionStorage = new $class();
50                         }
51
52                         $sessionSaveHandler = null;
53                         if (isset($session['save_handler'])) {
54                             // class should be fetched from service manager since it will require co
55                             $sessionSaveHandler = $sm->get($session['save_handler']);
56                         }
57
58                         $sessionManager = new SessionManager($sessionConfig, $sessionStorage, $sessio
59
60                         if (isset($session['validator'])) {
61                             $chain = $sessionManager->getValidatorChain();
62                             foreach ($session['validator'] as $validator) {
```

```
                                $validator = new $validator();
                                $chain->attach('session.validate', array($validator, 'isValid'));

                            }
                        }
                    } else {
                        $sessionManager = new SessionManager();
                    }
                    Container::setDefaultManager($sessionManager);
                    return $sessionManager;
                },
            ),
        )
    }
}
```

When you create a new `Zend\Session\Container` (see Session Container page) in a controller for example, it will use the session configured above.

226.2 Session Compatibility

In order to work with other 3rd party libraries and share sessions across software that may not be ZF2 related; you will need to ensure that you still provide access to the ZF2 autoloader as well as module autoloading.

In the shared software make certain before the session starts that you bootstrap the ZF2 autoloader and initialize the ZF2 Application.

```
$cwd = getcwd();
chdir('/path/to/zf2-application');
require 'init_autoloader.php';
Zend\Mvc\Application::init(require 'config/application.config.php');
chdir($cwd);
session_start();
```

CHAPTER

SEVEN

SESSION SAVE HANDLERS

Zend Framework comes with a standard set of save handler classes which are ready for you to use. Save Handlers themselves are decoupled from PHP's save handler functions and are *only* implemented as a PHP save handler when utilized in conjunction with `Zend\Session\SessionManager`.

CHAPTER

EIGHT

CACHE

`Zend\Session\SaveHandler\Cache` allows you to provide an instance of `Zend\Cache` to be utilized as a session save handler. Generally if you are utilizing the Cache save handler; you are likely using products such as memcached.

228.1 Basic usage

A basic example is one like the following:

```
use Zend\Cache\StorageFactory;
use Zend\Session\SaveHandler\Cache;
use Zend\Session\SessionManager;

$cache = StorageFactory::factory(array(
    'name' => 'memcached',
    'options' => array(
        'server' => '127.0.0.1',
    ),
));
$saveHandler = new Cache($cache);
$manager = new SessionManager();
$manager->setSaveHandler($saveHandler);
```

CHAPTER

NINE

DBTABLEGATEWAY

`Zend\Session\SaveHandler\DbTableGateway` allows you to utilize `Zend\Db` as a session save handler. Setup of the DbTableGateway requires an instance of `Zend\Db\TableGateway\TableGateway` and an instance of `Zend\Session\SaveHandler\DbTableGatewayOptions`. In the most basic setup, a TableGateway object and using the defaults of the DbTableGatewayOptions will provide you with what you need.

229.1 Creating the database table

```
CREATE TABLE `session` (
    `id` char(32),
    `name` char(32),
    `modified` int,
    `lifetime` int,
    `data` text,
     PRIMARY KEY (`id`, `name`)
);
```

229.2 Basic usage

A basic example is one like the following:

```
use Zend\Db\TableGateway\TableGateway;
use Zend\Session\SaveHandler\DbTableGateway;
use Zend\Session\SaveHandler\DbTableGatewayOptions;
use Zend\Session\SessionManager;

$tableGateway = new TableGateway('session', $adapter);
$saveHandler = new DbTableGateway($tableGateway, new DbTableGatewayOptions());
$manager = new SessionManager();
$manager->setSaveHandler($saveHandler);
```

CHAPTER

MONGODB

`Zend\Session\SaveHandler\MongoDB` allows you to provide a MongoDB instance to be utilized as a session save handler. You provide the options in the `Zend\Session\SaveHandler\MongoDBOptions` class.

230.1 Basic Usage

A basic example is one like the following:

```
use Mongo;
use Zend\Session\SaveHandler\MongoDB;
use Zend\Session\SaveHandler\MongoDBOptions;
use Zend\Session\SessionManager;

$mongo = new Mongo();
$options = new MongoDBOptions(array(
    'database' => 'myapp',
    'collection' => 'sessions',
));
$saveHandler = new MongoDB($mongo, $options);
$manager = new SessionManager();
$manager->setSaveHandler($saveHandler);
```

230.2 Custom Save Handlers

There may be cases where you want to create a save handler where a save handler currently does not exist. Creating a custom save handler is much like creating a custom PHP save handler. All save handlers *must* implement `Zend\Session\SaveHandler\SaveHandlerInterface`. Generally if your save handler has options you will create another options class for configuration of the save handler.

CHAPTER

ONE

SESSION STORAGE

Zend Framework comes with a standard set of storage classes which are ready for you to use. Storage handlers is the intermediary between when the session starts and when the session writes and closes. The default session storage is `Zend\Session\Storage\SessionArrayStorage`.

CHAPTER

TWO

ARRAY STORAGE

`Zend\Session\Storage\ArrayStorage` provides a facility to store all information in an ArrayObject. This storage method is likely incompatible with 3rd party libraries and all properties will be inaccessible through the $_SESSION property. Additionally ArrayStorage will not automatically repopulate the storage container in the case of each new request and would have to manually be re-populated.

232.1 Basic Usage

A basic example is one like the following:

```
use Zend\Session\Storage\ArrayStorage;
use Zend\Session\SessionManager;

$populateStorage = array('foo' => 'bar');
$storage = new ArrayStorage($populateStorage);
$manager = new SessionManager();
$manager->setStorage($storage);
```

CHAPTER

THREE

SESSION STORAGE

`Zend\Session\Storage\SessionStorage` replaces $_SESSION providing a facility to store all information in an ArrayObject. This means that it may not be compatible with 3rd party libraries. Although information stored in the $_SESSION superglobal should be available in other scopes.

233.1 Basic Usage

A basic example is one like the following:

```
use Zend\Session\Storage\SessionStorage;
use Zend\Session\SessionManager;

$manager = new SessionManager();
$manager->setStorage(new SessionStorage());
```

CHAPTER

FOUR

SESSION ARRAY STORAGE

`Zend\Session\Storage\SessionArrayStorage` provides a facility to store all information directly in the $_SESSION superglobal. This storage class provides the most compatibility with 3rd party libraries and allows for directly storing information into $_SESSION.

234.1 Basic Usage

A basic example is one like the following:

```
use Zend\Session\Storage\SessionArrayStorage;
use Zend\Session\SessionManager;

$manager = new SessionManager();
$manager->setStorage(new SessionArrayStorage());
```

234.2 Custom Storage

In the event that you prefer a different type of storage; to create a new custom storage container, you *must* implement `Zend\Session\Storage\StorageInterface` which is mostly in implementing ArrayAccess, Traversable, Serializable and Countable. StorageInterface defines some additional functionality that must be implemented.

CHAPTER

FIVE

SESSION VALIDATORS

Session validators provide various protection against session hijacking. Session hijacking in particular has various drawbacks when you are protecting against it. Such as an IP address may change from the end user depending on their ISP; or a browsers user agent may change during the request either by a web browser extension OR an upgrade that retains session cookies.

CHAPTER

SIX

HTTP USER AGENT

`Zend\Session\Validator\HttpUserAgent` provides a validator to check the session against the originally stored $_SERVER['HTTP_USER_AGENT'] variable. Validation will fail in the event that this does not match and throws an exception in `Zend\Session\SessionManager` after session_start() has been called.

236.1 Basic Usage

A basic example is one like the following:

```
use Zend\Session\Validator\HttpUserAgent;
use Zend\Session\SessionManager;

$manager = new SessionManager();
$manager->getValidatorChain()->attach('session.validate', array(new HttpUserAgent(), 'isValid'));
```

CHAPTER

SEVEN

REMOTE ADDR

`Zend\Session\Validator\RemoteAddr` provides a validator to check the session against the originally stored $_SERVER['REMOTE_ADDR'] variable. Validation will fail in the event that this does not match and throws an exception in `Zend\Session\SessionManager` after session_start() has been called.

237.1 Basic Usage

A basic example is one like the following:

```
use Zend\Session\Validator\RemoteAddr;
use Zend\Session\SessionManager;

$manager = new SessionManager();
$manager->getValidatorChain()->attach('session.validate', array(new RemoteAddr(), 'isValid'));
```

237.2 Custom Validators

You may want to provide your own custom validators to validate against other items from storing a token and validating a token to other various techniques. To create a custom validator you *must* implement the validation interface `Zend\Session\Validator\ValidatorInterface`.

CHAPTER

EIGHT

ZEND\SOAP\SERVER

`Zend\Soap\Server` class is intended to simplify Web Services server part development for *PHP* programmers.

It may be used in WSDL or non-WSDL mode, and using classes or functions to define Web Service *API*.

When `Zend\Soap\Server` component works in the WSDL mode, it uses already prepared WSDL document to define server object behavior and transport layer options.

WSDL document may be auto-generated with functionality provided by *Zend\Soap\AutoDiscovery component* or should be constructed manually using *Zend\Soap\Wsdl class* or any other *XML* generating tool.

If the non-WSDL mode is used, then all protocol options have to be set using options mechanism.

238.1 Zend\Soap\Server constructor

`Zend\Soap\Server` constructor should be used a bit differently for WSDL and non-WSDL modes.

238.1.1 Zend\Soap\Server constructor for the WSDL mode

`Zend\Soap\Server` constructor takes two optional parameters when it works in WSDL mode:

- `$wsdl`, which is an *URI* of a WSDL file [1].
- `$options`- options to create *SOAP* server object [2].

The following options are recognized in the WSDL mode:

- 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or *SOAP*_1_2).
- 'actor' - the actor *URI* for the server.
- 'classmap' ('classMap') which can be used to map some WSDL types to *PHP* classes. The option must be an array with WSDL types as keys and names of *PHP* classes as values.
- 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
- 'wsdl' which is equivalent to `setWsdl($wsdlValue)` call.

[1] May be set later using `setWsdl($wsdl)` method.
[2] Options may be set later using `setOptions($options)` method.

238.1.2 Zend\Soap\Server constructor for the non-WSDL mode

The first constructor parameter **must** be set to `NULL` if you plan to use `Zend\Soap\Server` functionality in non-WSDL mode.

You also have to set 'uri' option in this case (see below).

The second constructor parameter (`$options`) is an array with options to create *SOAP* server object [3].

The following options are recognized in the non-WSDL mode:

- 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or SOAP_1_2).
- 'actor' - the actor *URI* for the server.
- 'classmap' ('classMap') which can be used to map some WSDL types to *PHP* classes. The option must be an array with WSDL types as keys and names of *PHP* classes as values.
- 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
- 'uri' (required) - *URI* namespace for *SOAP* server.

238.2 Methods to define Web Service API

There are two ways to define Web Service *API* when your want to give access to your *PHP* code through *SOAP*.

The first one is to attach some class to the `Zend\Soap\Server` object which has to completely describe Web Service *API*:

```
...
class MyClass {
    /**
     * This method takes ...
     *
     * @param integer $inputParam
     * @return string
     */
    public function method1($inputParam) {
        ...
    }

    /**
     * This method takes ...
     *
     * @param integer $inputParam1
     * @param string  $inputParam2
     * @return float
     */
    public function method2($inputParam1, $inputParam2) {
        ...
    }

    ...
}
...
$server = new Zend\Soap\Server(null, $options);
// Bind Class to Soap Server
$server->setClass('MyClass');
```

[3] Options may be set later using `setOptions($options)` method.

```
// Bind already initialized object to Soap Server
$server->setObject(new MyClass());
...
$server->handle();
```

Note: Important!

You should completely describe each method using method docblock if you plan to use autodiscover functionality to prepare corresponding Web Service WSDL.

The second method of defining Web Service *API* is using set of functions and `addFunction()` or `loadFunctions()` methods:

```
...
/**
 * This function ...
 *
 * @param integer $inputParam
 * @return string
 */
function function1($inputParam) {
    ...
}

/**
 * This function ...
 *
 * @param integer $inputParam1
 * @param string  $inputParam2
 * @return float
 */
function function2($inputParam1, $inputParam2) {
    ...
}
...
$server = new Zend\Soap\Server(null, $options);
$server->addFunction('function1');
$server->addFunction('function2');
...
$server->handle();
```

238.3 Request and response objects handling

Note: Advanced

This section describes advanced request/response processing options and may be skipped.

`Zend\Soap\Server` component performs request/response processing automatically, but allows to catch it and do some pre- and post-processing.

238.3.1 Request processing

`Zend\Soap\Server::handle()` method takes request from the standard input stream ('php://input'). It may be overridden either by supplying optional parameter to the `handle()` method or by setting request using `setRequest()` method:

```
...
$server = new Zend\Soap\Server(...);
...
// Set request using optional $request parameter
$server->handle($request);
...
// Set request using setRequest() method
$server->setRequest();
$server->handle();
```

Request object may be represented using any of the following:

- DOMDocument (casted to *XML*)
- DOMNode (owner document is grabbed and casted to *XML*)
- SimpleXMLElement (casted to *XML*)
- stdClass (__toString() is called and verified to be valid *XML*)
- string (verified to be valid *XML*)

Last processed request may be retrieved using `getLastRequest()` method as an *XML* string:

```
...
$server = new Zend\Soap\Server(...);
...
$server->handle();
$request = $server->getLastRequest();
```

238.3.2 Response pre-processing

`Zend\Soap\Server::handle()` method automatically emits generated response to the output stream. It may be blocked using `setReturnResponse()` with `TRUE` or `FALSE` as a parameter [4]. Generated response is returned by `handle()` method in this case. Returned response can be a string or a SoapFault exception object.

> **Caution:** Check always the returned response type for avoid return SoapFault object as string, which will return to the customer a string with the exception stacktrace.

```
...
$server = new Zend\Soap\Server(...);
...
// Get a response as a return value of handle() method
// instead of emitting it to the standard output
$server->setReturnResponse(true);
...
$response = $server->handle();
if ($response instanceof \SoapFault) {
    ...
} else {
    ...
```

[4] Current state of the Return Response flag may be requested with `setReturnResponse()` method.

```
}
...
```

Last response may be also retrieved by `getLastResponse()` method for some post-processing:

```
...
$server = new Zend\Soap\Server(...);
...
$server->handle();
$response = $server->getLastResponse();
if ($response instanceof \SoapFault) {
    ...
} else {
    ...
}
...
```

238.4 Document/Literal WSDL Handling

Using the document/literal binding-style/encoding pattern is used to make SOAP messages as human-readable as possible and allow abstraction between very incompatible languages. The Dot NET framework uses this pattern for SOAP service generation by default. The central concept of this approach to SOAP is the introduction of a Request and an Response object for every function/method of the SOAP service. The parameters of the function are properties on request object and the response object contains a single parameter that is built in the style "methodName"Result

Zend SOAP supports this pattern in both AutoDiscovery and in the Server component. You can write your service object without knowledge about using this pattern. Use docblock comments to hint the parameter and return types as usual. The `Zend\Soap\Server\DocumentLiteralWrapper` wraps around your service object and converts request and response into normal method calls on your service.

See the class doc block of the `DocumentLiteralWrapper` for a detailed example and discussion.

CHAPTER

NINE

ZEND\SOAP\CLIENT

The `Zend\Soap\Client` class simplifies *SOAP* client development for *PHP* programmers.

It may be used in WSDL or non-WSDL mode.

Under the WSDL mode, the `Zend\Soap\Client` component uses a WSDL document to define transport layer options.

The WSDL description is usually provided by the web service the client will access. If the WSDL description is not made available, you may want to use `Zend\Soap\Client` in non-WSDL mode. Under this mode, all *SOAP* protocol options have to be set explicitly on the `Zend\Soap\Client` class.

239.1 Zend\Soap\Client Constructor

The `Zend\Soap\Client` constructor takes two parameters:

- `$wsdl`- the *URI* of a WSDL file.
- `$options`- options to create *SOAP* client object.

Both of these parameters may be set later using `setWsdl($wsdl)` and `setOptions($options)` methods respectively.

Note: Important!

If you use `Zend\Soap\Client` component in non-WSDL mode, you **must** set the 'location' and 'uri' options.

The following options are recognized:

- 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or *SOAP*_1_2).
- 'classmap' ('classMap') - can be used to map some WSDL types to *PHP* classes. The option must be an array with WSDL types as keys and names of *PHP* classes as values.
- 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
- 'wsdl' which is equivalent to `setWsdl($wsdlValue)` call. Changing this option may switch `Zend\Soap\Client` object to or from WSDL mode.
- 'uri' - target namespace for the *SOAP* service (required for non-WSDL-mode, doesn't work for WSDL mode).
- 'location' - the *URL* to request (required for non-WSDL-mode, doesn't work for WSDL mode).
- 'style' - request style (doesn't work for WSDL mode): `SOAP_RPC` or `SOAP_DOCUMENT`.
- 'use' - method to encode messages (doesn't work for WSDL mode): `SOAP_ENCODED` or `SOAP_LITERAL`.

- 'login' and 'password' - login and password for an *HTTP* authentication.
- 'proxy_host', 'proxy_port', 'proxy_login', and 'proxy_password' - an *HTTP* connection through a proxy server.
- 'local_cert' and 'passphrase' -*HTTPS* client certificate authentication options.
- 'compression' - compression options; it's a combination of `SOAP_COMPRESSION_ACCEPT`, `SOAP_COMPRESSION_GZIP` and `SOAP_COMPRESSION_DEFLATE` options which may be used like this:

```
// Accept response compression
$client = new Zend\Soap\Client("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT));
...

// Compress requests using gzip with compression level 5
$client = new Zend\Soap\Client("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_GZIP | 5));
...

// Compress requests using deflate compression
$client = new Zend\Soap\Client("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_DEFLATE));
```

239.2 Performing SOAP Requests

After we've created a `Zend\Soap\Client` object we are ready to perform *SOAP* requests.

Each web service method is mapped to the virtual `Zend\Soap\Client` object method which takes parameters with common *PHP* types.

Use it like in the following example:

```
//*****************************************************************
//                 Server code
//*****************************************************************
// class MyClass {
//     /**
//      * This method takes ...
//      *
//      * @param integer $inputParam
//      * @return string
//      */
//     public function method1($inputParam) {
//         ...
//     }
//
//     /**
//      * This method takes ...
//      *
//      * @param integer $inputParam1
//      * @param string  $inputParam2
//      * @return float
//      */
//     public function method2($inputParam1, $inputParam2) {
//         ...
//     }
//
```

```
//      ...
// }
// ...
// $server = new Zend\Soap\Server(null, $options);
// $server->setClass('MyClass');
// ...
// $server->handle();
//
//*****************************************************************
//                 End of server code
//*****************************************************************

$client = new Zend\Soap\Client("MyService.wsdl");
...

// $result1 is a string
$result1 = $client->method1(10);
...

// $result2 is a float
$result2 = $client->method2(22, 'some string');
```

CHAPTER

WSDL ACCESSOR

Note: `Zend\Soap\Wsdl` class is used by `Zend\Soap\Server` component internally to operate with WSDL documents. Nevertheless, you could also use functionality provided by this class for your own needs. The `Zend\Soap\Wsdl` package contains both a parser and a builder of WSDL documents.

If you don't plan to do this, you can skip this documentation section.

240.1 Zend\Soap\Wsdl constructor

`Zend\Soap\Wsdl` constructor takes three parameters:

- `$name` - name of the Web Service being described.
- `$uri` - *URI* where the WSDL will be available (could also be a reference to the file in the filesystem.)
- `$strategy` - optional flag used to identify the strategy for complex types (objects) detection. To read more on complex type detection strategies go to the section: *Add complex types*.
- `$classMap` - Optional array of class name translations from PHP Type (key) to WSDL type (value).

240.2 addMessage() method

`addMessage($name, $parts)` method adds new message description to the WSDL document (/definitions/message element).

Each message correspond to methods in terms of `Zend\Soap\Server` and `Zend\Soap\Client` functionality.

`$name` parameter represents the message name.

`$parts` parameter is an array of message parts which describes *SOAP* call parameters. It's an associative array: 'part name' (SOAP call parameter name) => 'part type'.

Type mapping management is performed using `addTypes()`, `addTypes()` and `addComplexType()` methods (see below).

Note: Messages parts can use either 'element' or 'type' attribute for typing (see http://www.w3.org/TR/wsdl#_messages).

'element' attribute must refer to a corresponding element of data type definition. 'type' attribute refers to a corresponding complexType entry.

All standard XSD types have both 'element' and 'complexType' definitions (see http://schemas.xmlsoap.org/soap/encoding/).

All non-standard types, which may be added using `Zend\Soap\Wsdl::addComplexType()` method, are described using 'complexType' node of '/definitions/types/schema/' section of WSDL document.

So `addMessage()` method always uses 'type' attribute to describe types.

240.3 addPortType() method

`addPortType($name)` method adds new port type to the WSDL document (/definitions/portType) with the specified port type name.

It joins a set of Web Service methods defined in terms of `Zend\Soap\Server` implementation.

See http://www.w3.org/TR/wsdl#_porttypes for the details.

240.4 addPortOperation() method

`addPortOperation($portType, $name, $input = false, $output = false, $fault = false)` method adds new port operation to the specified port type of the WSDL document (/definitions/portType/operation).

Each port operation corresponds to a class method (if Web Service is based on a class) or function (if Web Service is based on a set of methods) in terms of `Zend\Soap\Server` implementation.

It also adds corresponding port operation messages depending on specified `$input`, `$output` and `$fault` parameters.

Note: `Zend\Soap\Server` component generates two messages for each port operation while describing service based on `Zend\Soap\Server` class:

- input message with name *$methodName . 'Request'*.
- output message with name *$methodName . 'Response'*.

See http://www.w3.org/TR/wsdl#_request-response for the details.

240.5 addBinding() method

`addBinding($name, $portType)` method adds new binding to the WSDL document (/definitions/binding).

'binding' WSDL document node defines message format and protocol details for operations and messages defined by a particular portType (see http://www.w3.org/TR/wsdl#_bindings).

The method creates binding node and returns it. Then it may be used to fill with actual data.

`Zend\Soap\Server` implementation uses *$serviceName . 'Binding'* name for 'binding' element of WSDL document.

240.6 addBindingOperation() method

`addBindingOperation($binding, $name, $input = false, $output = false, $fault = false)` method adds an operation to a binding element (/definitions/binding/operation) with the specified name.

It takes an *XML_Tree_Node* object returned by `addBinding()` as an input (`$binding` parameter) to add 'operation' element with input/output/false entries depending on specified parameters

`Zend\Soap\Server` implementation adds corresponding binding entry for each Web Service method with input and output entries defining 'soap:body' element as '<soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>

See http://www.w3.org/TR/wsdl#_bindings for the details.

240.7 addSoapBinding() method

`addSoapBinding($binding, $style = 'document', $transport = 'http://schemas.xmlsoap.org/soap/http')` method adds *SOAP* binding ('soap:binding') entry to the binding element (which is already linked to some port type) with the specified style and transport (Zend\Soap\Server implementation uses RPC style over *HTTP*).

'/definitions/binding/soap:binding' element is used to signify that the binding is bound to the *SOAP* protocol format.

See http://www.w3.org/TR/wsdl#_bindings for the details.

240.8 addSoapOperation() method

`addSoapOperation($binding, $soap_action)` method adds *SOAP* operation ('soap:operation') entry to the binding element with the specified action. 'style' attribute of the 'soap:operation' element is not used since programming model (RPC-oriented or document-oriented) may be using `addSoapBinding()` method

'soapAction' attribute of '/definitions/binding/soap:operation' element specifies the value of the *SOAP*Action header for this operation. This attribute is required for *SOAP* over *HTTP* and **must not** be specified for other transports.

`Zend\Soap\Server` implementation uses *$serviceUri . '#' . $methodName* for *SOAP* operation action name.

See http://www.w3.org/TR/wsdl#_soap:operation for the details.

240.9 addService() method

`addService($name, $port_name, $binding, $location)` method adds '/definitions/service' element to the WSDL document with the specified Wed Service name, port name, binding, and location.

WSDL 1.1 allows to have several port types (sets of operations) per service. This ability is not used by `Zend\Soap\Server` implementation and not supported by `Zend\Soap\Wsdl` class.

`Zend\Soap\Server` implementation uses:

- *$name . 'Service'* as a Web Service name,
- *$name . 'Port'* as a port type name,
- *'tns:' . $name . 'Binding'* [1] as binding name,
- script *URI* [2] as a service URI for Web Service definition using classes.

[1] *'tns:' namespace* is defined as script *URI* (*'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']*).
[2] *'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']*

where `$name` is a class name for the Web Service definition mode using class and script name for the Web Service definition mode using set of functions.

See http://www.w3.org/TR/wsdl#_services for the details.

240.10 Type mapping

`ZendSoap` WSDL accessor implementation uses the following type mapping between *PHP* and *SOAP* types:

- PHP strings <-> *xsd:string*.
- PHP integers <-> *xsd:int*.
- PHP floats and doubles <-> *xsd:float*.
- PHP booleans <-> *xsd:boolean*.
- PHP arrays <-> *soap-enc:Array*.
- PHP object <-> *xsd:struct*.
- *PHP* class <-> based on complex type strategy (See: *this section*) [3].
- PHP void <-> empty type.
- If type is not matched to any of these types by some reason, then *xsd:anyType* is used.

Where *xsd:* is "http://www.w3.org/2001/XMLSchema" namespace, *soap-enc:* is a "http://schemas.xmlsoap.org/soap/encoding/" namespace, *tns:* is a "target namespace" for a service.

240.10.1 Retrieving type information

`getType($type)` method may be used to get mapping for a specified *PHP* type:

```
...
$wsdl = new Zend\Soap\Wsdl('My_Web_Service', $myWebServiceUri);

...
$soapIntType = $wsdl->getType('int');

...
class MyClass {
    ...
}
...
$soapMyClassType = $wsdl->getType('MyClass');
```

240.10.2 Adding complex type information

`addComplexType($type)` method is used to add complex types (PHP classes) to a WSDL document.

[3] By default `Zend\Soap\Wsdl` will be created with the `Zend\Soap\Wsdl\ComplexTypeStrategy\DefaultComplexType` class as detection algorithm for complex types. The first parameter of the AutoDiscover constructor takes any complex type strategy implementing `Zend\Soap\Wsdl\ComplexTypeStrategy\ComplexTypeStrategyInterface` or a string with the name of the class. For backwards compatibility with `$extractComplexType` boolean variables are parsed the following way: If `TRUE`, `Zend\Soap\Wsdl\ComplexTypeStrategy\DefaultComplexType`, if `FALSE` `Zend\Soap\Wsdl\ComplexTypeStrategy\AnyType`.

It's automatically used by `getType()` method to add corresponding complex types of method parameters or return types.

Its detection and building algorithm is based on the currently active detection strategy for complex types. You can set the detection strategy either by specifying the class name as string or instance of a `Zend\Soap\Wsdl\ComplexTypeStrategy` implementation as the third parameter of the constructor or using the `setComplexTypeStrategy($strategy)` function of `Zend\Soap\Wsdl`. The following detection strategies currently exist:

- Class `Zend\Soap\Wsdl\ComplexTypeStrategy\DefaultComplexType`: Enabled by default (when no third constructor parameter is set). Iterates over the public attributes of a class type and registers them as subtypes of the complex object type.
- Class `Zend\Soap\Wsdl\ComplexTypeStrategy\AnyType`: Casts all complex types into the simple XSD type xsd:anyType. Be careful this shortcut for complex type detection can probably only be handled successfully by weakly typed languages such as *PHP*.
- Class `Zend\Soap\Wsdl\ComplexTypeStrategy\ArrayOfTypeSequence`: This strategy allows to specify return parameters of the type: *int[]* or *string[]*. As of Zend Framework version 1.9 it can handle both simple *PHP* types such as int, string, boolean, float as well as objects and arrays of objects.
- Class `Zend\Soap\Wsdl\ComplexTypeStrategy\ArrayOfTypeComplex`: This strategy allows to detect very complex arrays of objects. Objects types are detected based on the `Zend\Soap\Wsdl\Strategy\DefaultComplexType` and an array is wrapped around that definition.
- Class `Zend\Soap\Wsdl\ComplexTypeStrategy\Composite`: This strategy can combine all strategies by connecting *PHP* Complex types (Classnames) to the desired strategy via the `connectTypeToStrategy($type, $strategy)` method. A complete typemap can be given to the constructor as an array with `$type-> $strategy` pairs. The second parameter specifies the default strategy that will be used if an unknown type is requested for adding. This parameter defaults to the `Zend\Soap\Wsdl\Strategy\DefaultComplexType` strategy.

`addComplexType()` method creates '/definitions/types/xsd:schema/xsd:complexType' element for each described complex type with name of the specified *PHP* class.

Class property **MUST** have docblock section with the described *PHP* type to have property included into WSDL description.

`addComplexType()` checks if type is already described within types section of the WSDL document.

It prevents duplications if this method is called two or more times and recursion in the types definition section.

See http://www.w3.org/TR/wsdl#_types for the details.

240.11 addDocumentation() method

`addDocumentation($input_node, $documentation)` method adds human readable documentation using optional 'wsdl:document' element.

'/definitions/binding/soap:binding' element is used to signify that the binding is bound to the *SOAP* protocol format.

See http://www.w3.org/TR/wsdl#_documentation for the details.

240.12 Get finalized WSDL document

`toXML()`, `toDomDocument()` and `dump($filename = false)` methods may be used to get WSDL document as an *XML*, DOM structure or a file.

CHAPTER

ONE

AUTODISCOVERY

241.1 AutoDiscovery Introduction

SOAP functionality implemented within Zend Framework is intended to make all steps required for *SOAP* communications more simple.

SOAP is language independent protocol. So it may be used not only for *PHP*-to-PHP communications.

There are three configurations for *SOAP* applications where Zend Framework may be utilized:

- SOAP server *PHP* application <—> *SOAP* client *PHP* application
- SOAP server non-PHP application <—> *SOAP* client *PHP* application
- SOAP server *PHP* application <—> *SOAP* client non-PHP application

We always have to know, which functionality is provided by *SOAP* server to operate with it. WSDL is used to describe network service *API* in details.

WSDL language is complex enough (see http://www.w3.org/TR/wsdl for the details). So it's difficult to prepare correct WSDL description.

Another problem is synchronizing changes in network service *API* with already existing WSDL.

Both these problem may be solved by WSDL autogeneration. A prerequisite for this is a *SOAP* server autodiscovery. It constructs object similar to object used in *SOAP* server application, extracts necessary information and generates correct WSDL using this information.

There are two ways for using Zend Framework for *SOAP* server application:

- Use separated class.
- Use set of functions.

Both methods are supported by Zend Framework Autodiscovery functionality.

The `Zend\Soap\AutoDiscover` class also supports datatypes mapping from *PHP* to XSD types.

Here is an example of common usage of the autodiscovery functionality. The `generate()` function generates the WSDL object and in conjunction with `toXml()` function you can posts it to the browser.

```
class MySoapServerClass {
...
}

$autodiscover = new Zend\Soap\AutoDiscover();
$autodiscover->setClass('MySoapServerClass')
             ->setUri('http://localhost/server.php')
```

```
                ->setServiceName('MySoapService');
$wsdl = $autodiscover->generate();
echo $wsdl->toXml();
$wsdl->dump("/path/to/file.wsdl");
$dom = $wsdl->toDomDocument();
```

Note: ZendSoapAutodiscover is not a Soap Server

It is very important to note, that the class `Zend\Soap\AutoDiscover` does not act as a *SOAP* Server on its own.

```
if (isset($_GET['wsdl'])) {
    $autodiscover = new Zend\Soap\AutoDiscover();
    $autodiscover->setClass('HelloWorldService')
                  ->setUri('http://example.com/soap.php');
    echo $autodiscover->toXml();
} else {
    // pointing to the current file here
    $soap = new Zend\Soap\Server("http://example.com/soap.php?wsdl");
    $soap->setClass('HelloWorldService');
    $soap->handle();
}
```

241.2 Class autodiscovering

If a class is used to provide SOAP server functionality, then the same class should be provided to `Zend\Soap\AutoDiscover` for WSDL generation:

```
$autodiscover = new Zend\Soap\AutoDiscover();
$autodiscover->setClass('My_SoapServer_Class')
              ->setUri('http://localhost/server.php')
              ->setServiceName('MySoapService');
$wsdl = $autodiscover->generate();
```

The following rules are used while WSDL generation:

- Generated WSDL describes an RPC/Encoded style Web Service. If you want to use a document/literal server use the `setBindingStyle()` and `setOperationBodyStyle()` methods.
- Class name is used as a name of the Web Service being described unless `setServiceName()` is used explicitly to set the name. When only functions are used for generation the service name has to be set explicitly or an exception is thrown during generation of the WSDL document.
- You can set the endpoint of the actual SOAP Server via the `setUri()` method. This is a required option.

It's also used as a target namespace for all service related names (including described complex types).

- Class methods are joined into one Port Type. *$serviceName* . *'Port'* is used as Port Type name.
- Each class method/function is registered as a corresponding port operation.
- Only the "longest" available method prototype is used for generation of the WSDL.
- WSDL autodiscover utilizes the *PHP* docblocks provided by the developer to determine the parameter and return types. In fact, for scalar types, this is the only way to determine the parameter types, and for return types, this is the only way to determine them. That means, providing correct and fully detailed docblocks is not only best practice, but is required for discovered class.

241.3 Functions autodiscovering

If set of functions are used to provide SOAP server functionality, then the same set should be provided to `Zend\Soap\AutoDiscovery` for WSDL generation:

```
$autodiscover = new Zend\Soap\AutoDiscover();
$autodiscover->addFunction('function1');
$autodiscover->addFunction('function2');
$autodiscover->addFunction('function3');
...
$wsdl = $autodiscover->generate();
```

The same rules apply to generation as described in the class autodiscover section above.

241.4 Autodiscovering Datatypes

Input/output datatypes are converted into network service types using the following mapping:

- PHP strings <-> *xsd:string*.
- PHP integers <-> *xsd:int*.
- PHP floats and doubles <-> *xsd:float*.
- PHP booleans <-> *xsd:boolean*.
- PHP arrays <-> *soap-enc:Array*.
- PHP object <-> *xsd:struct*.
- *PHP* class <-> based on complex type strategy (See: *this section*) [1].
- type[] or object[] (ie. int[]) <-> based on complex type strategy
- PHP void <-> empty type.
- If type is not matched to any of these types by some reason, then *xsd:anyType* is used.

Where *xsd:* is "http://www.w3.org/2001/XMLSchema" namespace, *soap-enc:* is a "http://schemas.xmlsoap.org/soap/encoding/" namespace, *tns:* is a "target namespace" for a service.

241.5 WSDL Binding Styles

WSDL offers different transport mechanisms and styles. This affects the *soap:binding* and *soap:body* tags within the Binding section of WSDL. Different clients have different requirements as to what options really work. Therefore you can set the styles before you call any *setClass* or *addFunction* method on the AutoDiscover class.

```
$autodiscover = new Zend\Soap\AutoDiscover();
// Default is 'use' => 'encoded' and
// 'encodingStyle' => 'http://schemas.xmlsoap.org/soap/encoding/'
$autodiscover->setOperationBodyStyle(
                    array('use' => 'literal',
                          'namespace' => 'http://framework.zend.com')
```

[1] `Zend\Soap\AutoDiscover` will be created with the `Zend\Soap\Wsdl\ComplexTypeStrategy\DefaultComplexType` class as detection algorithm for complex types. The first parameter of the AutoDiscover constructor takes any complex type strategy implementing `Zend\Soap\Wsdl\ComplexTypeStrategy\ComplexTypeStrategyInterface` or a string with the name of the class. See the *Zend\Soap\Wsdl manual on adding complex* types for more information.

```
                 );

// Default is 'style' => 'rpc' and
// 'transport' => 'http://schemas.xmlsoap.org/soap/http'
$autodiscover->setBindingStyle(
                    array('style' => 'document',
                          'transport' => 'http://framework.zend.com')
                 );
...
$autodiscover->addFunction('myfunc1');
$wsdl = $autodiscover->generate();
```

CHAPTER

TWO

ZEND\STDLIB\HYDRATOR

Hydration is the act of populating an object from a set of data.

The `Hydrator` is a simple component to provide mechanisms both for hydrating objects, as well as extracting data sets from them.

The component consists of an interface, and several implementations for common use cases.

242.1 HydratorInterface

```
namespace Zend\Stdlib\Hydrator;

interface HydratorInterface
{
    /**
     * Extract values from an object
     *
     * @param  object $object
     * @return array
     */
    public function extract($object);

    /**
     * Hydrate $object with the provided $data.
     *
     * @param  array $data
     * @param  object $object
     * @return void
     */
    public function hydrate(array $data, $object);
}
```

242.2 Usage

Usage is quite simple: simply instantiate the hydrator, and then pass information to it.

```
use Zend\Stdlib\Hydrator;
$hydrator = new Hydrator\ArraySerializable();

$object = new ArrayObject(array());

```

```
$hydrator->hydrate($someData, $object);

// or, if the object has data we want as an array:
$data = $hydrator->extract($object);
```

242.3 Available Implementations

- **Zend\Stdlib\Hydrator\ArraySerializable**

  Follows the definition of `ArrayObject`. Objects must implement either the `exchangeArray()` or `populate()` methods to support hydration, and the `getArrayCopy()` method to support extraction.

- **Zend\Stdlib\Hydrator\ClassMethods**

  Any data key matching a setter method will be called in order to hydrate; any method matching a getter method will be called for extraction.

- **Zend\Stdlib\Hydrator\ObjectProperty**

  Any data key matching a publicly accessible property will be hydrated; any public properties will be used for extraction.

CHAPTER

THREE

ZEND\STDLIB\HYDRATOR\FILTER

The hydrator filters, allows you to manipulate the behavior, when you want to `extract()` your stuff to arrays. This is especially useful, if you want to `extract()` your objects to the userland and strip some internals (e.g. `getServiceManager()`).

It comes with a helpful Composite Implementation and a few filters for common use cases. The filters are implemented on the `AbstractHydrator`, so you can directly start using them if you extend it - even on custom hydrators.

```
namespace Zend\Stdlib\Hydrator\Filter;

interface FilterInterface
{
    /**
     * Should return true, if the given filter
     * does not match
     *
     * @param string $property The name of the property
     * @return bool
     */
    public function filter($property);
}
```

If it returns true, the key / value pairs will be in the extracted arrays - if it will return false, you'll not see them again.

243.1 Filter implementations

- **Zend\Stdlib\Hydrator\Filter\GetFilter**

  This filter is used in the `ClassMethods` hydrator, to decide that getters will be extracted. It checks, if the key that should be extracted starts with `get` or looks like this `Zend\Foo\Bar::getFoo`

- **Zend\Stdlib\Hydrator\Filter\HasFilter**

  This filter is used in the `ClassMethods` hydrator, to decide that `has` methods will be extracted. It checks, if the key that should be extracted starts with `has` or looks like this `Zend\Foo\Bar::hasFoo`

- **Zend\Stdlib\Hydrator\Filter\IsFilter**

  This filter is used in the `ClassMethods` hydrator, to decide that `is` methods will be extracted. It checks, if the key that should be extracted starts with `is` or looks like this `Zend\Foo\Bar::isFoo`

- **Zend\Stdlib\Hydrator\Filter\MethodMatchFilter**

  This filter allows you to strip methods from the extraction with the correct condition in the composite. It checks, if the key that should be extracted matches a method name. Either `getServiceLocator` or

`Zend\Foo::getServicelocator`. The name of the method is specified in the constructor of this filter. The 2nd parameter decides whether to use white or blacklisting to decide. Default is blacklisting - pass `false` to change it.

- **Zend\Stdlib\Hydrator\Filter\NumberOfParameterFilter**

  This filter is used in the `ClassMethods` hydrator, to check the number of parameters. By convention, the `get`, `has` and `is` methods do not get any parameters - but it may happen. You can add your own number of needed parameters, simply add the number to the constructor. The default value is 0

243.2 Remove filters

If you want to tell e.g. the `ClassMethods` hydrator, to not extract methods that start with `is`, you can do so:

```
$hydrator = new ClassMethods(false);
$hydrator->removeFilter("is");
```

The key / value pairs for `is` methods will not end up in your extracted array anymore. The filters can be used in any hydrator, but the `ClassMethods` hydrator is the only one, that has pre-registered filters:

```
$this->filterComposite->addFilter("is", new IsFilter());
$this->filterComposite->addFilter("has", new HasFilter());
$this->filterComposite->addFilter("get", new GetFilter());
$this->filterComposite->addFilter("parameter", new NumberOfParameterFilter(), FilterComposite::CONDI
```

If you're not fine with this, you can unregister them as above.

Note: The parameter for the filter on the `ClassMethods` looks like this by default `Zend\Foo\Bar::methodName`

243.3 Add filters

You can easily add filters to any hydrator, that extends the `AbstractHydrator`. You can use the `FilterInterface` or any callable:

```
$hydrator->addFilter("len", function($property) {
    if (strlen($property) !== 3) {
        return false;
    }
    return true;
});
```

By default, every filter you add will be added with a conditional `or`. If you want to add it with `and` (as the `NumberOfParameterFilter` that is added to the `ClassMethods` hydrator by default) you can do that too:

```
$hydrator->addFilter("len", function($property) {
    if (strlen($property) !== 3) {
        return false;
    }
    return true;
}, FilterComposite::CONDITION_AND);
```

Or you can add the shipped ones:

```
$hydrator->addFilter(
  "servicemanager",
  new MethodMatchFilter("getServiceManager"),
  FilterComposite::CONDITION_AND
);
```

The example above will exclude the `getServiceManager` method or the key from the extraction, even if the `get` filter wants to add it.

243.4 Use the composite for complex filters

The composite implements the `FilterInterface` too, so you can add it as a regular filter to the hydrator. One goody of this implementation, is that you can add the filters with a condition and you can do even more complex stuff with different composites with different conditions. You can pass the condition to the 3rd parameter, when you add a filter:

Zend\Stdlib\Hydrator\Filter\FilterComposite::CONDITION_OR

At one level of the composite, one of all filters in that condition block has to return true in order to get extracted

Zend\Stdlib\Hydrator\Filter\FilterComposite::CONDITION_AND

At one level of the composite, all of the filters in that condition block has to return `true` in order to get extracted

This composition will have a similar logic as the if below:

```
 $composite = new FilterComposite();

 $composite->addFilter("one", $condition1);
 $composite->addFilter("two", $condition2);
 $composite->addFilter("three", $condition3);
 $composite->addFilter("four", $condition4, FilterComposite::CONDITION_AND);
 $composite->addFilter("five", $condition5, FilterComposite::CONDITION_AND);

 // This is what's happening internally
 if (
      (
         $condition1
         || $condition2
         || $condition3
      ) && (
         $condition4
         && $condition5
      )
  ) {
 //do extraction
 }
```

If you've only one condition (only `and` or `or`) block, the other one will be completely ignored.

A bit more complex filter can look like this:

```
 $composite = new FilterComposite();
 $composite->addFilter(
     "servicemanager",
     new MethodMatchFilter("getServiceManager"),
```

```
    FilterComposite::CONDITION_AND
);
$composite->addFilter(
    "eventmanager",
    new MethodMatchFilter("getEventManager"),
    FilterComposite::CONDITION_AND
);

$hydrator->addFilter("excludes", $composite, FilterComposite::CONDITION_AND);

// Internal
if (
      (  // default composite inside the hydrator
         (
             $getFilter
             || $hasFilter
             || $isFilter
          ) && (
             $numberOfParameterFilter
          )
      ) && (  // new composite, added to the one above
         $serviceManagerFilter
         && $eventManagerFilter
      )
) {
// do extraction
}
```

If you perform this on the `ClassMethods` hydrator, all getters will get extracted, but not `getServiceManager` and `getEventManager`.

243.5 Using the provider interface

There is also a provider interface, that allows you to configure the behavior of the hydrator inside your objects.

```
namespace Zend\Stdlib\Hydrator\Filter;

interface FilterProviderInterface
{
    /**
     * Provides a filter for hydration
     *
     * @return FilterInterface
     */
    public function getFilter();
}
```

The `getFilter()` method is getting automatically excluded from `extract()`. If the extracted object implements the `Zend\Stdlib\Hydrator\Filter\FilterProviderInterface`, the returned `FilterInterface` instance can also be a `FilterComposite`.

For example:

```
Class Foo implements FilterProviderInterface
{
    public function getFoo()
    {
```

```
        return "foo";
    }

    public function hasFoo()
    {
        return true;
    }

    public function getServiceManager()
    {
        return "servicemanager";
    }

    public function getEventManager()
    {
        return "eventmanager";
    }

    public function getFilter()
    {
        $composite = new FilterComposite();
        $composite->addFilter("get", new GetFilter());

        $exclusionComposite = new FilterComposite();
        $exclusionComposite->addFilter(
            "servicemanager",
            new MethodMatchFilter("getServiceManager"),
            FilterComposite::CONDITION_AND
            );
        $exclusionComposite->addFilter(
            "eventmanager",
            new MethodMatchFilter("getEventManager"),
            FilterComposite::CONDITION_AND
        );

        $composite->addFilter("excludes", $exclusionComposite, FilterComposite::CONDITION_AND);

        return $composite;
    }
}

$hydrator = new ClassMethods(false);
$extractedArray = $hydrator->extract(new Foo());
```

The `$extractedArray` does only have "foo" => "foo" in. All of the others are excluded from the extraction.

Note: All pre-registered filters from the `ClassMethods` hydrator are ignored if this interface is used.

CHAPTER

FOUR

ZEND\STDLIB\HYDRATOR\STRATEGY

You can add a `Zend\Stdlib\Hydrator\Strategy\StrategyInterface` to any of the hydrators (expect it extends `Zend\Stdlib\Hydrator\AbstractHydrator` or implements `Zend\Stdlib\Hydrator\HydratorInterface` and `Zend\Stdlib\Hydrator\Strategy\StrategyEnabledInte` to manipulate the way how they behave on `extract()` and `hydrate()` for specific key / value pairs. This is the interface that needs to be implemented:

```
namespace Zend\Stdlib\Hydrator\Strategy;

interface StrategyInterface
{
     /**
      * Converts the given value so that it can be extracted by the hydrator.
      *
      * @param mixed $value The original value.
      * @return mixed Returns the value that should be extracted.
      */
     public function extract($value);
     /**
      * Converts the given value so that it can be hydrated by the hydrator.
      *
      * @param mixed $value The original value.
      * @return mixed Returns the value that should be hydrated.
      */
     public function hydrate($value);
}
```

As you can see, this interface is similar to `Zend\Stdlib\Hydrator\HydratorInterface`. The reason why is, that the strategies provide a proxy implementation for `hydrate()` and `extract()`.

244.1 Adding strategies to the hydrators

To allow strategies within your hydrator, the `Zend\Stdlib\Hydrator\Strategy\StrategyEnabledInterface` provide the following methods:

```
namespace Zend\Stdlib\Hydrator;

use Zend\Stdlib\Hydrator\Strategy\StrategyInterface;

interface StrategyEnabledInterface
{
    /**
```

```
     * Adds the given strategy under the given name.
     *
     * @param string $name The name of the strategy to register.
     * @param StrategyInterface $strategy The strategy to register.
     * @return HydratorInterface
     */
    public function addStrategy($name, StrategyInterface $strategy);

    /**
     * Gets the strategy with the given name.
     *
     * @param string $name The name of the strategy to get.
     * @return StrategyInterface
     */
    public function getStrategy($name);

    /**
     * Checks if the strategy with the given name exists.
     *
     * @param string $name The name of the strategy to check for.
     * @return bool
     */
    public function hasStrategy($name);

    /**
     * Removes the strategy with the given name.
     *
     * @param string $name The name of the strategy to remove.
     * @return HydratorInterface
     */
    public function removeStrategy($name);
}
```

Every hydrator, that is shipped by default, provides this functionality. The `AbstractHydrator` has it fully functional implemented. If you want to use this functionality in your own hydrators, you should extend the `AbstractHydrator`.

244.2 Available implementations

- **Zend\Stdlib\Hydrator\Strategy\SerializableStrategy**

  This is the strategy, that provides the functionality for `Zend\Stdlib\Hydrator\ArraySerializable`. You can use it with custom implementations for `Zend\Serializer\Adapter\AdapterInterface` if you want to.

- **Zend\Stdlib\Hydrator\Strategy\DefaultStrategy**

  This is a kind of dummy-implementation, that simply proxies everything through, without doing anything on the parameters.

244.3 Writing custom strategies

As usual, this is not really a very useful example, but will give you a good point about how to start with writing your own strategies and where to use them. This strategy simply transform the value for the defined key to rot13 on `extract()` and back on `hydrate()`:

```
class Rot13Strategy implements StrategyInterface
{
    public function extract($value)
    {
        return str_rot13($value);
    }

    public function hydrate($value)
    {
        return str_rot13($value);
    }
}
```

This is the example class, we want to use for the hydrator example:

```
class Foo
{
    protected $foo = null;
    protected $bar = null;

    public function getFoo()
    {
        return $this->foo;
    }

    public function setFoo($foo)
    {
        $this->foo = $foo;
    }

    public function getBar()
    {
        return $this->bar;
    }

    public function setBar($bar)
    {
        $this->bar = $bar;
    }
}
```

Now, we want to add the rot13 strategy to the method `getFoo()` and `setFoo($foo)`:

```
$foo = new Foo();
$foo->setFoo("bar");
$foo->setBar("foo");

$hydrator = new ClassMethods();
$hydrator->addStrategy("foo", new Rot13Strategy());
```

When you now use the hydrator, to get an array of the object $foo, this is the array you'll get:

```
$extractedArray = $hydrator->extract($foo);

// array(2) {
//     ["foo"]=>
//     string(3) "one"
//     ["bar"]=>
//     string(3) "foo"
```

```
// }
```

And the the way back:

```
$hydrator->hydrate($extractedArray, $foo)

// object(Foo)#2 (2) {
//   ["foo":protected]=>
//   string(3) "bar"
//   ["bar":protected]=>
//   string(3) "foo"
// }
```

CHAPTER

FIVE

INTRODUCTION TO ZEND\TAG

`Zend\Tag` is a component suite which provides a facility to work with taggable Items. As its base, it provides two classes to work with Tags, `Zend\Tag\Item` and `Zend\Tag\ItemList`. Additionally, it comes with the interface `Zend\Tag\TaggableInterface`, which allows you to use any of your models as a taggable item in conjunction with `Zend\Tag`.

`Zend\Tag\Item` is a basic taggable item implementation which comes with the essential functionality required to work with the `Zend\Tag` suite. A taggable item always consists of a title and a relative weight (e.g. number of occurrences). It also stores parameters which are used by the different sub-components of `Zend\Tag`.

To group multiple items together, `Zend\Tag\ItemList` exists as an array iterator and provides additional functionality to calculate absolute weight values based on the given relative weights of each item in it.

Using Zend\Tag

This example illustrates how to create a list of tags and spread absolute weight values on them.

```
// Create the item list
$list = new Zend\Tag\ItemList();

// Assign tags to it
$list[] = new Zend\Tag\Item(array('title' => 'Code', 'weight' => 50));
$list[] = new Zend\Tag\Item(array('title' => 'Zend Framework', 'weight' => 1));
$list[] = new Zend\Tag\Item(array('title' => 'PHP', 'weight' => 5));

// Spread absolute values on the items
$list->spreadWeightValues(array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

// Output the items with their absolute values
foreach ($list as $item) {
    printf("%s: %d\n", $item->getTitle(), $item->getParam('weightValue'));
}
```

This will output the three items Code, Zend Framework and *PHP* with the absolute values 10, 1 and 2.

CHAPTER

SIX

CREATING TAG CLOUDS WITH ZEND\TAG\CLOUD

`Zend\Tag\Cloud` is the rendering part of `Zend\Tag`. By default it comes with a set of *HTML* decorators, which allow you to create tag clouds for a website, but also supplies you with two abstract classes to create your own decorators, to create tag clouds in *PDF* documents for example.

You can instantiate and configure `Zend\Tag\Cloud` either programmatically or completely via an array or an instance of `Traversable`. The available options are:

Table 246.1: Zend\Tag\Cloud Options

Option	Description
`cloudDecorator`	Defines the decorator for the cloud. Can either be the name of the class which should be loaded by the plugin manager, an instance of `Zend\Tag\Cloud\Decorator\AbstractCloud` or an array containing the decorator under the key `decorator` and optionally an array under the key `options`, which will be passed to the decorator's constructor.
`tagDecorator`	Defines the decorator for individual tags. This can either be the name of the class which should be loaded by the plugin manager, an instance of `Zend\Tag\Cloud\Decorator\AbstractTag` or an array containing the decorator under the key `decorator` and optionally an array under the key `options`, which will be passed to the decorator's constructor.
`decoratorPluginManager`	A different plugin manager to use. Must be an instance of `Zend\ServiceManager\AbstractPluginManager`.
`itemList`	A different item list to use. Must be an instance of `Zend\Tag\ItemList`.
`tags`	A array of tags to assign to the cloud. Each tag must either implement `Zend\Tag\TaggableInterface` or be an array which can be used to instantiate `Zend\Tag\Item`.

Using Zend\Tag\Cloud

This example illustrates a basic example of how to create a tag cloud, add multiple tags to it and finally render it.

```
// Create the cloud and assign static tags to it
$cloud = new Zend\Tag\Cloud(array(
    'tags' => array(
        array(
            'title'  => 'Code',
            'weight' => 50,
            'params' => array('url' => '/tag/code'),
```

```
        ),
        array(
            'title'  => 'Zend Framework',
            'weight' => 1,
            'params' => array('url' => '/tag/zend-framework'),
        ),
        array(
            'title' => 'PHP',
            'weight' => 5,
            'params' => array('url' => '/tag/php'),
        ),
    ),
));

// Render the cloud
echo $cloud;
```

This will output the tag cloud with the three tags, spread with the default font-sizes.

The following example shows how create a tag cloud from a `Zend\Config\Config` object.

```
# An example tags.ini file
tags.1.title = "Code"
tags.1.weight = 50
tags.1.params.url = "/tag/code"
tags.2.title = "Zend Framework"
tags.2.weight = 1
tags.2.params.url = "/tag/zend-framework"
tags.3.title = "PHP"
tags.3.weight = 2
tags.3.params.url = "/tag/php"
```

```
// Create the cloud from a Zend\Config\Config object
$config = Zend\Config\Factory::fromFile('tags.ini');
$cloud = new Zend\Tag\Cloud($config);

// Render the cloud
echo $cloud;
```

246.1 Decorators

`Zend\Tag\Cloud` requires two types of decorators to be able to render a tag cloud. This includes a decorator which renders the single tags as well as a decorator which renders the surrounding cloud. `Zend\Tag\Cloud` ships a default decorator set for formatting a tag cloud in *HTML*. This set will, by default, create a tag cloud as ul/li -list, spread with different font-sizes according to the weight values of the tags assigned to them.

246.1.1 HTML Tag decorator

The *HTML* tag decorator will by default render every tag in an anchor element, surrounded by a `<li>` element. The anchor itself is fixed and cannot be changed, but the surrounding element(s) can.

Note: URL parameter

As the *HTML* tag decorator always surounds the tag title with an anchor, you should define a *URL* parameter for every tag used in it.

The tag decorator can either spread different font-sizes over the anchors or a defined list of classnames. When setting options for one of those possibilities, the corresponding one will automatically be enabled. The following configuration options are available:

Table 246.2: HTML Tag decorator Options

Option	Default	Description
`fontSizeUnit`	`px`	Defines the font-size unit used for all font-sizes. The possible values are: em, ex, px, in, cm, mm, pt, pc and %.
`minFontSize`	`10`	The minimum font-size distributed through the tags (must be numeric).
`maxFontSize`	`20`	The maximum font-size distributed through the tags (must be numeric).
`classList`	`null`	An array of classes distributed through the tags.
`htmlTags`	`array('li')`	An array of *HTML* tags surrounding the anchor. Each element can either be a string, which is used as element type, or an array containing an attribute list for the element, defined as key/value pair. In this case, the array key is used as element type.

The following example shows how to create a tag cloud with a customized *HTML* tag decorator.

```
$cloud = new Zend\Tag\Cloud(array(
    'tagDecorator' => array(
        'decorator' => 'htmltag',
        'options'   => array(
            'minFontSize' => '20',
            'maxFontSize' => '50',
            'htmlTags'    => array(
                'li' => array('class' => 'my_custom_class'),
            ),
        ),
    ),
    'tags' => array(
       array(
           'title'  => 'Code',
           'weight' => 50,
           'params' => array('url' => '/tag/code'),
       ),
       array(
           'title'  => 'Zend Framework',
           'weight' => 1,
           'params' => array('url' => '/tag/zend-framework'),
       ),
       array(
           'title'  => 'PHP',
           'weight' => 5,
           'params' => array('url' => '/tag/php')
       ),
   ),
));

// Render the cloud
echo $cloud;
```

246.1.2 HTML Cloud decorator

By default the *HTML* cloud decorator will surround the *HTML* tags with a `<ul>` element and add no separation. Like in the tag decorator, you can define multiple surrounding *HTML* tags and additionally define a separator. The available

options are:

Table 246.3: HTML Cloud decorator Options

Option	Default	Description
`separator`	*' ' (a whitespace)*	Defines the separator which is placed between all tags.
`htmlTags`	`array('ul' => array('class' => 'Zend\Tag\Cloud'))`	An array of *HTML* tags surrounding all tags. Each element can either be a string, which is used as element type, or an array containing an attribute list for the element, defined as key/value pair. In this case, the array key is used as element type.

CHAPTER

SEVEN

INTRODUCTION TO ZEND\TEST

The `Zend\Test` component provides tools to facilitate unit testing of your Zend Framework applications. At this time, we offer facilities to enable testing of your Zend Framework MVC applications.

PHPUnit is the only library supported currently.

CHAPTER

EIGHT

UNIT TESTING WITH PHPUNIT

`Zend\Test\PHPUnit` provides a TestCase for MVC applications that contains assertions for testing against a variety of responsibilities. Probably the easiest way to understand what it can do is to see an example.

The following is a simple test case for a IndexController to verify things like HTTP code, controller and action name :

```
<?php

namespace ApplicationTest\Controller;

use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;

class IndexControllerTest extends AbstractHttpControllerTestCase
{
    public function setUp()
    {
        $this->setApplicationConfig(
            include '/path/to/application/config/test/application.config.php'
        );
        parent::setUp();
    }

    public function testIndexActionCanBeAccessed()
    {
        $this->dispatch('/');
        $this->assertResponseStatusCode(200);

        $this->assertModuleName('application');
        $this->assertControllerName('application_index');
        $this->assertControllerClass('IndexController');
        $this->assertMatchedRouteName('home');
    }
}
```

The setup of the test case can to define the application config. You can use several config to test modules dependencies or your current application config.

CHAPTER

NINE

SETUP YOUR TESTCASE

As noted in the previous example, all MVC test cases should extend AbstractHttpControllerTestCase. This class in turn extends `PHPUnit_Framework_TestCase`, and gives you all the structure and assertions you'd expect from PHPUnit – as well as some scaffolding and assertions specific to Zend Framework's MVC implementation.

In order to test your MVC application, you will need to setup the application config. Use simply the the `setApplicationConfig` method :

```
public function setUp()
{
    $this->setApplicationConfig(
        include '/path/to/application/config/test/application.config.php'
    );
    parent::setUp();
}
```

Once the application is set up, you can write your tests. To help debug tests, you can activate the flag `traceError` to throw MVC exception during the tests writing :

```
<?php

namespace ApplicationTest\Controller;

use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;

class IndexControllerTest extends AbstractHttpControllerTestCase
{
    protected $traceError = true;
}
```

CHAPTER

TESTING YOUR CONTROLLERS AND MVC APPLICATIONS

Once you have your application config in place, you can begin testing. Testing is basically as you would expect in an PHPUnit test suite, with a few minor differences.

First, you will need to dispatch a URL to test, using the `dispatch` method of the TestCase:

```
public function testIndexAction()
{
    $this->dispatch('/');
}
```

There will be times, however, that you need to provide extra information – GET and POST variables, COOKIE information, etc. You can populate the request with that information:

```
public function testIndexAction()
{
    $this->getRequest()
        ->setMethod('POST')
        ->setPost(new Parameters(array('argument' => 'value')));
    $this->dispatch('/');
}
```

You can populate GET or POST variables directly with the `dispatch` method :

```
public function testIndexAction()
{
    $this->dispatch('/', 'POST', array('argument' => 'value'));
}
```

You can use directly yours query args in the url :

```
public function testIndexAction()
{
    $this->dispatch('/tests?foo=bar&baz=foo');
}
```

Now that the request is made, it's time to start making assertions against it.

CHAPTER

ONE

ASSERTIONS

Assertions are at the heart of Unit Testing; you use them to verify that the results are what you expect. To this end, `Zend\Test\PHPUnit\AbstractControllerTestCase` provides a number of assertions to make testing your MVC apps and controllers simpler.

CHAPTER

TWO

REQUEST ASSERTIONS

It's often useful to assert against the last run action, controller, and module; additionally, you may want to assert against the route that was matched. The following assertions can help you in this regard:

- `assertModulesLoaded(array $modules)`: Assert that the given modules was loaded by the application.
- `assertModuleName($module)`: Assert that the given module was used in the last dispatched action.
- `assertControllerName($controller)`: Assert that the given controller identifier was selected in the last dispatched action.
- `assertControllerClass($controller)`: Assert that the given controller class was selected in the last dispatched action.
- `assertActionName($action)`: Assert that the given action was last dispatched.
- `assertMatchedRouteName($route)`: Assert that the given named route was matched by the router.

Each also has a 'Not' variant for negative assertions.

CHAPTER

THREE

CSS SELECTOR ASSERTIONS

CSS selectors are an easy way to verify that certain artifacts are present in the response content. They also make it trivial to ensure that items necessary for Javascript UIs and/or AJAX integration will be present; most JS toolkits provide some mechanism for pulling DOM elements based on CSS selectors, so the syntax would be the same.

This functionality is provided via `Zend\Dom\Query`, and integrated into a set of 'Query' assertions. Each of these assertions takes as their first argument a CSS selector, with optionally additional arguments and/or an error message, based on the assertion type. You can find the rules for writing the CSS selectors in the `Zend\Dom\Query` *Theory of Operation* chapter. Query assertions include:

- `assertQuery($path)`: assert that one or more DOM elements matching the given CSS selector are present.
- `assertQueryContentContains($path, $match)`: assert that one or more DOM elements matching the given CSS selector are present,

and that at least one contains the content provided in $match.

- `assertQueryContentRegex($path, $pattern)`: assert that one or more DOM elements matching the given CSS selector are present,

and that at least one matches the regular expression provided in $pattern. If a $message is present, it will be prepended to any failed assertion message.

- `assertQueryCount($path, $count)`: assert that there are exactly $count DOM elements matching the given CSS selector present.
- `assertQueryCountMin($path, $count)`: assert that there are at least $count DOM elements matching the given CSS selector present.
- `assertQueryCountMax($path, $count)`: assert that there are no more than $count DOM elements matching the given CSS selector present.

Additionally, each of the above has a 'Not' variant that provides a negative assertion: assertNotQuery(), assertNotQueryContentContains(), assertNotQueryContentRegex(), and assertNotQueryCount(). (Note that the min and max counts do not have these variants, for what should be obvious reasons.)

CHAPTER

FOUR

XPATH ASSERTIONS

Some developers are more familiar with XPath than with CSS selectors, and thus XPath variants of all the Query assertions are also provided. These are:

- `assertXpathQuery($path)`
- `assertNotXpathQuery($path)`
- `assertXpathQueryCount($path, $count)`
- `assertNotXpathQueryCount($path, $count)`
- `assertXpathQueryCountMin($path, $count)`
- `assertXpathQueryCountMax($path, $count)`
- `assertXpathQueryContentContains($path, $match)`
- `assertNotXpathQueryContentContains($path, $match)`
- `assertXpathQueryContentRegex($path, $pattern)`
- `assertNotXpathQueryContentRegex($path, $pattern)`

CHAPTER

FIVE

REDIRECT ASSERTIONS

Often an action will redirect. Instead of following the redirect, `Zend\Test\PHPUnit\ControllerTestCase` allows you to test for redirects with a handful of assertions.

- `assertRedirect()`: assert simply that a redirect has occurred.
- `assertRedirectTo($url)`: assert that a redirect has occurred, and that the value of the Location header is the $url provided.
- `assertRedirectRegex($pattern)`: assert that a redirect has occurred, and that the value of the Location header matches the regular

expression provided by $pattern.

Each also has a 'Not' variant for negative assertions.

CHAPTER

SIX

RESPONSE HEADER ASSERTIONS

In addition to checking for redirect headers, you will often need to check for specific HTTP response codes and headers – for instance, to determine whether an action results in a 404 or 500 response, or to ensure that JSON responses contain the appropriate Content-Type header. The following assertions are available.

- `assertResponseCode($code)`: assert that the response resulted in the given HTTP response code.
- `assertResponseHeader($header)`: assert that the response contains the given header.
- `assertResponseHeaderContains($header, $match)`: assert that the response contains the given header and that its content contains the given string.
- `assertResponseHeaderRegex($header, $pattern)`: assert that the response contains the given header and that its content matches the given regex.

Additionally, each of the above assertions have a 'Not' variant for negative assertions.

CHAPTER

SEVEN

ZEND\TEXT\FIGLET

`Zend\Text\Figlet` is a component which enables developers to create a so called FIGlet text. A FIGlet text is a string, which is represented as *ASCII* art. FIGlets use a special font format, called FLT (FigLet Font). By default, one standard font is shipped with `Zend\Text\Figlet`, but you can download additional fonts at http://www.figlet.org.

Note: Compressed fonts

`Zend\Text\Figlet` supports gzipped fonts. This means that you can take an `.flf` file and gzip it. To allow `Zend\Text\Figlet` to recognize this, the gzipped font must have the extension `.gz`. Further, to be able to use gzipped fonts, you have to have enabled the GZIP extension of *PHP*.

Note: Encoding

`Zend\Text\Figlet` expects your strings to be UTF-8 encoded by default. If this is not the case, you can supply the character encoding as second parameter to the `render()` method.

You can define multiple options for a FIGlet. When instantiating `Zend\Text\Figlet\Figlet`, you can supply an array or an instance of `Zend\Config`.

- `font`- Defines the font which should be used for rendering. If not defines, the built-in font will be used.
- `outputWidth`- Defines the maximum width of the output string. This is used for word-wrap as well as justification. Beware of too small values, they may result in an undefined behaviour. The default value is 80.
- `handleParagraphs`- A boolean which indicates, how new lines are handled. When set to `TRUE`, single new lines are ignored and instead treated as single spaces. Only multiple new lines will be handled as such. The default value is `FALSE`.
- `justification`- May be one of the values of `Zend\Text\Figlet\Figlet::JUSTIFICATION_*`. There is `JUSTIFICATION_LEFT`, `JUSTIFICATION_CENTER` and `JUSTIFICATION_RIGHT` The default justification is defined by the `rightToLeft` value.
- `rightToLeft`- Defines in which direction the text is written. May be either `Zend\Text\Figlet\Figlet::DIRECTION_LEFT_TO_RIGHT` or `Zend\Text\Figlet\Figlet::DIRECTION_RIGHT_TO_LEFT`. By default the setting of the font file is used. When justification is not defined, a text written from right-to-left is automatically right-aligned.
- `smushMode`- An integer bitfield which defines, how the single characters are smushed together. Can be the sum of multiple values from `Zend\Text\Figlet\Figlet::SM_*`. There are the following smush modes: `SM_EQUAL`, `SM_LOWLINE`, `SM_HIERARCHY`, `SM_PAIR`, `SM_BIGX`, `SM_HARDBLANK`, `SM_KERN` and `SM_SMUSH`. A value of 0 doesn't disable the entire smushing, but forces SM_KERN to be applied, while a value of -1 disables it. An explanation of the different smush modes can be found here. By default the setting

of the font file is used. The smush mode option is normally used only by font designers testing the various layoutmodes with a new font.

Using Zend\Text\Figlet

This example illustrates the basic use of `Zend\Text\Figlet` to create a simple FIGlet text:

```
$figlet = new Zend\Text\Figlet\Figlet();
echo $figlet->render('Zend');
```

Assuming you are using a monospace font, this would look as follows:

```
  _______     _______     _  __    _______
 |__  //    |  ___||   | \| ||  |   __ \\
   / //     | ||__     |  ' ||  | |  \ ||
  / //__    | ||___    | . ||  | |__/ ||
 /______||  |______||  |_|\_||  |______//
 `------'   `------'   `_` _`'   ------`
```

CHAPTER

EIGHT

ZEND\TEXT\TABLE

`Zend\Text\Table` is a component to create text based tables on the fly with different decorators. This can be helpful, if you either want to send structured data in text emails, which are used to have mono-spaced fonts, or to display table information in a CLI application. `Zend\Text\Table` supports multi-line columns, colspan and align as well.

Note: Encoding

`Zend\Text\Table` expects your strings to be UTF-8 encoded by default. If this is not the case, you can either supply the character encoding as a parameter to the `constructor()` or the `setContent()` method of `Zend\Text\Table\Column`. Alternatively if you have a different encoding in the entire process, you can define the standard input charset with `Zend\Text\Table\Table::setInputCharset($charset)`. In case you need another output charset for the table, you can set this with `Zend\Text\Table\Table::setOutputCharset($charset)`.

A `Zend\Text\Table\Table` object consists of rows, which contain columns, represented by `Zend\Text\Table\Row` and `Zend\Text\Table\Column`. When creating a table, you can supply an array with options for the table. Those are:

- `columnWidths` (required): An array defining all columns width their widths in characters.
- `decorator`: The decorator to use for the table borders. The default is **unicode**, but you may also specify **ascii** or give an instance of a custom decorator object.
- `padding`: The left and right padding withing the columns in characters. The default padding is zero.
- `AutoSeparate`: The way how the rows are separated with horizontal lines. The default is a separation between all rows. This is defined as a bitmask containing one ore more of the following constants of `Zend\Text\Table`:
  - `Zend\Text\Table\Table::AUTO_SEPARATE_NONE`
  - `Zend\Text\Table\Table::AUTO_SEPARATE_HEADER`
  - `Zend\Text\Table\Table::AUTO_SEPARATE_FOOTER`
  - `Zend\Text\Table\Table::AUTO_SEPARATE_ALL`

  Where header is always the first row, and the footer is always the last row.

Rows are simply added to the table by creating a new instance of `Zend\Text\Table\Row`, and appending it to the table via the `appendRow()` method. Rows themselves have no options. You can also give an array to directly to the `appendRow()` method, which then will automatically converted to a row object, containing multiple column objects.

The same way you can add columns to the rows. Create a new instance of `Zend\Text\Table\Column` and then either set the column options in the constructor or later with the `set*()` methods. The first parameter is the content of the column which may have multiple lines, which in the best case are separated by just the '\n' character. The second parameter defines the align, which is 'left' by default and can be one of the class constants of `Zend\Text\Table\Column`:

- `ALIGN_LEFT`
- `ALIGN_CENTER`
- `ALIGN_RIGHT`

The third parameter is the colspan of the column. For example, when you choose "2" as colspan, the column will span over two columns of the table. The last parameter defines the encoding of the content, which should be supplied, if the content is neither ASCII nor UTF-8. To append the column to the row, you simply call `appendColumn()` in your row object with the column object as parameter. Alternatively you can directly give a string to the `appendColumn()` method.

To finally render the table, you can either use the `render()` method of the table, or use the magic method `__toString()` by doing `echo $table;` or `$tableString = (string) $table`.

Using Zend\Text\Table

This example illustrates the basic use of `Zend\Text\Table` to create a simple table:

```
$table = new Zend\Text\Table\Table(array('columnWidths' => array(10, 20)));

// Either simple
$table->appendRow(array('Zend', 'Framework'));

// Or verbose
$row = new Zend\Text\Table\Row();

$row->appendColumn(new Zend\Text\Table\Column('Zend'));
$row->appendColumn(new Zend\Text\Table\Column('Framework'));

$table->appendRow($row);

echo $table;
```

This will result in the following output:

```
------------------------------
|Zend      |Framework          |
|----------|-------------------|
|Zend      |Framework          |
------------------------------
```

CHAPTER

NINE

ZEND\URI

259.1 Overview

`Zend\Uri` is a component that aids in manipulating and validating Uniform Resource Identifiers (URIs) [1]. `Zend\Uri` exists primarily to service other components, such as `Zend\Http\`, but is also useful as a standalone utility.

*URI*s always begin with a scheme, followed by a colon. The construction of the many different schemes varies significantly. The `Zend\Uri` component provides the `Zend\Uri\UriFactory` that returns a class implementing the `Zend\Uri\UriInterface` which specializes in the scheme if such a class is registered with the Factory.

259.2 Creating a New URI

`Zend\Uri\UriFactory` will build a new *URI* from scratch if only a scheme is passed to `Zend\Uri\UriFactory::factory()`.

Creating a New URI with ZendUriUriFactory::factory()

```
// To create a new URI from scratch, pass only the scheme
// followed by a colon.
$uri = Zend\Uri\UriFactory::factory('http:');

// $uri instanceof Zend\Uri\UriInterface
```

To create a new *URI* from scratch, pass only the scheme followed by a colon to `Zend\Uri\UriFactory::factory()` [2]. If an unsupported scheme is passed and no scheme-specific class is specified, a `Zend\Uri\Exception\InvalidArgumentException` will be thrown.

If the scheme or *URI* passed is supported, `Zend\Uri\UriFactory::factory()` will return a class implementing `Zend\Uri\UriInterface` that specializes in the scheme to be created.

259.2.1 Creating a New Custom-Class URI

You can specify a custom class to be used when using the `Zend\Uri\UriFactory` by registering your class with the Factory using `\Zend\Uri\UriFactory::registerScheme()` which takes the scheme as first parameter. This enables you to create your own *URI*-class and instantiate new URI objects based on your own custom classes.

[1] See http://www.ietf.org/rfc/rfc3986.txt for more information on URIs

[2] At the time of writing, `Zend\Uri` provides built-in support for the following schemes: *HTTP*, *HTTPS*, *MAILTO* and *FILE*

The 2nd parameter passed to `Zend\Uri\UriFactory::registerScheme()` must be a string with the name of a class implementing `Zend\Uri\UriInterface`. The class must either be already loaded, or be loadable by the autoloader.

Creating a URI using a custom class

```
// Create a new 'ftp' URI based on a custom class
use Zend\Uri\UriFactory

UriFactory::registerScheme('ftp', 'MyNamespace\MyClass');

$ftpUri = UriFactory::factory(
    'ftp://user@ftp.example.com/path/file'
);

// $ftpUri is an instance of MyLibrary\MyClass, which implements
// Zend\Uri\UriInterface
```

259.3 Manipulating an Existing URI

To manipulate an existing *URI*, pass the entire *URI* as string to `Zend\Uri\UriFactory::factory()`.

Manipulating an Existing URI with Zend\Uri\UriFactory::factory()

```
// To manipulate an existing URI, pass it in.
$uri = Zend\Uri\UriFactory::factory('http://www.zend.com');

// $uri instanceof Zend\Uri\UriInterface
```

The *URI* will be parsed and validated. If it is found to be invalid, a `Zend\Uri\Exception\InvalidArgumentException` will be thrown immediately. Otherwise, `Zend\Uri\UriFactory::factory()` will return a class implementing `Zend\Uri\UriInterface` that specializes in the scheme to be manipulated.

259.4 Common Instance Methods

The `Zend\Uri\UriInterface` defines several instance methods that are useful for working with any kind of *URI*.

259.4.1 Getting the Scheme of the URI

The scheme of the *URI* is the part of the *URI* that precedes the colon. For example, the scheme of `http://johndoe@example.com/my/path?query#token` is 'http'.

Getting the Scheme from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');

$scheme = $uri->getScheme();  // "mailto"
```

The `getScheme()` instance method returns only the scheme part of the *URI* object.

259.4.2 Getting the Userinfo of the URI

The userinfo of the *URI* is the optional part of the *URI* that follows the colon and comes before the host-part. For example, the userinfo of `http://johndoe@example.com/my/path?query#token` is 'johndoe'.

Getting the Username from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');

$scheme = $uri->getUserinfo();  // "john.doe"
```

The `getUserinfo()` method returns only the userinfo part of the *URI* object.

259.4.3 Getting the host of the URI

The host of the *URI* is the optional part of the *URI* that follows the user-part and comes before the path-part. For example, the host of `http://johndoe@example.com/my/path?query#token` is 'example.com'.

Getting the host from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');

$scheme = $uri->getHost();  // "example.com"
```

The `getHost()` method returns only the host part of the *URI* object.

259.4.4 Getting the port of the URI

The port of the *URI* is the optional part of the *URI* that follows the host-part and comes before the path-part. For example, the host of `http://johndoe@example.com:80/my/path?query#token` is '80'. The URI-class can define default-ports that can be returned when no port is given in the `URI`.

Getting the port from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:8080');

$scheme = $uri->getPort();  // "8080"
```

Getting a default port from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com');

$scheme = $uri->getPort();  // "80"
```

The `getHost()` method returns only the port part of the *URI* object.

259.4.5 Getting the path of the URI

The path of the *URI* is the mandatory part of the *URI* that follows the port and comes before the query-part. For example, the path of `http://johndoe@example.com:80/my/path?query#token` is '/my/path'.

Getting the path from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getPath();  // "/my/path"
```

The `getPath()` method returns only the path of the *URI* object.

259.4.6 Getting the query-part of the URI

The query-part of the *URI* is the optional part of the *URI* that follows the path and comes before the fragment. For example, the query of `http://johndoe@example.com:80/my/path?query#token` is 'query'.

Getting the query from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getQuery();  // "a=b&c=d"
```

The `getQuery()` method returns only the query-part of the *URI* object.

Getting the query as array from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getQueryAsArray();
// array(
//  'a' => 'b',
//  'c' => 'd',
// )
```

The query-part often contains key=value pairs and therefore can be split into an associative array. This array can be retrieved using `getQueryAsArray()`

259.4.7 Getting the fragment-part of the URI

The fragment-part of the *URI* is the optional part of the *URI* that follows the query. For example, the fragment of `http://johndoe@example.com:80/my/path?query#token` is 'token'.

Getting the fragment from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getFragment();  // "token"
```

The `getFragment()` method returns only the fragment-part of the *URI* object.

259.4.8 Getting the Entire URI

Getting the Entire URI from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://www.zend.com');

echo $uri->toString();  // "http://www.zend.com"

// Alternate method:
echo (string) $uri;     // "http://www.zend.com"
```

The `toString()` method returns the string representation of the entire *URI*.

The `Zend\Uri\UriInterface` defines also a magic `__toString()` method that returns the string representation of the *URI* when the Object is cast to a string.

259.4.9 Validating the URI

When using `Zend\Uri\UriFactory::factory()` the given *URI* will always be validated and a `Zend\Uri\Exception\InvalidArgumentException` will be thrown when the *URI* is invalid. However, after the `Zend\Uri\UriInterface` is instantiated for a new *URI* or an existing valid one, it is possible that the *URI* can later become invalid after it is manipulated.

Validating a ZendUri* Object

```
$uri = Zend\Uri\UriFactory::factory('http://www.zend.com');

$isValid = $uri->isValid();  // TRUE
```

The `isValid()` instance method provides a means to check that the *URI* object is still valid.

CHAPTER

INTRODUCTION TO ZEND\VALIDATOR

The `Zend\Validator` component provides a set of commonly needed validators. It also provides a simple validator chaining mechanism by which multiple validators may be applied to a single datum in a user-defined order.

260.1 What is a validator?

A validator examines its input with respect to some requirements and produces a boolean result - whether the input successfully validates against the requirements. If the input does not meet the requirements, a validator may additionally provide information about which requirement(s) the input does not meet.

For example, a web application might require that a username be between six and twelve characters in length and may only contain alphanumeric characters. A validator can be used for ensuring that a username meets these requirements. If a chosen username does not meet one or both of the requirements, it would be useful to know which of the requirements the username fails to meet.

260.2 Basic usage of validators

Having defined validation in this way provides the foundation for `Zend\Validator\ValidatorInterface`, which defines two methods, `isValid()` and `getMessages()`. The `isValid()` method performs validation upon the provided value, returning `TRUE` if and only if the value passes against the validation criteria.

If `isValid()` returns `FALSE`, the `getMessages()` returns an array of messages explaining the reason(s) for validation failure. The array keys are short strings that identify the reasons for validation failure, and the array values are the corresponding human-readable string messages. The keys and values are class-dependent; each validation class defines its own set of validation failure messages and the unique keys that identify them. Each class also has a const definition that matches each identifier for a validation failure cause.

Note: The `getMessages()` methods return validation failure information only for the most recent `isValid()` call. Each call to `isValid()` clears any messages and errors caused by a previous `isValid()` call, because it's likely that each call to `isValid()` is made for a different input value.

The following example illustrates validation of an e-mail address:

```
$validator = new Zend\Validator\EmailAddress();

if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $messageId => $message) {
        echo "Validation failure '$messageId': $message\n";
```

```
    }
}
```

260.3 Customizing messages

Validator classes provide a `setMessage()` method with which you can specify the format of a message returned by `getMessages()` in case of validation failure. The first argument of this method is a string containing the error message. You can include tokens in this string which will be substituted with data relevant to the validator. The token **%value%** is supported by all validators; this is substituted with the value you passed to `isValid()`. Other tokens may be supported on a case-by-case basis in each validation class. For example, **%max%** is a token supported by `Zend\Validator\LessThan`. The `getMessageVariables()` method returns an array of variable tokens supported by the validator.

The second optional argument is a string that identifies the validation failure message template to be set, which is useful when a validation class defines more than one cause for failure. If you omit the second argument, `setMessage()` assumes the message you specify should be used for the first message template declared in the validation class. Many validation classes only have one error message template defined, so there is no need to specify which message template you are changing.

```
$validator = new Zend\Validator\StringLength(8);

$validator->setMessage(
    'The string \'%value%\' is too short; it must be at least %min% ' .
    'characters',
    Zend\Validator\StringLength::TOO_SHORT);

if (!$validator->isValid('word')) {
    $messages = $validator->getMessages();
    echo current($messages);

    // "The string 'word' is too short; it must be at least 8 characters"
}
```

You can set multiple messages using the `setMessages()` method. Its argument is an array containing key/message pairs.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));

$validator->setMessages( array(
    Zend\Validator\StringLength::TOO_SHORT =>
        'The string \'%value%\' is too short',
    Zend\Validator\StringLength::TOO_LONG  =>
        'The string \'%value%\' is too long'
));
```

If your application requires even greater flexibility with which it reports validation failures, you can access properties by the same name as the message tokens supported by a given validation class. The `value` property is always available in a validator; it is the value you specified as the argument of `isValid()`. Other properties may be supported on a case-by-case basis in each validation class.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));

if (!$validator->isValid('word')) {
    echo 'Word failed: '
        . $validator->value
        . '; its length is not between '
```

```
        . $validator->min
        . ' and '
        . $validator->max
        . "\n";
}
```

260.4 Translating messages

Validator classes provide a `setTranslator()` method with which you can specify an instance of `Zend\I18n\Translator\Translator` which will translate the messages in case of a validation failure. The `getTranslator()` method returns the set translator instance.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
$translate = new Zend\I18n\Translator\Translator();
// configure the translator...

$validator->setTranslator($translate);
```

With the static `setDefaultTranslator()` method you can set a instance of `Zend\I18n\Translator\Translator` which will be used for all validation classes, and can be retrieved with `getDefaultTranslator()`. This prevents you from setting a translator manually for all validator classes, and simplifies your code.

```
$translate = new Zend\I18n\Translator\Translator();
// configure the translator...

Zend\Validator\AbstractValidator::setDefaultTranslator($translate);
```

Sometimes it is necessary to disable the translator within a validator. To archive this you can use the `setDisableTranslator()` method, which accepts a boolean parameter, and `isTranslatorDisabled()` to get the set value.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
if (!$validator->isTranslatorDisabled()) {
    $validator->setDisableTranslator();
}
```

It is also possible to use a translator instead of setting own messages with `setMessage()`. But doing so, you should keep in mind, that the translator works also on messages you set your own.

CHAPTER

ONE

STANDARD VALIDATION CLASSES

Zend Framework comes with a standard set of validation classes, which are ready for you to use.

CHAPTER

TWO

ALNUM

`Zend\Validator\Alnum` allows you to validate if a given value contains only alphabetical characters and digits. There is no length limitation for the input you want to validate.

262.1 Supported options for Zend\Validator\Alnum

The following options are supported for `Zend\Validator\Alnum`:

- **allowWhiteSpace**: If whitespace characters are allowed. This option defaults to `FALSE`

262.2 Basic usage

A basic example is the following one:

```
$validator = new Zend\Validator\Alnum();
if ($validator->isValid('Abcd12')) {
    // value contains only allowed chars
} else {
    // false
}
```

262.3 Using whitespaces

Per default whitespaces are not accepted because they are not part of the alphabet. Still, there is a way to accept them as input. This allows to validate complete sentences or phrases.

To allow the usage of whitespaces you need to give the `allowWhiteSpace` option. This can be done while creating an instance of the validator, or afterwards by using `setAllowWhiteSpace()`. To get the actual state you can use `getAllowWhiteSpace()`.

```
$validator = new Zend\Validator\Alnum(array('allowWhiteSpace' => true));
if ($validator->isValid('Abcd and 12')) {
    // value contains only allowed chars
} else {
    // false
}
```

262.4 Using different languages

When using `Zend\Validator\Alnum` then the language which the user sets within his browser will be used to set the allowed characters. This means when your user sets **de** for german then he can also enter characters like **ä**, **ö** and **ü** additionally to the characters from the english alphabet.

Which characters are allowed depends completely on the used language as every language defines it's own set of characters.

There are actually 3 languages which are not accepted in their own script. These languages are **korean**, **japanese** and **chinese** because this languages are using an alphabet where a single character is build by using multiple characters.

In the case you are using these languages, the input will only be validated by using the english alphabet.

CHAPTER

THREE

ALPHA

`Zend\Validator\Alpha` allows you to validate if a given value contains only alphabetical characters. There is no length limitation for the input you want to validate. This validator is related to the `Zend\Validator\Alnum` validator with the exception that it does not accept digits.

263.1 Supported options for Zend\Validator\Alpha

The following options are supported for `Zend\Validator\Alpha`:

- **allowWhiteSpace**: If whitespace characters are allowed. This option defaults to `FALSE`

263.2 Basic usage

A basic example is the following one:

```
$validator = new Zend\Validator\Alpha();
if ($validator->isValid('Abcd')) {
    // value contains only allowed chars
} else {
    // false
}
```

263.3 Using whitespaces

Per default whitespaces are not accepted because they are not part of the alphabet. Still, there is a way to accept them as input. This allows to validate complete sentences or phrases.

To allow the usage of whitespaces you need to give the `allowWhiteSpace` option. This can be done while creating an instance of the validator, or afterwards by using `setAllowWhiteSpace()`. To get the actual state you can use `getAllowWhiteSpace()`.

```
$validator = new Zend\Validator\Alpha(array('allowWhiteSpace' => true));
if ($validator->isValid('Abcd and efg')) {
    // value contains only allowed chars
} else {
    // false
}
```

263.4 Using different languages

When using `Zend\Validator\Alpha` then the language which the user sets within his browser will be used to set the allowed characters. This means when your user sets **de** for german then he can also enter characters like **ä**, **ö** and **ü** additionally to the characters from the english alphabet.

Which characters are allowed depends completely on the used language as every language defines it's own set of characters.

There are actually 3 languages which are not accepted in their own script. These languages are **korean**, **japanese** and **chinese** because this languages are using an alphabet where a single character is build by using multiple characters.

In the case you are using these languages, the input will only be validated by using the english alphabet.

CHAPTER

FOUR

BARCODE

`Zend\Validator\Barcode` allows you to check if a given value can be represented as barcode.

`Zend\Validator\Barcode` supports multiple barcode standards and can be extended with proprietary barcode implementations very easily. The following barcode standards are supported:

- **CODABAR**: Also known as Code-a-bar.

  This barcode has no length limitation. It supports only digits, and 6 special chars. Codabar is a self-checking barcode. This standard is very old. Common use cases are within airbills or photo labs where multi-part forms are used with dot-matrix printers.

- **CODE128**: CODE128 is a high density barcode.

  This barcode has no length limitation. It supports the first 128 ascii characters. When used with printing characters it has an checksum which is calculated modulo 103. This standard is used worldwide as it supports upper and lowercase characters.

- **CODE25**: Often called "two of five" or "Code25 Industrial".

  This barcode has no length limitation. It supports only digits, and the last digit can be an optional checksum which is calculated with modulo 10. This standard is very old and nowadays not often used. Common use cases are within the industry.

- **CODE25INTERLEAVED**: Often called "Code 2 of 5 Interleaved".

  This standard is a variant of CODE25. It has no length limitation, but it must contain an even amount of characters. It supports only digits, and the last digit can be an optional checksum which is calculated with modulo 10. It is used worldwide and common on the market.

- **CODE39**: CODE39 is one of the oldest available codes.

  This barcode has a variable length. It supports digits, upper cased alphabetical characters and 7 special characters like whitespace, point and dollar sign. It can have an optional checksum which is calculated with modulo 43. This standard is used worldwide and common within the industry.

- **CODE39EXT**: CODE39EXT is an extension of CODE39.

  This barcode has the same properties as CODE39. Additionally it allows the usage of all 128 ASCII characters. This standard is used worldwide and common within the industry.

- **CODE93**: CODE93 is the successor of CODE39.

  This barcode has a variable length. It supports digits, alphabetical characters and 7 special characters. It has an optional checksum which is calculated with modulo 47 and contains 2 characters. This standard produces a denser code than CODE39 and is more secure.

- **CODE93EXT**: CODE93EXT is an extension of CODE93.

This barcode has the same properties as CODE93. Additionally it allows the usage of all 128 ASCII characters. This standard is used worldwide and common within the industry.

- **EAN2**: EAN is the shortcut for "European Article Number".

  These barcode must have 2 characters. It supports only digits and does not have a checksum. This standard is mainly used as addition to EAN13 (ISBN) when printed on books.

- **EAN5**: EAN is the shortcut for "European Article Number".

  These barcode must have 5 characters. It supports only digits and does not have a checksum. This standard is mainly used as addition to EAN13 (ISBN) when printed on books.

- **EAN8**: EAN is the shortcut for "European Article Number".

  These barcode can have 7 or 8 characters. It supports only digits. When it has a length of 8 characters it includes a checksum. This standard is used worldwide but has a very limited range. It can be found on small articles where a longer barcode could not be printed.

- **EAN12**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 12 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used within the USA and common on the market. It has been superseded by EAN13.

- **EAN13**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 13 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used worldwide and common on the market.

- **EAN14**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 14 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used worldwide and common on the market. It is the successor for EAN13.

- **EAN18**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 18 characters. It support only digits. The last digit is always a checksum digit which is calculated with modulo 10. This code is often used for the identification of shipping containers.

- **GTIN12**: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN12 and is its successor. It's commonly used within the USA.

- **GTIN13**: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN13 and is its successor. It is used worldwide by industry.

- **GTIN14**: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN14 and is its successor. It is used worldwide and common on the market.

- **IDENTCODE**: Identcode is used by Deutsche Post and DHL. It's an specialized implementation of Code25.

  This barcode must have a length of 12 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is mainly used by the companies DP and DHL.

- **INTELLIGENTMAIL**: Intelligent Mail is a postal barcode.

  This barcode can have a length of 20, 25, 29 or 31 characters. It supports only digits, and contains no checksum. This standard is the successor of *PLANET* and *POSTNET*. It is mainly used by the United States Postal Services.

- **ISSN**: *ISSN* is the abbreviation for International Standard Serial Number.

This barcode can have a length of 8 or 13 characters. It supports only digits, and the last digit must be a checksum digit which is calculated with modulo 11. It is used worldwide for printed publications.

- **ITF14**: ITF14 is the GS1 implementation of an Interleaved Two of Five bar code.

  This barcode is a special variant of Interleaved 2 of 5. It must have a length of 14 characters and is based on GTIN14. It supports only digits, and the last digit must be a checksum digit which is calculated with modulo 10. It is used worldwide and common within the market.

- **LEITCODE**: Leitcode is used by Deutsche Post and DHL. It's an specialized implementation of Code25.

  This barcode must have a length of 14 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is mainly used by the companies DP and DHL.

- **PLANET**: Planet is the abbreviation for Postal Alpha Numeric Encoding Technique.

  This barcode can have a length of 12 or 14 characters. It supports only digits, and the last digit is always a checksum. This standard is mainly used by the United States Postal Services.

- **POSTNET**: Postnet is used by the US Postal Service.

  This barcode can have a length of 6, 7, 10 or 12 characters. It supports only digits, and the last digit is always a checksum. This standard is mainly used by the United States Postal Services.

- **ROYALMAIL**: Royalmail is used by Royal Mail.

  This barcode has no defined length. It supports digits, uppercase letters, and the last digit is always a checksum. This standard is mainly used by Royal Mail for their Cleanmail Service. It is also called *RM4SCC*.

- **SSCC**: SSCC is the shortcut for "Serial Shipping Container Code".

  This barcode is a variant of EAN barcode. It must have a length of 18 characters and supports only digits. The last digit must be a checksum digit which is calculated with modulo 10. It is commonly used by the transport industry.

- **UPCA**: UPC is the shortcut for "Universal Product Code".

  This barcode preceded EAN13. It must have a length of 12 characters and supports only digits. The last digit must be a checksum digit which is calculated with modulo 10. It is commonly used within the USA.

- **UPCE**: UPCE is the short variant from UPCA.

  This barcode is a smaller variant of UPCA. It can have a length of 6, 7 or 8 characters and supports only digits. When the barcode is 8 chars long it includes a checksum which is calculated with modulo 10. It is commonly used with small products where a UPCA barcode would not fit.

264.1 Supported options for Zend\Validator\Barcode

The following options are supported for `Zend\Validator\Barcode`:

- **adapter**: Sets the barcode adapter which will be used. Supported are all above noted adapters. When using a self defined adapter, then you have to set the complete class name.
- **checksum**: `TRUE` when the barcode should contain a checksum. The default value depends on the used adapter. Note that some adapters don't allow to set this option.
- **options**: Defines optional options for a self written adapters.

264.2 Basic usage

To validate if a given string is a barcode you just need to know its type. See the following example for an EAN13 barcode:

```
$valid = new Zend\Validator\Barcode('EAN13');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

264.3 Optional checksum

Some barcodes can be provided with an optional checksum. These barcodes would be valid even without checksum. Still, when you provide a checksum, then you should also validate it. By default, these barcode types perform no checksum validation. By using the `checksum` option you can define if the checksum will be validated or ignored.

```
$valid = new Zend\Validator\Barcode(array(
    'adapter'  => 'EAN13',
    'checksum' => false,
));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Note: Reduced security by disabling checksum validation

By switching off checksum validation you will also reduce the security of the used barcodes. Additionally you should note that you can also turn off the checksum validation for those barcode types which must contain a checksum value. Barcodes which would not be valid could then be returned as valid even if they are not.

264.4 Writing custom adapters

You may write custom barcode validators for usage with `Zend\Validator\Barcode`; this is often necessary when dealing with proprietary barcode types. To write your own barcode validator, you need the following information.

- **Length**: The length your barcode must have. It can have one of the following values:
  - **Integer**: A value greater 0, which means that the barcode must have this length.
  - **-1**: There is no limitation for the length of this barcode.
  - **"even"**: The length of this barcode must have a even amount of digits.
  - **"odd"**: The length of this barcode must have a odd amount of digits.
  - **array**: An array of integer values. The length of this barcode must have one of the set array values.
- **Characters**: A string which contains all allowed characters for this barcode. Also the integer value 128 is allowed, which means the first 128 characters of the ASCII table.
- **Checksum**: A string which will be used as callback for a method which does the checksum validation.

Your custom barcode validator must extend `Zend\Validator\Barcode\AbstractAdapter` or implement `Zend\Validator\Barcode\AdapterInterface`.

As an example, let's create a validator that expects an even number of characters that include all digits and the letters 'ABCDE', and which requires a checksum.

```
class My\Barcode\MyBar extends Zend\Validator\Barcode\AbstractAdapter
{
    protected $length     = 'even';
    protected $characters = '0123456789ABCDE';
    protected $checksum   = 'mod66';

    protected function mod66($barcode)
    {
        // do some validations and return a boolean
    }
}

$valid = new Zend\Validator\Barcode('My\Barcode\MyBar');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

CHAPTER

FIVE

BETWEEN

`Zend\Validator\Between` allows you to validate if a given value is between two other values.

Note: Zend\Validator\Between supports only number validation

It should be noted that `Zend\Validator\Between` supports only the validation of numbers. Strings or dates can not be validated with this validator.

265.1 Supported options for Zend\Validator\Between

The following options are supported for `Zend\Validator\Between`:

- **inclusive**: Defines if the validation is inclusive the minimum and maximum border values or exclusive. It defaults to `TRUE`.
- **max**: Sets the maximum border for the validation.
- **min**: Sets the minimum border for the validation.

265.2 Default behaviour for Zend\Validator\Between

Per default this validator checks if a value is between `min` and `max` where both border values are allowed as value.

```
$valid  = new Zend\Validator\Between(array('min' => 0, 'max' => 10));
$value  = 10;
$result = $valid->isValid($value);
// returns true
```

In the above example the result is `TRUE` due to the reason that per default the search is inclusively the border values. This means in our case that any value from '0' to '10' is allowed. And values like '-1' and '11' will return `FALSE`.

265.3 Validation exclusive the border values

Sometimes it is useful to validate a value by excluding the border values. See the following example:

```
$valid  = new Zend\Validator\Between(
    array(
        'min' => 0,
        'max' => 10,
        'inclusive' => false
    )
);
$value  = 10;
$result = $valid->isValid($value);
// returns false
```

The example is almost equal to our first example but we excluded the border value. Now the values '0' and '10' are no longer allowed and will return `FALSE`.

CHAPTER

SIX

CALLBACK

`Zend\Validator\Callback` allows you to provide a callback with which to validate a given value.

266.1 Supported options for Zend\Validator\Callback

The following options are supported for `Zend\Validator\Callback`:

- **callback**: Sets the callback which will be called for the validation.
- **options**: Sets the additional options which will be given to the callback.

266.2 Basic usage

The simplest usecase is to have a single function and use it as a callback. Let's expect we have the following function.

```
function myMethod($value)
{
    // some validation
    return true;
}
```

To use it within `Zend\Validator\Callback` you just have to call it this way:

```
$valid = new Zend\Validator\Callback('myMethod');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

266.3 Usage with closures

PHP 5.3 introduces closures, which are basically self-contained or **anonymous** functions. *PHP* considers closures another form of callback, and, as such, may be used with `Zend\Validator\Callback`. As an example:

```
$valid = new Zend\Validator\Callback(function($value){
    // some validation
    return true;
});
```

```

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

266.4 Usage with class-based callbacks

Of course it's also possible to use a class method as callback. Let's expect we have the following class method:

```
class MyClass
{
    public function myMethod($value)
    {
        // some validation
        return true;
    }
}
```

The definition of the callback is in this case almost the same. You have just to create an instance of the class before the method and create an array describing the callback:

```
$object = new MyClass;
$valid = new Zend\Validator\Callback(array($object, 'myMethod'));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

You may also define a static method as a callback. Consider the following class definition and validator usage:

```
class MyClass
{
    public static function test($value)
    {
        // some validation
        return true;
    }
}

$valid = new Zend\Validator\Callback(array('MyClass', 'test'));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Finally, if you are using *PHP* 5.3, you may define the magic method `__invoke()` in your class. If you do so, simply providing an instance of the class as the callback will also work:

```
class MyClass
{
    public function __invoke($value)
    {
        // some validation
```

```
        return true;
    }
}

$object = new MyClass();
$valid = new Zend\Validator\Callback($object);
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

266.5 Adding options

`Zend\Validator\Callback` also allows the usage of options which are provided as additional arguments to the callback.

Consider the following class and method definition:

```
class MyClass
{
    function myMethod($value, $option)
    {
        // some validation
        return true;
    }

    //if a context is present
    function myMethod($value, $context, $option)
    {
        // some validation
        return true;
    }

}
```

There are two ways to inform the validator of additional options: pass them in the constructor, or pass them to the `setOptions()` method.

To pass them to the constructor, you would need to pass an array containing two keys, "callback" and "options":

```
$valid = new Zend\Validator\Callback(array(
    'callback' => array('MyClass', 'myMethod'),
    'options'  => $option,
));

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Otherwise, you may pass them to the validator after instantiation:

```
$valid = new Zend\Validator\Callback(array('MyClass', 'myMethod'));
$valid->setOptions($option);

```

```
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

When there are additional values given to `isValid()` then these values will be added immediately after `$value`.

```
$valid = new Zend\Validator\Callback(array('MyClass', 'myMethod'));
$valid->setOptions($option);

if ($valid->isValid($input, $additional)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

When making the call to the callback, the value to be validated will always be passed as the first argument to the callback followed by all other values given to `isValid()`; all other options will follow it. The amount and type of options which can be used is not limited.

CHAPTER

SEVEN

CREDITCARD

`Zend\Validator\CreditCard` allows you to validate if a given value could be a credit card number.

A credit card contains several items of metadata, including a hologram, account number, logo, expiration date, security code and the card holder name. The algorithms for verifying the combination of metadata are only known to the issuing company, and should be verified with them for purposes of payment. However, it's often useful to know whether or not a given number actually falls within the ranges of possible numbers **prior** to performing such verification, and, as such, `Zend\Validator\CreditCard` simply verifies that the credit card number provided is well-formed.

For those cases where you have a service that can perform comprehensive verification, `Zend\Validator\CreditCard` also provides the ability to attach a service callback to trigger once the credit card number has been deemed valid; this callback will then be triggered, and its return value will determine overall validity.

The following issuing institutes are accepted:

- **American Express**
- **China UnionPay**
- **Diners Club Card Blanche**
- **Diners Club International**
- **Diners Club US & Canada**
- **Discover Card**
- **JCB**
- **Laser**
- **Maestro**
- **MasterCard**
- **Solo**
- **Visa**
- **Visa Electron**

Note: Invalid institutes

The institutes **Bankcard** and **Diners Club enRoute** do not exist anymore. Therefore they are treated as invalid.

Switch has been rebranded to **Visa** and is therefore also treated as invalid.

267.1 Supported options for Zend\Validator\CreditCard

The following options are supported for `Zend\Validator\CreditCard`:

- **service**: A callback to an online service which will additionally be used for the validation.
- **type**: The type of credit card which will be validated. See the below list of institutes for details.

267.2 Basic usage

There are several credit card institutes which can be validated by `Zend\Validator\CreditCard`. Per default, all known institutes will be accepted. See the following example:

```
$valid = new Zend\Validator\CreditCard();
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

The above example would validate against all known credit card institutes.

267.3 Accepting defined credit cards

Sometimes it is necessary to accept only defined credit card institutes instead of all; e.g., when you have a webshop which accepts only Visa and American Express cards. `Zend\Validator\CreditCard` allows you to do exactly this by limiting it to exactly these institutes.

To use a limitation you can either provide specific institutes at initiation, or afterwards by using `setType()`. Each can take several arguments.

You can provide a single institute:

```
$valid = new Zend\Validator\CreditCard(
    Zend\Validator\CreditCard::AMERICAN_EXPRESS
);
```

When you want to allow multiple institutes, then you can provide them as array:

```
$valid = new Zend\Validator\CreditCard(array(
    Zend\Validator\CreditCard::AMERICAN_EXPRESS,
    Zend\Validator\CreditCard::VISA
));
```

And as with all validators, you can also pass an associative array of options or an instance of `Traversable`. In this case you have to provide the institutes with the `type` array key as simulated here:

```
$valid = new Zend\Validator\CreditCard(array(
    'type' => array(Zend\Validator\CreditCard::AMERICAN_EXPRESS)
));
```

Table 267.1: Constants for credit card institutes

Institute	Constant
American Express	ZendValidatorCreditCard::AMERICAN_EXPRESS
China UnionPay	ZendValidatorCreditCard::UNIONPAY
Diners Club Card Blanche	ZendValidatorCreditCard::DINERS_CLUB
Diners Club International	ZendValidatorCreditCard::DINERS_CLUB
Diners Club US & Canada	ZendValidatorCreditCard::DINERS_CLUB_US
Discover Card	ZendValidatorCreditCard::DISCOVER
JCB	ZendValidatorCreditCard::JCB
Laser	ZendValidatorCreditCard::LASER
Maestro	ZendValidatorCreditCard::MAESTRO
MasterCard	ZendValidatorCreditCard::MASTERCARD
Solo	ZendValidatorCreditCard::SOLO
Visa	ZendValidatorCreditCard::VISA
Visa Electron	ZendValidatorCreditCard::VISA

You can also set or add institutes afterward instantiation by using the methods `setType()`, `addType()` and `getType()`.

```
$valid = new Zend\Validator\CreditCard();
$valid->setType(array(
    Zend\Validator\CreditCard::AMERICAN_EXPRESS,
    Zend\Validator\CreditCard::VISA
));
```

Note: Default institute

When no institute is given at initiation then `ALL` will be used, which sets all institutes at once.

In this case the usage of `addType()` is useless because all institutes are already added.

267.4 Validation by using foreign APIs

As said before `Zend\Validator\CreditCard` will only validate the credit card number. Fortunately, some institutes provide online *API*s which can validate a credit card number by using algorithms which are not available to the public. Most of these services are paid services. Therefore, this check is deactivated per default.

When you have access to such an *API*, then you can use it as an add on for `Zend\Validator\CreditCard` and increase the security of the validation.

To do so, you simply need to give a callback which will be called when the generic validation has passed. This prevents the *API* from being called for invalid numbers, which increases the performance of the application.

`setService()` sets a new service, and `getService()` returns the set service. As a configuration option, you can give the array key '`service`' at initiation. For details about possible options take a look into *Callback*.

```
// Your service class
class CcService
{
    public function checkOnline($cardnumber, $types)
    {
        // some online validation
    }
}
```

```
9
10 // The validation
11 $service = new CcService();
12 $valid   = new Zend\Validator\CreditCard(Zend\Validator\CreditCard::VISA);
13 $valid->setService(array($service, 'checkOnline'));
```

As you can see the callback method will be called with the credit card number as the first parameter, and the accepted types as the second parameter.

267.5 Ccnum

Note: The `Ccnum` validator has been deprecated in favor of the `CreditCard` validator. For security reasons you should use CreditCard instead of Ccnum.

CHAPTER

EIGHT

DATE

`Zend\Validator\Date` allows you to validate if a given value contains a date. This validator validates also localized input.

268.1 Supported options for Zend\Validator\Date

The following options are supported for `Zend\Validator\Date`:

- **format**: Sets the format which is used to write the date.
- **locale**: Sets the locale which will be used to validate date values.

268.2 Default date validation

The easiest way to validate a date is by using the default date format. It is used when no locale and no format has been given.

```
$validator = new Zend\Validator\Date();

$validator->isValid('2000-10-10');   // returns true
$validator->isValid('10.10.2000'); // returns false
```

The default date format for `Zend\Validator\Date` is 'yyyy-MM-dd'.

268.3 Localized date validation

`Zend\Validator\Date` validates also dates which are given in a localized format. By using the `locale` option you can define the locale which the date format should use for validation.

```
$validator = new Zend\Validator\Date(array('locale' => 'de'));

$validator->isValid('10.Feb.2010'); // returns true
$validator->isValid('10.May.2010'); // returns false
```

The `locale` option sets the default date format. In the above example this is 'j.M.Y' which is defined as default date format for 'de'.

268.4 Self defined date validation

`Zend\Validator\Date` supports also self defined date formats. When you want to validate such a date you can use the `format` option. This option accepts format as specified in the standard PHP function date().

```
$validator = new Zend\Validator\Date(array('format' => 'Y'));

$validator->isValid('2010'); // returns true
$validator->isValid('May');  // returns false
```

Of course you can combine `format` and `locale`. In this case you can also use localized month or day names.

```
$validator = new Zend\Validator\Date(array('format' => 'Y F', 'locale' => 'de'));

$validator->isValid('2010 Dezember'); // returns true
$validator->isValid('2010 June');     // returns false
```

CHAPTER

NINE

DB\RECORDEXISTS AND DB\NORECORDEXISTS

`Zend\Validator\Db\RecordExists` and `Zend\Validator\Db\NoRecordExists` provide a means to test whether a record exists in a given table of a database, with a given value.

269.1 Supported options for Zend\Validator\Db*

The following options are supported for `Zend\Validator\Db\NoRecordExists` and `Zend\Validator\Db\RecordExists`:

- **adapter**: The database adapter that will be used for the search.
- **exclude**: Sets records that will be excluded from the search.
- **field**: The database field within this table that will be searched for the record.
- **schema**: Sets the schema that will be used for the search.
- **table**: The table that will be searched for the record.

Note: In ZF1 it was possible to set an application wide default database adapter that was consumed by this class. As this is not possible in ZF2, it is now always required to supply an adapter.

269.2 Basic usage

An example of basic usage of the validators:

```
//Check that the email address exists in the database
$validator = new Zend\Validator\Db\RecordExists(
    array(
        'table'   => 'users',
        'field'   => 'emailaddress',
        'adapter' => $dbAdapter
    )
);

if ($validator->isValid($emailaddress)) {
    // email address appears to be valid
} else {
```

```
    // email address is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

The above will test that a given email address is in the database table. If no record is found containing the value of `$emailaddress` in the specified column, then an error message is displayed.

```
//Check that the username is not present in the database
$validator = new Zend\Validator\Db\NoRecordExists(
    array(
        'table'   => 'users',
        'field'   => 'username',
        'adapter' => $dbAdapter
    )
);
if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above will test that a given username is not in the database table. If a record is found containing the value of `$username` in the specified column, then an error message is displayed.

269.3 Excluding records

`Zend\Validator\Db\RecordExists` and `Zend\Validator\Db\NoRecordExists` also provide a means to test the database, excluding a part of the table, either by providing a where clause as a string, or an array with the keys "field" and "value".

When providing an array for the exclude clause, the **!=** operator is used, so you can check the rest of a table for a value before altering a record (for example on a user profile form)

```
//Check no other users have the username
$user_id   = $user->getId();
$validator = new Zend\Validator\Db\NoRecordExists(
    array(
        'table' => 'users',
        'field' => 'username',
        'exclude' => array(
            'field' => 'id',
            'value' => $user_id
        )
    )
);

if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
```

```
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above example will check the table to ensure no records other than the one where `id = $user_id` contains the value $username.

You can also provide a string to the exclude clause so you can use an operator other than **!=**. This can be useful for testing against composite keys.

```
$email     = 'user@example.com';
$clause    = $dbAdapter->quoteIdentifier('email') . ' = ' . $dbAdapter->quoteValue($email);
$validator = new Zend\Validator\Db\RecordExists(
    array(
        'table'   => 'users',
        'field'   => 'username',
        'adapter' => $dbAdapter,
        'exclude' => $clause
    )
);

if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above example will check the 'users' table to ensure that only a record with both the username `$username` and with the email `$email` is valid.

269.4 Database Schemas

You can specify a schema within your database for adapters such as PostgreSQL and DB/2 by simply supplying an array with `table` and `schema` keys. As in the example below:

```
$validator = new Zend\Validator\Db\RecordExists(
    array(
        'table'  => 'users',
        'schema' => 'my',
        'field'  => 'id'
    )
);
```

269.5 Using a Select object

It is also possible to supply the validators with a `Zend\Db\Sql\Select` object in place of options. The validator then uses this object instead of building its own. This allows for greater flexibility with selection of records used for validation.

```
$select = new Zend\Db\Sql\Select();
$select->from('users')
       ->where->equalTo('id', $user_id)
       ->where->equalTo('email', $email);

$validator = new Zend\Validator\Db\RecordExists($select);

// We still need to set our database adapter
$validator->setAdapter($dbAdapter);

// Validation is then performed as usual
if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}

The above example will check the 'users' table to ensure that only a record with both the username `
and with the email ``$email`` is valid.
```

CHAPTER

DIGITS

`Zend\Validator\Digits` validates if a given value contains only digits.

270.1 Supported options for Zend\Validator\Digits

There are no additional options for `Zend\Validator\Digits`:

270.2 Validating digits

To validate if a given value contains only digits and no other characters, simply call the validator like shown in this example:

```
$validator = new Zend\Validator\Digits();

$validator->isValid("1234567890"); // returns true
$validator->isValid(1234);         // returns true
$validator->isValid('1a234');      // returns false
```

Note: Validating numbers

When you want to validate numbers or numeric values, be aware that this validator only validates digits. This means that any other sign like a thousand separator or a comma will not pass this validator. In this case you should use `Zend\I18n\Validator\Int` or `Zend\I18n\Validator\Float`.

CHAPTER

ONE

EMAILADDRESS

`Zend\Validator\EmailAddress` allows you to validate an email address. The validator first splits the email address on local-part @ hostname and attempts to match these against known specifications for email addresses and hostnames.

271.1 Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\EmailAddress();
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This will match the email address `$email` and on failure populate `getMessages()` with useful error messages.

271.2 Options for validating Email Addresses

`Zend\Validator\EmailAddress` supports several options which can either be set at initiation, by giving an array with the related options, or afterwards, by using `setOptions()`. The following options are supported:

- **allow**: Defines which type of domain names are accepted. This option is used in conjunction with the hostname option to set the hostname validator. For more information about possible values of this option, look at *Hostname* and possible `ALLOW*` constants. This option defaults to `ALLOW_DNS`.
- **deep**: Defines if the servers MX records should be verified by a deep check. When this option is set to `TRUE` then additionally to MX records also the A, A6 and `AAAA` records are used to verify if the server accepts emails. This option defaults to `FALSE`.
- **domain**: Defines if the domain part should be checked. When this option is set to `FALSE`, then only the local part of the email address will be checked. In this case the hostname validator will not be called. This option defaults to `TRUE`.
- **hostname**: Sets the hostname validator with which the domain part of the email address will be validated.
- **mx**: Defines if the MX records from the server should be detected. If this option is defined to `TRUE` then the MX records are used to verify if the server accepts emails. This option defaults to `FALSE`.

```
$validator = new Zend\Validator\EmailAddress();
$validator->setOptions(array('domain' => false));
```

271.3 Complex local parts

`Zend\Validator\EmailAddress` will match any valid email address according to RFC2822. For example, valid emails include **bob@domain.com**, **bob+jones@domain.us**, **"bob@jones"@domain.com** and **"bob jones"@domain.com**

Some obsolete email formats will not currently validate (e.g. carriage returns or a "\" character in an email address).

271.4 Validating only the local part

If you need `Zend\Validator\EmailAddress` to check only the local part of an email address, and want to disable validation of the hostname, you can set the `domain` option to `FALSE`. This forces `Zend\Validator\EmailAddress` not to validate the hostname part of the email address.

```
$validator = new Zend\Validator\EmailAddress();
$validator->setOptions(array('domain' => FALSE));
```

271.5 Validating different types of hostnames

The hostname part of an email address is validated against *Zend\Validator\Hostname*. By default only DNS hostnames of the form `domain.com` are accepted, though if you wish you can accept IP addresses and Local hostnames too.

To do this you need to instantiate `Zend\Validator\EmailAddress` passing a parameter to indicate the type of hostnames you want to accept. More details are included in `Zend\Validator\Hostname`, though an example of how to accept both DNS and Local hostnames appears below:

```
$validator = new Zend\Validator\EmailAddress(
                    Zend\Validator\Hostname::ALLOW_DNS |
                    Zend\Validator\Hostname::ALLOW_LOCAL);
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

271.6 Checking if the hostname actually accepts email

Just because an email address is in the correct format, it doesn't necessarily mean that email address actually exists. To help solve this problem, you can use MX validation to check whether an MX (email) entry exists in the DNS record for the email's hostname. This tells you that the hostname accepts email, but doesn't tell you the exact email address itself is valid.

MX checking is not enabled by default. To enable MX checking you can pass a second parameter to the `Zend\Validator\EmailAddress` constructor.

```
$validator = new Zend\Validator\EmailAddress(
    array(
        'allow' => Zend\Validator\Hostname::ALLOW_DNS,
        'useMxCheck'    => true
    )
);
```

Note: MX Check under Windows

Within Windows environments MX checking is only available when *PHP* 5.3 or above is used. Below *PHP* 5.3 MX checking will not be used even if it's activated within the options.

Alternatively you can either pass `TRUE` or `FALSE` to `setValidateMx()` to enable or disable MX validation.

By enabling this setting network functions will be used to check for the presence of an MX record on the hostname of the email address you wish to validate. Please be aware this will likely slow your script down.

Sometimes validation for MX records returns `FALSE`, even if emails are accepted. The reason behind this behaviour is, that servers can accept emails even if they do not provide a MX record. In this case they can provide A, A6 or `AAAA` records. To allow `Zend\Validator\EmailAddress` to check also for these other records, you need to set deep MX validation. This can be done at initiation by setting the `deep` option or by using `setOptions()`.

```
$validator = new Zend\Validator\EmailAddress(
    array(
        'allow' => Zend\Validator\Hostname::ALLOW_DNS,
        'useMxCheck'    => true,
        'useDeepMxCheck'  => true
    )
);
```

Sometimes it can be useful to get the server's MX information which have been used to do further processing. Simply use `getMXRecord()` after validation. This method returns the received MX record including weight and sorted by it.

Warning: Performance warning
You should be aware that enabling MX check will slow down you script because of the used network functions. Enabling deep check will slow down your script even more as it searches the given server for 3 additional types.

Note: Disallowed IP addresses

You should note that MX validation is only accepted for external servers. When deep MX validation is enabled, then local IP addresses like `192.168.*` or `169.254.*` are not accepted.

271.7 Validating International Domains Names

`Zend\Validator\EmailAddress` will also match international characters that exist in some domains. This is known as International Domain Name (IDN) support. This is enabled by default, though you can disable this by changing the setting via the internal `Zend\Validator\Hostname` object that exists within `Zend\Validator\EmailAddress`.

```
$validator->getHostnameValidator()->setValidateIdn(false);
```

More information on the usage of `setValidateIdn()` appears in the `Zend\Validator\Hostname` documentation.

Please note IDNs are only validated if you allow DNS hostnames to be validated.

271.8 Validating Top Level Domains

By default a hostname will be checked against a list of known TLDs. This is enabled by default, though you can disable this by changing the setting via the internal `Zend\Validator\Hostname` object that exists within `Zend\Validator\EmailAddress`.

```
$validator->getHostnameValidator()->setValidateTld(false);
```

More information on the usage of `setValidateTld()` appears in the `Zend\Validator\Hostname` documentation.

Please note TLDs are only validated if you allow DNS hostnames to be validated.

271.9 Setting messages

`Zend\Validator\EmailAddress` makes also use of `Zend\Validator\Hostname` to check the hostname part of a given email address. As with Zend Framework 1.10 you can simply set messages for `Zend\Validator\Hostname` from within `Zend\Validator\EmailAddress`.

```
$validator = new Zend\Validator\EmailAddress();
$validator->setMessages(
    array(
        Zend\Validator\Hostname::UNKNOWN_TLD => 'I don\'t know the TLD you gave'
    )
);
```

Before Zend Framework 1.10 you had to attach the messages to your own `Zend\Validator\Hostname`, and then set this validator within `Zend\Validator\EmailAddress` to get your own messages returned.

CHAPTER

TWO

GREATERTHAN

`Zend\Validator\GreaterThan` allows you to validate if a given value is greater than a minimum border value.

Note: Zend\Validator\GreaterThan supports only number validation

It should be noted that `Zend\Validator\GreaterThan` supports only the validation of numbers. Strings or dates can not be validated with this validator.

272.1 Supported options for Zend\Validator\GreaterThan

The following options are supported for `Zend\Validator\GreaterThan`:

- **inclusive**: Defines if the validation is inclusive the minimum border value or exclusive. It defaults to `FALSE`.
- **min**: Sets the minimum allowed value.

272.2 Basic usage

To validate if a given value is greater than a defined border simply use the following example.

```
$valid  = new Zend\Validator\GreaterThan(array('min' => 10));
$value  = 8;
$return = $valid->isValid($value);
// returns false
```

The above example returns `TRUE` for all values which are greater than 10.

272.3 Validation inclusive the border value

Sometimes it is useful to validate a value by including the border value. See the following example:

```
$valid  = new Zend\Validator\GreaterThan(
    array(
        'min' => 10,
        'inclusive' => true
    )
);
$value  = 10;
```

```
$result = $valid->isValid($value);
// returns true
```

The example is almost equal to our first example but we included the border value. Now the value '10' is allowed and will return `TRUE`.

CHAPTER

THREE

HEX

`Zend\Validator\Hex` allows you to validate if a given value contains only hexadecimal characters. These are all characters from **0 to 9** and **A to F** case insensitive. There is no length limitation for the input you want to validate.

```
$validator = new Zend\Validator\Hex();
if ($validator->isValid('123ABC')) {
    // value contains only hex chars
} else {
    // false
}
```

Note: Invalid characters

All other characters will return false, including whitespace and decimal point. Also unicode zeros and numbers from other scripts than latin will not be treated as valid.

273.1 Supported options for Zend\Validator\Hex

There are no additional options for `Zend\Validator\Hex`:

CHAPTER

FOUR

HOSTNAME

`Zend\Validator\Hostname` allows you to validate a hostname against a set of known specifications. It is possible to check for three different types of hostnames: a *DNS* Hostname (i.e. `domain.com`), IP address (i.e. 1.2.3.4), and Local hostnames (i.e. localhost). By default only *DNS* hostnames are matched.

274.1 Supported options for Zend\Validator\Hostname

The following options are supported for `Zend\Validator\Hostname`:

- **allow**: Defines the sort of hostname which is allowed to be used. See *Hostname types* for details.
- **idn**: Defines if *IDN* domains are allowed or not. This option defaults to `TRUE`.
- **ip**: Allows to define a own IP validator. This option defaults to a new instance of `Zend\Validator\Ip`.
- **tld**: Defines if *TLD*s are validated. This option defaults to `TRUE`.

274.2 Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\Hostname();
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This will match the hostname `$hostname` and on failure populate `getMessages()` with useful error messages.

274.3 Validating different types of hostnames

You may find you also want to match IP addresses, Local hostnames, or a combination of all allowed types. This can be done by passing a parameter to `Zend\Validator\Hostname` when you instantiate it. The parameter should be an integer which determines what types of hostnames are allowed. You are encouraged to use the `Zend\Validator\Hostname` constants to do this.

The `Zend\Validator\Hostname` constants are: `ALLOW_DNS` to allow only *DNS* hostnames, `ALLOW_IP` to allow IP addresses, `ALLOW_LOCAL` to allow local network names, `ALLOW_URI` to allow RFC3986-compliant addresses, and `ALLOW_ALL` to allow all four above types.

Note: Additional Information on ALLOW_URI

`ALLOW_URI` allows to check hostnames according to RFC3986. These are registered names which are used by *WINS*, NetInfo and also local hostnames like those defined within your `.hosts` file.

To just check for IP addresses you can use the example below:

```
$validator = new Zend\Validator\Hostname(Zend\Validator\Hostname::ALLOW_IP);
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

As well as using `ALLOW_ALL` to accept all common hostnames types you can combine these types to allow for combinations. For example, to accept *DNS* and Local hostnames instantiate your `Zend\Validator\Hostname` class as so:

```
$validator = new Zend\Validator\Hostname(Zend\Validator\Hostname::ALLOW_DNS |
                                         Zend\Validator\Hostname::ALLOW_IP);
```

274.4 Validating International Domains Names

Some Country Code Top Level Domains (ccTLDs), such as 'de' (Germany), support international characters in domain names. These are known as International Domain Names (*IDN*). These domains can be matched by `Zend\Validator\Hostname` via extended characters that are used in the validation process.

Note: IDN domains

Until now more than 50 ccTLDs support *IDN* domains.

To match an *IDN* domain it's as simple as just using the standard Hostname validator since *IDN* matching is enabled by default. If you wish to disable *IDN* validation this can be done by either passing a parameter to the `Zend\Validator\Hostname` constructor or via the `setValidateIdn()` method.

You can disable *IDN* validation by passing a second parameter to the `Zend\Validator\Hostname` constructor in the following way.

```
$validator =
    new Zend\Validator\Hostname(
        array(
            'allow' => Zend\Validator\Hostname::ALLOW_DNS,
            'useIdnCheck'   => false
        )
    );
```

Alternatively you can either pass `TRUE` or `FALSE` to `setValidateIdn()` to enable or disable *IDN* validation. If you are trying to match an *IDN* hostname which isn't currently supported it is likely it will fail validation if it has any

international characters in it. Where a ccTLD file doesn't exist in `Zend/Validator/Hostname` specifying the additional characters a normal hostname validation is performed.

Note: IDN validation

Please note that *IDN*s are only validated if you allow *DNS* hostnames to be validated.

274.5 Validating Top Level Domains

By default a hostname will be checked against a list of known *TLD*s. If this functionality is not required it can be disabled in much the same way as disabling *IDN* support. You can disable *TLD* validation by passing a third parameter to the `Zend\Validator\Hostname` constructor. In the example below we are supporting *IDN* validation via the second parameter.

```
$validator =
    new Zend\Validator\Hostname(
        array(
            'allow' => Zend\Validator\Hostname::ALLOW_DNS,
            'useIdnCheck'   => true,
            'useTldCheck'   => false
        )
    );
```

Alternatively you can either pass `TRUE` or `FALSE` to `setValidateTld()` to enable or disable *TLD* validation.

Note: TLD validation

Please note *TLD*s are only validated if you allow *DNS* hostnames to be validated.

CHAPTER

FIVE

IBAN

`Zend\Validator\Iban` validates if a given value could be a *IBAN* number. *IBAN* is the abbreviation for "International Bank Account Number".

275.1 Supported options for Zend\Validator\Iban

The following options are supported for `Zend\Validator\Iban`:

- **locale**: Sets the locale which is used to get the *IBAN* format for validation.

275.2 IBAN validation

IBAN numbers are always related to a country. This means that different countries use different formats for their *IBAN* numbers. This is the reason why *IBAN* numbers always need a locale. By knowing this we already know how to use `Zend\Validator\Iban`.

275.2.1 Application wide locale

We could use the application wide locale. This means that when no option is given at initiation, `Zend\Validator\Iban` searches for the application wide locale. See the following code snippet:

```
// within bootstrap
Locale::setDefault('de_AT');

// within the module
$validator = new Zend\Validator\Iban();

if ($validator->isValid('AT611904300234573201')) {
    // IBAN appears to be valid
} else {
    // IBAN is not valid
}
```

Note: Application wide locale

Of course this works only when an application wide locale was set within the registry previously. Otherwise `Locale` will try to use the locale which the client sends or, when non has been send, it uses the environment locale. Be aware that this can lead to unwanted behaviour within the validation.

275.2.2 Ungreedy IBAN validation

Sometime it is useful, just to validate if the given value **is** a *IBAN* number or not. This means that you don't want to validate it against a defined country. This can be done by using a `FALSE` as locale.

```
$validator = new Zend\Validator\Iban(array('locale' => false));
// Note: you can also set a FALSE as single parameter

if ($validator->isValid('AT611904300234573201')) {
    // IBAN appears to be valid
} else {
    // IBAN is not valid
}
```

So **any** *IBAN* number will be valid. Note that this should not be done when you accept only accounts from a single country.

275.2.3 Region aware IBAN validation

To validate against a defined country, you just need to give the wished locale. You can do this by the option `locale` and also afterwards by using `setLocale()`.

```
$validator = new Zend\Validator\Iban(array('locale' => 'de_AT'));

if ($validator->isValid('AT611904300234573201')) {
    // IBAN appears to be valid
} else {
    // IBAN is not valid
}
```

Note: Use full qualified locales

You must give a full qualified locale, otherwise the country could not be detected correct because languages are spoken in multiple countries.

CHAPTER

SIX

IDENTICAL

`Zend\Validator\Identical` allows you to validate if a given value is identical with a set token.

276.1 Supported options for Zend\Validator\Identical

The following options are supported for `Zend\Validator\Identical`:

- **strict**: Defines if the validation should be done strict. The default value is `TRUE`.
- **token**: Sets the token with which the input will be validated against.

276.2 Basic usage

To validate if two values are identical you need to set the origin value as the token. See the following example which validates a string against the given token.

```
$valid = new Zend\Validator\Identical('origin');
if ($valid->isValid($value)) {
    return true;
}
```

The validation will only then return `TRUE` when both values are 100% identical. In our example, when `$value` is 'origin'.

You can set the wished token also afterwards by using the method `setToken()` and `getToken()` to get the actual set token.

276.3 Identical objects

Of course `Zend\Validator\Identical` can not only validate strings, but also any other variable type like Boolean, Integer, Float, Array or even Objects. As already noted Token and Value must be identical.

```
$valid = new Zend\Validator\Identical(123);
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Note: Type comparison

You should be aware that also the type of a variable is used for validation. This means that the string **'3'** is not identical with the integer **3**. When you want such a non strict validation you must set the `strict` option to `false`.

276.4 Form elements

`Zend\Validator\Identical` supports also the comparison of form elements. This can be done by using the element's name as `token`. See the following example:

```
$form->add(array(
    'name' => 'elementOne',
    'type' => 'Password',
));
$form->add(array(
    'name'       => 'elementTwo',
    'type'       => 'Password',
    'validators' => array(
        array(
            'name'    => 'Identical',
            'options' => array(
                'token' => 'elementOne',
            ),
        ),
    ),
));
```

By using the elements name from the first element as `token` for the second element, the validator validates if the second element is equal with the first element. In the case your user does not enter two identical values, you will get a validation error.

276.4.1 Validating a Value From a Fieldset

Sometimes you will need to validate an input that lives inside a fieldset, and this can be accomplished, see the following example.

```
use Zend\Form\Element;
use Zend\Form\Fieldset;
use Zend\Form\Form;
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;

$userFieldset = new Fieldset('user'); // (1)
$userFieldset->add(array(
    'name' => 'email', // (2)
    'type' => 'Email',
));

// Let's add one fieldset inside the 'user' fieldset,
// so we can see how to manage the token in a different deepness
$deeperFieldset = new Fieldset('deeperFieldset'); // (3)
$deeperFieldset->add(array(
    'name'    => 'deeperFieldsetInput', // (4)
```

```
    'type'    => 'Text',
    'options' => array(
        'label' => 'What validator are we testing?',
    ),
));
$userFieldset->add($deeperFieldset);

$signUpForm = new Form('signUp');
$signUpForm->add($userFieldset);
// Add an input that will validate the 'email' input from 'user' fieldset
$signUpForm->add(array(
    'name' => 'confirmEmail', // (5)
    'type' => 'Email',
));
// Add an input that will validate the 'deeperFieldsetInput' from 'deeperFieldset'
// that lives inside the 'user' fieldset
$signUpForm->add(array(
    'name' => 'confirmTestingValidator', // (6)
    'type' => 'Text',
));

$inputFilter = new InputFilter();
// This will ensure the user enter the same email in 'email' (2) and 'confirmEmail' (5)
$inputFilter->add(array(
    'name' => 'confirmEmail', // references (5)
    'validators' => array(
        array(
            'name' => 'Identical',
            'options' => array(
                // 'user' key references 'user' fieldset (1), and 'email' references 'email' element
                // 'user' fieldset (2)
                'token' => array('user' => 'email'),
            ),
        ),
    ),
));
// This will ensure the user enter the same string in 'deeperFieldsetInput' (4)
// and 'confirmTestingValidator' (6)
$inputFilter->add(array(
    'name' => 'confirmTestingValidator', // references (6)
    'validators' => array(
        array(
            'name' => 'Identical',
            'options' => array(
                'token' => array(
                    'user' => array( // references 'user' fieldset (1)
                        // 'deeperFieldset' key references 'deeperFieldset' fieldset (3)
                        // 'deeperFieldsetInput' references 'deeperFieldsetInput' element (4)
                        'deeperFieldset' => 'deeperFieldsetInput'
                    )
                ),
            ),
        ),
    ),
));

$signUpForm->setInputFilter($inputFilter);
```

Note: Aways make sure that your token array have just one key per level all the way till the leaf, otherwise you can end up with unexpected results.

276.5 Strict validation

As mentioned before `Zend\Validator\Identical` validates tokens strict. You can change this behaviour by using the `strict` option. The default value for this property is `TRUE`.

```
$valid = new Zend\Validator\Identical(array('token' => 123, 'strict' => FALSE));
$input = '123';
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

The difference to the previous example is that the validation returns in this case `TRUE`, even if you compare a integer with string value as long as the content is identical but not the type.

For convenience you can also use `setStrict()` and `getStrict()`.

276.6 Configuration

As all other validators, `Zend\Validator\Identical` also supports the usage of configuration settings as input parameter. This means that you can configure this validator with a `Traversable` object.

There is a case which you should be aware of. If you are using an array as token, and it contains a `'token'` key, you should wrap it within another `'token'` key. See the examples below to undestand this situation.

```
// This will not validate array('token' => 123), it will actually validate the integer 123
$valid = new Zend\Validator\Identical(array('token' => 123));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

The reason for this special case is that you can configure the token which has to be used by giving the `'token'` key.

So, when you are using an array as token, and it contains one element with a `'token'` key, then you have to wrap it like shown in the example below.

```
// Unlike the previous example, this will validate array('token' => 123)
$valid = new Zend\Validator\Identical(array('token' => array('token' => 123)));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

If the array you are willing to validate does not have a `'token'` key, you do not need to wrap it.

CHAPTER

SEVEN

INARRAY

`Zend\Validator\InArray` allows you to validate if a given value is contained within an array. It is also able to validate multidimensional arrays.

277.1 Supported options for Zend\Validator\InArray

The following options are supported for `Zend\Validator\InArray`:

- **haystack**: Sets the haystack for the validation.
- **recursive**: Defines if the validation should be done recursive. This option defaults to `FALSE`.
- **strict**: Three modes of comparison are offered owing to an often overlooked, and potentially dangerous security issue when validating string input from user input.
  - *InArray::COMPARE_STRICT*

    This is a normal in_array strict comparison that checks value and type.
  - *InArray::COMPARE_NOT_STRICT*

    This is a normal in_array non-strict comparison that checks value only.

> **Warning:** This mode may give false positives when strings are compared against ints or floats owing to in_array's behaviour of converting strings to int in such cases. Therefore, *"foo"* would become *0*, *"43foo"* would become *43*, while *"foo43"* would also become *0*.

- *InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_VULNERABILITY*

  To remedy the above warning, this mode offers a middle-ground which allows string representations of numbers to be successfully matched against either their string or int counterpart and vice versa. For example: *"0"* will successfully match against *0*, but *"foo"* would not match against *0* as would be true in the `*COMPARE_NOT_STRICT*` mode. This is the safest option to use when validating web input, and is the default.

Defines if the validation should be done strict. This option defaults to `FALSE`.

277.2 Simple array validation

The simplest way, is just to give the array which should be searched against at initiation:

```
$validator = new Zend\Validator\InArray(array('haystack' => array('value1', 'value2',...'valueN')));
if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

This will behave exactly like *PHP*'s `in_array()` method.

Note: Per default this validation is not strict nor can it validate multidimensional arrays.

Alternatively, you can define the array to validate against after object construction by using the `setHaystack()` method. `getHaystack()` returns the actual set haystack array.

```
$validator = new Zend\Validator\InArray();
$validator->setHaystack(array('value1', 'value2',...'valueN'));

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

277.3 Array validation modes

As previously mentioned, there are possible security issues when using the default non-strict comparison mode, so rather than restricting the developer, we've chosen to offer both strict and non-strict comparisons and adding a safer middle-ground.

It's possible to set the strict mode at initialisation and afterwards with the `setStrict` method. `InArray::COMPARE_STRICT` equates to `true` and `InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_VULNERABILITY` equates to false.

```
// defaults to InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_VULNERABILITY
$validator = new Zend\Validator\InArray(
    array(
         'haystack' => array('value1', 'value2',...'valueN'),
    )
);

// set strict mode
$validator = new Zend\Validator\InArray(
    array(
         'haystack' => array('value1', 'value2',...'valueN'),
         'strict'   => InArray::COMPARE_STRICT  // equates to ``true``
    )
);

// set non-strict mode
$validator = new Zend\Validator\InArray(
    array(
         'haystack' => array('value1', 'value2',...'valueN'),
         'strict'   => InArray:COMPARE_NOT_STRICT  // equates to ``false``
    )
);
```

```

// or

$validator->setStrict(InArray::COMPARE_STRICT);
$validator->setStrict(InArray::COMPARE_NOT_STRICT);
$validator->setStrict(InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_VULNERABILITY);
```

Note: Note that the **strict** setting is per default `FALSE`.

277.4 Recursive array validation

In addition to *PHP*'s `in_array()` method this validator can also be used to validate multidimensional arrays.

To validate multidimensional arrays you have to set the **recursive** option.

```
$validator = new Zend\Validator\InArray(
    array(
        'haystack' => array(
            'firstDimension' => array('value1', 'value2',...'valueN'),
            'secondDimension' => array('foo1', 'foo2',...'fooN')),
        'recursive' => true
    )
);

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

Your array will then be validated recursively to see if the given value is contained. Additionally you could use `setRecursive()` to set this option afterwards and `getRecursive()` to retrieve it.

```
$validator = new Zend\Validator\InArray(
    array(
        'firstDimension' => array('value1', 'value2',...'valueN'),
        'secondDimension' => array('foo1', 'foo2',...'fooN')
    )
);

$validator->setRecursive(true);

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

Note: Default setting for recursion

Per default the recursive validation is turned off.

Note: Option keys within the haystack

When you are using the keys ‘`haystack`‘, ‘`strict`‘ or ‘`recursive`‘ within your haystack, then you must wrap the `haystack` key.

CHAPTER

EIGHT

IP

`Zend\Validator\Ip` allows you to validate if a given value is an IP address. It supports the IPv4, IPv6 and IPvFeature definitions.

278.1 Supported options for Zend\Validator\Ip

The following options are supported for `Zend\Validator\Ip`:

- **allowipv4**: Defines if the validator allows IPv4 addresses. This option defaults to `TRUE`.
- **allowipv6**: Defines if the validator allows IPv6 addresses. This option defaults to `TRUE`.
- **allowipvfuture**: Defines if the validator allows IPvFuture addresses. This option defaults to `false`.
- **allowliteral**: Defines if the validator allows IPv6 or IPvFuture with URI literal style (the IP surrounded by brackets). This option defaults to `true`.

278.2 Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\Ip();
if ($validator->isValid($ip)) {
    // ip appears to be valid
} else {
    // ip is invalid; print the reasons
}
```

Note: Invalid IP addresses

Keep in mind that `Zend\Validator\Ip` only validates IP addresses. Addresses like '`mydomain.com`' or '`192.168.50.1/index.html`' are no valid IP addresses. They are either hostnames or valid *URL*s but not IP addresses.

Note: IPv6/IPvFuture validation

`Zend\Validator\Ip` validates IPv6/IPvFuture addresses with regex. The reason is that the filters and methods from *PHP* itself don't follow the *RFC*. Many other available classes also don't follow it.

278.3 Validate IPv4 or IPV6 alone

Sometimes it's useful to validate only one of the supported formats. For example when your network only supports IPv4. In this case it would be useless to allow IPv6 within this validator.

To limit `Zend\Validator\Ip` to one protocol you can set the options `allowipv4` or `allowipv6` to `FALSE`. You can do this either by giving the option to the constructor or by using `setOptions()` afterwards.

```
$validator = new Zend\Validator\Ip(array('allowipv6' => false));
if ($validator->isValid($ip)) {
    // ip appears to be valid ipv4 address
} else {
    // ip is no ipv4 address
}
```

Note: Default behaviour

The default behaviour which `Zend\Validator\Ip` follows is to allow both standards.

CHAPTER

NINE

ISBN

`Zend\Validator\Isbn` allows you to validate an *ISBN-10* or *ISBN-13* value.

279.1 Supported options for Zend\Validator\Isbn

The following options are supported for `Zend\Validator\Isbn`:

- **separator**: Defines the allowed separator for the *ISBN* number. It defaults to an empty string.
- **type**: Defines the allowed type of *ISBN* numbers. It defaults to `Zend\Validator\Isbn::AUTO`. For details take a look at *this section*.

279.2 Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\Isbn();
if ($validator->isValid($isbn)) {
    // isbn is valid
} else {
    // isbn is not valid
}
```

This will validate any *ISBN-10* and *ISBN-13* without separator.

279.3 Setting an explicit ISBN validation type

An example of an *ISBN* type restriction is below:

```
$validator = new Zend\Validator\Isbn();
$validator->setType(Zend\Validator\Isbn::ISBN13);
// OR
$validator = new Zend\Validator\Isbn(array(
    'type' => Zend\Validator\Isbn::ISBN13,
));

if ($validator->isValid($isbn)) {
    // this is a valid ISBN-13 value
} else {
```

```
    // this is an invalid ISBN-13 value
}
```

The above will validate only *ISBN-13* values.

Valid types include:

- `Zend\Validator\Isbn::AUTO` (default)
- `Zend\Validator\Isbn::ISBN10`
- `Zend\Validator\Isbn::ISBN13`

279.4 Specifying a separator restriction

An example of separator restriction is below:

```
$validator = new Zend\Validator\Isbn();
$validator->setSeparator('-');
// OR
$validator = new Zend\Validator\Isbn(array(
    'separator' => '-',
));

if ($validator->isValid($isbn)) {
    // this is a valid ISBN with separator
} else {
    // this is an invalid ISBN with separator
}
```

Note: Values without separator

This will return `FALSE` if `$isbn` doesn't contain a separator **or** if it's an invalid *ISBN* value.

Valid separators include:

- "" (empty) (default)
- "-" (hyphen)
- " " (space)

CHAPTER

LESSTHAN

`Zend\Validator\LessThan` allows you to validate if a given value is less than a maximum border value.

Note: Zend\Validator\LessThan supports only number validation

It should be noted that `Zend\Validator\LessThan` supports only the validation of numbers. Strings or dates can not be validated with this validator.

280.1 Supported options for Zend\Validator\LessThan

The following options are supported for `Zend\Validator\LessThan`:

- **inclusive**: Defines if the validation is inclusive the maximum border value or exclusive. It defaults to `FALSE`.
- **max**: Sets the maximum allowed value.

280.2 Basic usage

To validate if a given value is less than a defined border simply use the following example.

```
$valid  = new Zend\Validator\LessThan(array('max' => 10));
$value  = 12;
$return = $valid->isValid($value);
// returns false
```

The above example returns `TRUE` for all values which are lower than 10.

280.3 Validation inclusive the border value

Sometimes it is useful to validate a value by including the border value. See the following example:

```
$valid  = new Zend\Validator\LessThan(
    array(
        'max' => 10,
        'inclusive' => true
    )
);
$value  = 10;
$result = $valid->isValid($value);
// returns true
```

The example is almost equal to our first example but we included the border value. Now the value '10' is allowed and will return `TRUE`.

CHAPTER

ONE

NOTEMPTY

This validator allows you to validate if a given value is not empty. This is often useful when working with form elements or other user input, where you can use it to ensure required elements have values associated with them.

281.1 Supported options for Zend\Validator\NotEmpty

The following options are supported for `Zend\Validator\NotEmpty`:

- **type**: Sets the type of validation which will be processed. For details take a look into *this section*.

281.2 Default behaviour for Zend\Validator\NotEmpty

By default, this validator works differently than you would expect when you've worked with *PHP*'s `empty()` function. In particular, this validator will evaluate both the integer **0** and string '**0**' as empty.

```
$valid = new Zend\Validator\NotEmpty();
$value  = '';
$result = $valid->isValid($value);
// returns false
```

Note: Default behaviour differs from PHP

Without providing configuration, `Zend\Validator\NotEmpty`'s behaviour differs from *PHP*.

281.3 Changing behaviour for Zend\Validator\NotEmpty

Some projects have differing opinions of what is considered an "empty" value: a string with only whitespace might be considered empty, or **0** may be considered non-empty (particularly for boolean sequences). To accommodate differing needs, `Zend\Validator\NotEmpty` allows you to configure which types should be validated as empty and which not.

The following types can be handled:

- **boolean**: Returns `FALSE` when the boolean value is `FALSE`.
- **integer**: Returns `FALSE` when an integer **0** value is given. Per default this validation is not activated and returns `TRUE` on any integer values.

- **float**: Returns `FALSE` when an float **0.0** value is given. Per default this validation is not activated and returns `TRUE` on any float values.
- **string**: Returns `FALSE` when an empty string **''** is given.
- **zero**: Returns `FALSE` when the single character zero (**'0'**) is given.
- **empty_array**: Returns `FALSE` when an empty **array** is given.
- **null**: Returns `FALSE` when an `NULL` value is given.
- **php**: Returns `FALSE` on the same reasons where *PHP* method `empty()` would return `TRUE`.
- **space**: Returns `FALSE` when an string is given which contains only whitespaces.
- **object**: Returns `TRUE`. `FALSE` will be returned when `object` is not allowed but an object is given.
- **object_string**: Returns `FALSE` when an object is given and it's `__toString()` method returns an empty string.
- **object_count**: Returns `FALSE` when an object is given, it has an `Countable` interface and it's count is 0.
- **all**: Returns `FALSE` on all above types.

All other given values will return `TRUE` per default.

There are several ways to select which of the above types are validated. You can give one or multiple types and add them, you can give an array, you can use constants, or you can give a textual string. See the following examples:

```
// Returns false on 0
$validator = new Zend\Validator\NotEmpty(Zend\Validator\NotEmpty::INTEGER);

// Returns false on 0 or '0'
$validator = new Zend\Validator\NotEmpty(
    Zend\Validator\NotEmpty::INTEGER + Zend\Validator\NotEmpty::ZERO
);

// Returns false on 0 or '0'
$validator = new Zend\Validator\NotEmpty(array(
    Zend\Validator\NotEmpty::INTEGER,
    Zend\Validator\NotEmpty::ZERO
));

// Returns false on 0 or '0'
$validator = new Zend\Validator\NotEmpty(array(
    'integer',
    'zero',
));
```

You can also provide an instance of `Traversable` to set the desired types. To set types after instantiation, use the `setType()` method.

CHAPTER

TWO

POSTCODE

`Zend\Validator\PostCode` allows you to determine if a given value is a valid postal code. Postal codes are specific to cities, and in some locales termed *ZIP* codes.

`Zend\Validator\PostCode` knows more than 160 different postal code formats. To select the correct format there are 2 ways. You can either use a fully qualified locale or you can set your own format manually.

Using a locale is more convenient as Zend Framework already knows the appropriate postal code format for each locale; however, you need to use the fully qualified locale (one containing a region specifier) to do so. For instance, the locale "de" is a locale but could not be used with `Zend\Validator\PostCode` as it does not include the region; "de_AT", however, would be a valid locale, as it specifies the region code ("AT", for Austria).

```
$validator = new Zend\Validator\PostCode('de_AT');
```

When you don't set a locale yourself, then `Zend\Validator\PostCode` will use the application wide set locale, or, when there is none, the locale returned by `Locale`.

```
// application wide locale within your bootstrap
Locale::setDefault('de_AT');

$validator = new Zend\Validator\PostCode();
```

You can also change the locale afterwards by calling `setLocale()`. And of course you can get the actual used locale by calling `getLocale()`.

```
$validator = new Zend\Validator\PostCode('de_AT');
$validator->setLocale('en_GB');
```

Postal code formats are simply regular expression strings. When the international postal code format, which is used by setting the locale, does not fit your needs, then you can also manually set a format by calling `setFormat()`.

```
$validator = new Zend\Validator\PostCode('de_AT');
$validator->setFormat('AT-\d{5}');
```

Note: Conventions for self defined formats

When using self defined formats you should omit the starting (`'/^'`) and ending tags (`'$/'`). They are attached automatically.

You should also be aware that postcode values are always be validated in a strict way. This means that they have to be written standalone without additional characters when they are not covered by the format.

282.1 Constructor options

At it's most basic, you may pass a string representing a fully qualified locale to the constructor of `Zend\Validator\PostCode`.

```
$validator = new Zend\Validator\PostCode('de_AT');
$validator = new Zend\Validator\PostCode($locale);
```

Additionally, you may pass either an array or a `Traversable` instance to the constructor. When you do so, you must include either the key "locale" or "format"; these will be used to set the appropriate values in the validator object.

```
$validator = new Zend\Validator\PostCode(array(
    'locale' => 'de_AT',
    'format' => 'AT_\d+'
));
```

282.2 Supported options for Zend\Validator\PostCode

The following options are supported for `Zend\Validator\PostCode`:

- **format**: Sets a postcode format which will be used for validation of the input.
- **locale**: Sets a locale from which the postcode will be taken from.

CHAPTER

THREE

REGEX

This validator allows you to validate if a given string conforms a defined regular expression.

283.1 Supported options for Zend\Validator\Regex

The following options are supported for `Zend\Validator\Regex`:

- **pattern**: Sets the regular expression pattern for this validator.

283.2 Validation with Zend\Validator\Regex

Validation with regular expressions allows to have complicated validations being done without writing a own validator. The usage of regular expression is quite common and simple. Let's look at some examples:

```
$validator = new Zend\Validator\Regex(array('pattern' => '/^Test/');

$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns true
$validator->isValid("Pest"); // returns false
```

As you can see, the pattern has to be given using the same syntax as for `preg_match()`. For details about regular expressions take a look into PHP's manual about PCRE pattern syntax.

283.3 Pattern handling

It is also possible to set a different pattern afterwards by using `setPattern()` and to get the actual set pattern with `getPattern()`.

```
$validator = new Zend\Validator\Regex(array('pattern' => '/^Test/');
$validator->setPattern('ing$/');

$validator->isValid("Test"); // returns false
$validator->isValid("Testing"); // returns true
$validator->isValid("Pest"); // returns false
```

CHAPTER

FOUR

SITEMAP VALIDATORS

The following validators conform to the Sitemap XML protocol.

284.1 Sitemap\Changefreq

Validates whether a string is valid for using as a 'changefreq' element in a Sitemap *XML* document. Valid values are: 'always', 'hourly', 'daily', 'weekly', 'monthly', 'yearly', or 'never'.

Returns `TRUE` if and only if the value is a string and is equal to one of the frequencies specified above.

284.2 Sitemap\Lastmod

Validates whether a string is valid for using as a 'lastmod' element in a Sitemap *XML* document. The lastmod element should contain a *W3C* date string, optionally discarding information about time.

Returns `TRUE` if and only if the given value is a string and is valid according to the protocol.

Sitemap Lastmod Validator

```
$validator = new Zend\Validator\Sitemap\Lastmod();

$validator->isValid('1999-11-11T22:23:52-02:00'); // true
$validator->isValid('2008-05-12T00:42:52+02:00'); // true
$validator->isValid('1999-11-11'); // true
$validator->isValid('2008-05-12'); // true

$validator->isValid('1999-11-11t22:23:52-02:00'); // false
$validator->isValid('2008-05-12T00:42:60+02:00'); // false
$validator->isValid('1999-13-11'); // false
$validator->isValid('2008-05-32'); // false
$validator->isValid('yesterday'); // false
```

284.3 Sitemap\Loc

Validates whether a string is valid for using as a 'loc' element in a Sitemap *XML* document. This uses `Zend\Uri\Uri::isValid()` internally. Read more at *URI Validation*.

284.4 Sitemap\Priority

Validates whether a value is valid for using as a 'priority' element in a Sitemap *XML* document. The value should be a decimal between 0.0 and 1.0. This validator accepts both numeric values and string values.

Sitemap Priority Validator

```
$validator = new Zend\Validator\Sitemap\Priority();

$validator->isValid('0.1'); // true
$validator->isValid('0.789'); // true
$validator->isValid(0.8); // true
$validator->isValid(1.0); // true

$validator->isValid('1.1'); // false
$validator->isValid('-0.4'); // false
$validator->isValid(1.00001); // false
$validator->isValid(0xFF); // false
$validator->isValid('foo'); // false
```

284.5 Supported options for Zend\Validator\Sitemap_*

There are no supported options for any of the Sitemap validators.

CHAPTER

FIVE

STEP

`Zend\Validator\Step` allows you to validate if a given value is a valid step value. This validator requires the value to be a numeric value (either string, int or float).

285.1 Supported options for Zend\Validator\Step

The following options are supported for `Zend\Validator\Step`:

- **baseValue**: This is the base value from which the step should be computed. This option defaults to `0`
- **step**: This is the step value. This option defaults to `1`

285.2 Basic usage

A basic example is the following one:

```
$validator = new Zend\Validator\Step();
if ($validator->isValid(1)) {
// value is a valid step value
} else {
// false
}
```

285.3 Using floating-point values

This validator also supports floating-point base value and step value. Here is a basic example of this feature:

```
$validator = new Zend\Validator\Step(array(
'baseValue' => 1.1,
'step' => 2.2
));

echo $validator->isValid(1.1); // prints true
echo $validator->isValid(3.3); // prints true
echo $validator->isValid(3.35); // prints false
echo $validator->isValid(2.2); // prints false
```

CHAPTER

SIX

STRINGLENGTH

This validator allows you to validate if a given string is between a defined length.

Note: Zend\Validator\StringLength supports only string validation

It should be noted that `Zend\Validator\StringLength` supports only the validation of strings. Integers, floats, dates or objects can not be validated with this validator.

286.1 Supported options for Zend\Validator\StringLength

The following options are supported for `Zend\Validator\StringLength`:

- **encoding**: Sets the `ICONV` encoding which has to be used for this string.
- **min**: Sets the minimum allowed length for a string.
- **max**: Sets the maximum allowed length for a string.

286.2 Default behaviour for Zend\Validator\StringLength

Per default this validator checks if a value is between `min` and `max`. But for `min` the default value is **0** and for `max` it is **NULL** which means unlimited.

So per default, without giving any options, this validator only checks if the input is a string.

286.3 Limiting the maximum allowed length of a string

To limit the maximum allowed length of a string you need to set the `max` property. It accepts an integer value as input.

```
$validator = new Zend\Validator\StringLength(array('max' => 6));

$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns false
```

You can set the maximum allowed length also afterwards by using the `setMax()` method. And `getMax()` to retrieve the actual maximum border.

```
$validator = new Zend\Validator\StringLength();
$validator->setMax(6);

$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns false
```

286.4 Limiting the minimal required length of a string

To limit the minimal required length of a string you need to set the `min` property. It accepts also an integer value as input.

```
$validator = new Zend\Validator\StringLength(array('min' => 5));

$validator->isValid("Test"); // returns false
$validator->isValid("Testing"); // returns true
```

You can set the minimal requested length also afterwards by using the `setMin()` method. And `getMin()` to retrieve the actual minimum border.

```
$validator = new Zend\Validator\StringLength();
$validator->setMin(5);

$validator->isValid("Test"); // returns false
$validator->isValid("Testing"); // returns true
```

286.5 Limiting a string on both sides

Sometimes it is required to get a string which has a maximal defined length but which is also minimal chars long. For example when you have a textbox where a user can enter his name, then you may want to limit the name to maximum 30 chars but want to get sure that he entered his name. So you limit the minimum required length to 3 chars. See the following example:

```
$validator = new Zend\Validator\StringLength(array('min' => 3, 'max' => 30));

$validator->isValid("."); // returns false
$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns true
```

Note: Setting a lower maximum border than the minimum border

When you try to set a lower maximum value as the actual minimum value, or a higher minimum value as the actual maximum value, then an exception will be raised.

286.6 Encoding of values

Strings are always using a encoding. Even when you don't set the encoding explicit, *PHP* uses one. When your application is using a different encoding than *PHP* itself then you should set an encoding yourself.

You can set your own encoding at initiation with the `encoding` option, or by using the `setEncoding()` method. We assume that your installation uses *ISO* and your application it set to *ISO*. In this case you will see the below behaviour.

```
$validator = new Zend\Validator\StringLength(
    array('min' => 6)
);
$validator->isValid("Ärger"); // returns false

$validator->setEncoding("UTF-8");
$validator->isValid("Ärger"); // returns true

$validator2 = new Zend\Validator\StringLength(
    array('min' => 6, 'encoding' => 'UTF-8')
);
$validator2->isValid("Ärger"); // returns true
```

So when your installation and your application are using different encodings, then you should always set an encoding yourself.

CHAPTER

SEVEN

FILE VALIDATION CLASSES

Zend Framework comes with a set of classes for validating files, such as file size validation and CRC checking.

Note: All of the File validators' `filter()` methods support both a file path `string` *or* a `$_FILES` array as the supplied argument. When a `$_FILES` array is passed in, the `tmp_name` is used for the file path.

287.1 Crc32

`Zend\Validator\File\Crc32` allows you to validate if a given file's hashed contents matches the supplied crc32 hash(es). It is subclassed from the *Hash validator* to provide a convenient validator that only supports the `crc32` algorithm.

Note: This validator requires the Hash extension from PHP with the `crc32` algorithm.

Supported Options

The following set of options are supported:

- **hash (`string`)** Hash to test the file against.

Usage Examples

```
// Does file have the given hash?
$validator = new \Zend\Validator\File\Crc32('3b3652f');

// Or, check file against multiple hashes
$validator = new \Zend\Validator\File\Crc32(array('3b3652f', 'e612b69'));

// Perform validation with file path
if ($validator->isValid('./myfile.txt')) {
    // file is valid
}
```

Public Methods

getCrc32()
Returns the current set of crc32 hashes.

Return type `array`

addCrc32(*string|array $options*)
Adds a crc32 hash for one or multiple files to the internal set of hashes.

Parameters **$options** – See *Supported Options* section for more information.

setCrc32(*string|array $options*)
Sets a crc32 hash for one or multiple files. Removes any previously set hashes.

Parameters **$options** – See *Supported Options* section for more information.

287.2 ExcludeExtension

`Zend\Validator\File\ExcludeExtension` checks the extension of files. It will assert `false` when a given file has one the a defined extensions.

This validator is inversely related to the *Extension validator*.

Please refer to the *Extension validator* for options and usage examples.

287.3 ExcludeMimeType

`Zend\Validator\File\ExcludeMimeType` checks the MIME type of files. It will assert `false` when a given file has one the a defined MIME types.

This validator is inversely related to the *MimeType validator*.

Please refer to the *MimeType validator* for options and usage examples.

287.4 Exists

`Zend\Validator\File\Exists` checks for the existence of files in specified directories.

This validator is inversely related to the *NotExists validator*.

Supported Options

The following set of options are supported:

- **directory (string|array)** Comma-delimited string (or array) of directories.

Usage Examples

```
// Only allow files that exist in ~both~ directories
$validator = new \Zend\Validator\File\Exists('/tmp,/var/tmp');

// ...or with array notation
$validator = new \Zend\Validator\File\Exists(array('/tmp', '/var/tmp'));

// Perform validation
if ($validator->isValid('/tmp/myfile.txt')) {
    // file is valid
}
```

Note: This validator checks whether the specified file exists in **all** of the given directories. The validation will fail if the file does not exist in one (or more) of the given directories.

287.5 Extension

`Zend\Validator\File\Extension` checks the extension of files. It will assert `true` when a given file has one the a defined extensions.

This validator is inversely related to the *ExcludeExtension validator*.

Supported Options

The following set of options are supported:

- **extension (string|array)** Comma-delimited string (or array) of extensions to test against.
- **case (boolean) default: "false"** Should comparison of extensions be case-sensitive?

Usage Examples

```
// Allow files with 'php' or 'exe' extensions
$validator = new \Zend\Validator\File\Extension('php,exe');

// ...or with array notation
$validator = new \Zend\Validator\File\Extension(array('php', 'exe'));

// Test with case-sensitivity on
$validator = new \Zend\Validator\File\Extension(array('php', 'exe'), true);

// Perform validation
if ($validator->isValid('./myfile.php')) {
    // file is valid
}
```

Public Methods

addExtension (*string|array $options*)
Adds extension(s) via a comma-delimited string or an array.

287.6 Hash

`Zend\Validator\File\Hash` allows you to validate if a given file's hashed contents matches the supplied hash(es) and algorithm(s).

Note: This validator requires the Hash extension from PHP. A list of supported hash algorithms can be found with the hash_algos() function.

Supported Options

The following set of options are supported:

- **hash (string)** Hash to test the file against.
- **algorithm (string) default: "crc32"** Algorithm to use for the hashing validation.

Usage Examples

```
// Does file have the given hash?
$validator = new \Zend\Validator\File\Hash('3b3652f', 'crc32');

// Or, check file against multiple hashes
$validator = new \Zend\Validator\File\Hash(array('3b3652f', 'e612b69'), 'crc32');

// Perform validation with file path
if ($validator->isValid('./myfile.txt')) {
   // file is valid
}
```

Public Methods

getHash()
Returns the current set of hashes.

Return type `array`

addHash(*string|array $options*)
Adds a hash for one or multiple files to the internal set of hashes.

Parameters **$options** – See *Supported Options* section for more information.

setHash(*string|array $options*)
Sets a hash for one or multiple files. Removes any previously set hashes.

Parameters **$options** – See *Supported Options* section for more information.

287.7 ImageSize

`Zend\Validator\File\ImageSize` checks the size of image files. Minimum and/or maximum dimensions can be set to validate against.

Supported Options

The following set of options are supported:

- **minWidth** `(int|null)` `default:` `null`
- **minHeight** `(int|null)` `default:` `null`
- **maxWidth** `(int|null)` `default:` `null`
- **maxHeight** `(int|null)` `default:` `null` To bypass validation of a particular dimension, the relevant option should be set to `null`.

Usage Examples

```
// Is image size between 320x200 (min) and 640x480 (max)?
$validator = new \Zend\Validator\File\ImageSize(320, 200, 640, 480);

// ...or with array notation
$validator = new \Zend\Validator\File\ImageSize(array(
    'minWidth' => 320, 'minHeight' => 200,
    'maxWidth' => 640, 'maxHeight' => 480,
));

// Is image size equal to or larger than 320x200?
$validator = new \Zend\Validator\File\ImageSize(array(
    'minWidth' => 320, 'minHeight' => 200,
));

// Is image size equal to or smaller than 640x480?
$validator = new \Zend\Validator\File\ImageSize(array(
    'maxWidth' => 640, 'maxHeight' => 480,
));

// Perform validation with file path
if ($validator->isValid('./myfile.jpg')) {
    // file is valid
}
```

Public Methods

getImageMin()
Returns the minimum dimensions (width and height)

Return type `array`

getImageMax()
Returns the maximum dimensions (width and height)

Return type `array`

287.8 IsCompressed

`Zend\Validator\File\IsCompressed` checks if a file is a compressed archive, such as zip or gzip. This validator is based on the *MimeType validator* and supports the same methods and options.

The default list of compressed file MIME types can be found in the source code.

Please refer to the *MimeType validator* for options and public methods.

Usage Examples

```
$validator = new \Zend\Validator\File\IsCompressed();
if ($validator->isValid('./myfile.zip')) {
    // file is valid
}
```

287.9 IsImage

`Zend\Validator\File\IsImage` checks if a file is an image, such as *jpg* or *png*. This validator is based on the *MimeType validator* and supports the same methods and options.

The default list of image file MIME types can be found in the source code.

Please refer to the *MimeType validator* for options and public methods.

Usage Examples

```
$validator = new \Zend\Validator\File\IsImage();
if ($validator->isValid('./myfile.jpg')) {
    // file is valid
}
```

287.10 Md5

`Zend\Validator\File\Md5` allows you to validate if a given file's hashed contents matches the supplied md5 hash(es). It is subclassed from the *Hash validator* to provide a convenient validator that only supports the `md5` algorithm.

Note: This validator requires the Hash extension from PHP with the `md5` algorithm.

Supported Options

The following set of options are supported:

- **hash (`string`)** Hash to test the file against.

Usage Examples

```
// Does file have the given hash?
$validator = new \Zend\Validator\File\Md5('3b3652f336522365223');

// Or, check file against multiple hashes
$validator = new \Zend\Validator\File\Md5(array(
```

```
    '3b3652f336522365223', 'eb3365f3365ddc65365'
));

// Perform validation with file path
if ($validator->isValid('./myfile.txt')) {
    // file is valid
}
```

Public Methods

getMd5()
Returns the current set of md5 hashes.

Return type array

addMd5(*string|array $options*)
Adds a md5 hash for one or multiple files to the internal set of hashes.

Parameters **$options** – See *Supported Options* section for more information.

setMd5(*string|array $options*)
Sets a md5 hash for one or multiple files. Removes any previously set hashes.

Parameters **$options** – See *Supported Options* section for more information.

287.11 MimeType

`Zend\Validator\File\MimeType` checks the MIME type of files. It will assert `true` when a given file has one the a defined MIME types.

This validator is inversely related to the *ExcludeMimeType validator*.

Note: This component will use the `FileInfo` extension if it is available. If it's not, it will degrade to the `mime_content_type()` function. And if the function call fails it will use the MIME type which is given by HTTP. You should be aware of possible security problems when you do not have `FileInfo` or `mime_content_type()` available. The MIME type given by HTTP is not secure and can be easily manipulated.

Supported Options

The following set of options are supported:

- **mimeType (string|array)** Comma-delimited string (or array) of MIME types to test against.
- **magicFile (string|null) default: "MAGIC" constant** Specify the location of the magicfile to use. By default the `MAGIC` constant value will be used.
- **enableHeaderCheck (boolean) default: "false"** Check the HTTP Information for the file type when the fileInfo or mimeMagic extensions can not be found.

Usage Examples

```
// Only allow 'gif' or 'jpg' files
$validator = new \Zend\Validator\File\MimeType('image/gif,image/jpg');

// ...or with array notation
$validator = new \Zend\Validator\File\MimeType(array('image/gif', 'image/jpg'));

// ...or restrict an entire group of types
$validator = new \Zend\Validator\File\MimeType(array('image', 'audio'));

// Use a different magicFile
$validator = new \Zend\Validator\File\MimeType(array(
    'image/gif', 'image/jpg',
    'magicFile' => '/path/to/magicfile.mgx'
));

// Use the HTTP information for the file type
$validator = new \Zend\Validator\File\MimeType(array(
    'image/gif', 'image/jpg',
    'enableHeaderCheck' => true
));

// Perform validation
if ($validator->isValid('./myfile.jpg')) {
    // file is valid
}
```

Warning: Allowing "groups" of MIME types will accept **all** members of this group even if your application does not support them. When you allow 'image' you also allow 'image/xpixmap' and 'image/vasa' which could be problematic.

287.12 NotExists

`Zend\Validator\File\NotExists` checks for the existence of files in specified directories.

This validator is inversely related to the *Exists validator*.

Supported Options

The following set of options are supported:

- **directory (`string|array`)** Comma-delimited string (or array) of directories.

Usage Examples

```
// Only allow files that do not exist in ~either~ directories
$validator = new \Zend\Validator\File\NotExists('/tmp,/var/tmp');

// ...or with array notation
$validator = new \Zend\Validator\File\NotExists(array('/tmp', '/var/tmp'));

```

```
// Perform validation
if ($validator->isValid('/home/myfile.txt')) {
    // file is valid
}
```

Note: This validator checks whether the specified file does not exist in **any** of the given directories. The validation will fail if the file exists in one (or more) of the given directories.

287.13 Sha1

`Zend\Validator\File\Sha1` allows you to validate if a given file's hashed contents matches the supplied sha1 hash(es). It is subclassed from the *Hash validator* to provide a convenient validator that only supports the `sha1` algorithm.

Note: This validator requires the Hash extension from PHP with the `sha1` algorithm.

Supported Options

The following set of options are supported:

- **hash (string)** Hash to test the file against.

Usage Examples

```
// Does file have the given hash?
$validator = new \Zend\Validator\File\Sha1('3b3652f336522365223');

// Or, check file against multiple hashes
$validator = new \Zend\Validator\File\Sha1(array(
    '3b3652f336522365223', 'eb3365f3365ddc65365'
));

// Perform validation with file path
if ($validator->isValid('./myfile.txt')) {
    // file is valid
}
```

Public Methods

getSha1()

Returns the current set of sha1 hashes.

Return type `array`

addSha1(*string|array $options*)

Adds a sha1 hash for one or multiple files to the internal set of hashes.

Parameters **$options** – See *Supported Options* section for more information.

setSha1 (*string|array $options*)
Sets a sha1 hash for one or multiple files. Removes any previously set hashes.

Parameters **$options** – See *Supported Options* section for more information.

287.14 Size

`Zend\Validator\File\Size` checks for the size of a file.

Supported Options

The following set of options are supported:

- **min** `(integer|string) default: null`
- **max** **(integer|string) default: null** The *integer* number of bytes, or a *string* in SI notation (ie. 1kB, 2MB, 0.2GB).

  The accepted SI notation units are: kB, MB, GB, TB, PB, and EB. All sizes are converted using 1024 as the base value (ie. 1kB == 1024 bytes, 1MB == 1024kB).
- **useByteString** **(boolean) default: true** Display error messages with size in user-friendly number or with the plain byte size.

Usage Examples

```
// Limit the file size to 40000 bytes
$validator = new \Zend\Validator\File\Size(40000);

// Limit the file size to between 10kB and 4MB
$validator = new \Zend\Validator\File\Size(array(
    'min' => '10kB', 'max' => '4MB'
));

// Perform validation with file path
if ($validator->isValid('./myfile.txt')) {
    // file is valid
}
```

287.15 UploadFile

`Zend\Validator\File\UploadFile` checks whether a single file has been uploaded via a form `POST` and will return descriptive messages for any upload errors.

Note: *Zend\InputFilter\FileInput* will automatically prepend this validator in it's validation chain.

Usage Examples

```
use Zend\Http\PhpEnvironment\Request;

$request = new Request();
$files   = $request->getFiles();
// i.e. $files['my-upload']['error'] == 0

$validator = \Zend\Validator\File\UploadFile();
if ($validator->isValid($files['my-upload'])) {
    // file is valid
}
```

287.16 WordCount

`Zend\Validator\File\WordCount` checks for the number of words within a file.

Supported Options

The following set of options are supported:

- **min** `(integer) default: null`
- **max `(integer) default: null`** The number of words allowed.

Usage Examples

```
// Limit the amount of words to a maximum of 2000
$validator = new \Zend\Validator\File\WordCount(2000);

// Limit the amount of words to between 100 and 5000
$validator = new \Zend\Validator\File\WordCount(100, 5000);

// ... or with array notation
$validator = new \Zend\Validator\File\WordCount(array(
    'min' => 1000, 'max' => 5000
));

// Perform validation with file path
if ($validator->isValid('./myfile.txt')) {
    // file is valid
}
```

CHAPTER

EIGHT

VALIDATOR CHAINS

Often multiple validations should be applied to some value in a particular order. The following code demonstrates a way to solve the example from the *introduction*, where a username must be between 6 and 12 alphanumeric characters:

```
// Create a validator chain and add validators to it
$validatorChain = new Zend\Validator\ValidatorChain();
$validatorChain->attach(
                    new Zend\Validator\StringLength(array('min' => 6,
                                                          'max' => 12)))
               ->attach(new Zend\Validator\Alnum());

// Validate the username
if ($validatorChain->isValid($username)) {
    // username passed validation
} else {
    // username failed validation; print reasons
    foreach ($validatorChain->getMessages() as $message) {
        echo "$message\n";
    }
}
```

Validators are run in the order they were added to `Zend\Validator\ValidatorChain`. In the above example, the username is first checked to ensure that its length is between 6 and 12 characters, and then it is checked to ensure that it contains only alphanumeric characters. The second validation, for alphanumeric characters, is performed regardless of whether the first validation, for length between 6 and 12 characters, succeeds. This means that if both validations fail, `getMessages()` will return failure messages from both validators.

In some cases it makes sense to have a validator break the chain if its validation process fails. `Zend\Validator\ValidatorChain` supports such use cases with the second parameter to the `attach()` method. By setting `$breakChainOnFailure` to `TRUE`, the added validator will break the chain execution upon failure, which avoids running any other validations that are determined to be unnecessary or inappropriate for the situation. If the above example were written as follows, then the alphanumeric validation would not occur if the string length validation fails:

```
$validatorChain->attach(
                    new Zend\Validator\StringLength(array('min' => 6,
                                                          'max' => 12)),
                    true)
               ->attach(new Zend\Validator\Alnum());
```

Any object that implements `Zend\Validator\ValidatorInterface` may be used in a validator chain.

CHAPTER

NINE

WRITING VALIDATORS

`Zend\Validator` supplies a set of commonly needed validators, but inevitably, developers will wish to write custom validators for their particular needs. The task of writing a custom validator is described in this section.

`Zend\Validator\ValidatorInterface` defines two methods, `isValid()` and `getMessages()`, that may be implemented by user classes in order to create custom validation objects. An object that implements `Zend\Validator\AbstractValidator` interface may be added to a validator chain with `Zend\Validator\ValidatorChain::addValidator()`. Such objects may also be used with *Zend\Filter\Input*.

As you may already have inferred from the above description of `Zend\Validator\ValidatorInterface`, validation classes provided with Zend Framework return a boolean value for whether or not a value validates successfully. They also provide information about **why** a value failed validation. The availability of the reasons for validation failures may be valuable to an application for various purposes, such as providing statistics for usability analysis.

Basic validation failure message functionality is implemented in `Zend\Validator\AbstractValidator`. To include this functionality when creating a validation class, simply extend `Zend\Validator\AbstractValidator`. In the extending class you would implement the `isValid()` method logic and define the message variables and message templates that correspond to the types of validation failures that can occur. If a value fails your validation tests, then `isValid()` should return `FALSE`. If the value passes your validation tests, then `isValid()` should return `TRUE`.

In general, the `isValid()` method should not throw any exceptions, except where it is impossible to determine whether or not the input value is valid. A few examples of reasonable cases for throwing an exception might be if a file cannot be opened, an *LDAP* server could not be contacted, or a database connection is unavailable, where such a thing may be required for validation success or failure to be determined.

Creating a Simple Validation Class

The following example demonstrates how a very simple custom validator might be written. In this case the validation rules are simply that the input value must be a floating point value.

```
class MyValid\Float extends Zend\Validator\AbstractValidator
{
    const FLOAT = 'float';

    protected $messageTemplates = array(
        self::FLOAT => "'%value%' is not a floating point value"
    );

    public function isValid($value)
    {
        $this->setValue($value);
```

```

        if (!is_float($value)) {
            $this->error(self::FLOAT);
            return false;
        }

        return true;
    }
}
```

The class defines a template for its single validation failure message, which includes the built-in magic parameter, **%value%**. The call to `setValue()` prepares the object to insert the tested value into the failure message automatically, should the value fail validation. The call to `error()` tracks a reason for validation failure. Since this class only defines one failure message, it is not necessary to provide `error()` with the name of the failure message template.

Writing a Validation Class having Dependent Conditions

The following example demonstrates a more complex set of validation rules, where it is required that the input value be numeric and within the range of minimum and maximum boundary values. An input value would fail validation for exactly one of the following reasons:

- The input value is not numeric.
- The input value is less than the minimum allowed value.
- The input value is more than the maximum allowed value.

These validation failure reasons are then translated to definitions in the class:

```
class MyValid\NumericBetween extends Zend\Validator\AbstractValidator
{
    const MSG_NUMERIC = 'msgNumeric';
    const MSG_MINIMUM = 'msgMinimum';
    const MSG_MAXIMUM = 'msgMaximum';

    public $minimum = 0;
    public $maximum = 100;

    protected $messageVariables = array(
        'min' => 'minimum',
        'max' => 'maximum'
    );

    protected $messageTemplates = array(
        self::MSG_NUMERIC => "'%value%' is not numeric",
        self::MSG_MINIMUM => "'%value%' must be at least '%min%'",
        self::MSG_MAXIMUM => "'%value%' must be no more than '%max%'"
    );

    public function isValid($value)
    {
        $this->setValue($value);

        if (!is_numeric($value)) {
            $this->error(self::MSG_NUMERIC);
            return false;
        }

```

```
        if ($value < $this->minimum) {
            $this->error(self::MSG_MINIMUM);
            return false;
        }

        if ($value > $this->maximum) {
            $this->error(self::MSG_MAXIMUM);
            return false;
        }

        return true;
    }
}
```

The public properties `$minimum` and `$maximum` have been established to provide the minimum and maximum boundaries, respectively, for a value to successfully validate. The class also defines two message variables that correspond to the public properties and allow `min` and `max` to be used in message templates as magic parameters, just as with `value`.

Note that if any one of the validation checks in `isValid()` fails, an appropriate failure message is prepared, and the method immediately returns `FALSE`. These validation rules are therefore sequentially dependent. That is, if one test should fail, there is no need to test any subsequent validation rules. This need not be the case, however. The following example illustrates how to write a class having independent validation rules, where the validation object may return multiple reasons why a particular validation attempt failed.

Validation with Independent Conditions, Multiple Reasons for Failure

Consider writing a validation class for password strength enforcement - when a user is required to choose a password that meets certain criteria for helping secure user accounts. Let us assume that the password security criteria enforce that the password:

- is at least 8 characters in length,
- contains at least one uppercase letter,
- contains at least one lowercase letter,
- and contains at least one digit character.

The following class implements these validation criteria:

```
class MyValid\PasswordStrength extends Zend\Validator\AbstractValidator
{
    const LENGTH = 'length';
    const UPPER  = 'upper';
    const LOWER  = 'lower';
    const DIGIT  = 'digit';

    protected $messageTemplates = array(
        self::LENGTH => "'%value%' must be at least 8 characters in length",
        self::UPPER  => "'%value%' must contain at least one uppercase letter",
        self::LOWER  => "'%value%' must contain at least one lowercase letter",
        self::DIGIT  => "'%value%' must contain at least one digit character"
    );

    public function isValid($value)
    {
        $this->setValue($value);
```

```

        $isValid = true;

        if (strlen($value) < 8) {
            $this->error(self::LENGTH);
            $isValid = false;
        }

        if (!preg_match('/[A-Z]/', $value)) {
            $this->error(self::UPPER);
            $isValid = false;
        }

        if (!preg_match('/[a-z]/', $value)) {
            $this->error(self::LOWER);
            $isValid = false;
        }

        if (!preg_match('/\d/', $value)) {
            $this->error(self::DIGIT);
            $isValid = false;
        }

        return $isValid;
    }
}
```

Note that the four criteria tests in `isValid()` do not immediately return `FALSE`. This allows the validation class to provide **all** of the reasons that the input password failed to meet the validation requirements. if, for example, a user were to input the string "#$%" as a password, `isValid()` would cause all four validation failure messages to be returned by a subsequent call to `getMessages()`.

CHAPTER

VALIDATION MESSAGES

Each validator which is based on `Zend\Validator\ValidatorInterface` provides one or multiple messages in the case of a failed validation. You can use this information to set your own messages, or to translate existing messages which a validator could return to something different.

These validation messages are constants which can be found at top of each validator class. Let's look into `Zend\Validator\GreaterThan` for an descriptive example:

```
protected $messageTemplates = array(
    self::NOT_GREATER => "'%value%' is not greater than '%min%'",
);
```

As you can see the constant `self::NOT_GREATER` refers to the failure and is used as key, and the message itself is used as value of the message array.

You can retrieve all message templates from a validator by using the `getMessageTemplates()` method. It returns you the above array which contains all messages a validator could return in the case of a failed validation.

```
$validator = new Zend\Validator\GreaterThan();
$messages  = $validator->getMessageTemplates();
```

Using the `setMessage()` method you can set another message to be returned in case of the specified failure.

```
$validator = new Zend\Validator\GreaterThan();
$validator->setMessage(
    'Please enter a lower value',
    Zend\Validator\GreaterThan::NOT_GREATER
);
```

The second parameter defines the failure which will be overridden. When you omit this parameter, then the given message will be set for all possible failures of this validator.

290.1 Using pre-translated validation messages

Zend Framework is shipped with more than 45 different validators with more than 200 failure messages. It can be a tedious task to translate all of these messages. But for your convenience Zend Framework comes with already pre-translated validation messages. You can find them within the path `/resources/languages` in your Zend Framework installation.

Note: Used path

The resource files are outside of the library path because all of your translations should also be outside of this path.

So to translate all validation messages to German for example, all you have to do is to attach a translator to `Zend\Validator\AbstractValidator` using these resource files.

```
$translator = new Zend\I18n\Translator\Translator();
$translator->addTranslationFile(
    'phpArray',
    'resources/languages/en.php',
    'default',
    'en_US'
);
Zend\Validator\AbstractValidator::setDefaultTranslator($translator);
```

Note: Supported languages

This feature is very young, so the amount of supported languages may not be complete. New languages will be added with each release. Additionally feel free to use the existing resource files to make your own translations.

You could also use these resource files to rewrite existing translations. So you are not in need to create these files manually yourself.

290.2 Limit the size of a validation message

Sometimes it is necessary to limit the maximum size a validation message can have. For example when your view allows a maximum size of 100 chars to be rendered on one line. To simplify the usage, `Zend\Validator\AbstractValidator` is able to automatically limit the maximum returned size of a validation message.

To get the actual set size use `Zend\Validator\AbstractValidator::getMessageLength()`. If it is -1, then the returned message will not be truncated. This is default behaviour.

To limit the returned message size use `Zend\Validator\AbstractValidator::setMessageLength()`. Set it to any integer size you need. When the returned message exceeds the set size, then the message will be truncated and the string '**...**' will be added instead of the rest of the message.

```
Zend\Validator\AbstractValidator::setMessageLength(100);
```

Note: Where is this parameter used?

The set message length is used for all validators, even for self defined ones, as long as they extend `Zend\Validator\AbstractValidator`.

CHAPTER

ONE

GETTING THE ZEND FRAMEWORK VERSION

`Zend\Version` provides a class constant `Zend\Version\Version::VERSION` that contains a string identifying the version number of your Zend Framework installation. `Zend\Version\Version::VERSION` might contain "1.7.4", for example.

The static method `Zend\Version\Version::compareVersion($version)` is based on the *PHP* function version_compare(). This method returns -1 if the specified version is older than the installed Zend Framework version, 0 if they are the same and +1 if the specified version is newer than the version of the Zend Framework installation.

Example of the compareVersion() Method

```
// returns -1, 0 or 1
$cmp = Zend\Version\Version::compareVersion('2.0.0');
```

The static method `Zend\Version\Version::getLatest()` provides the version number of the last stable release available for download on the site Zend Framework.

Example of the getLatest() Method

```
// returns 1.11.0 (or a later version)
echo Zend\Version\Version::getLatest();
```

CHAPTER

TWO

ZEND\VIEW QUICK START

CHAPTER

THREE

OVERVIEW

`Zend\View` provides the "View" layer of Zend Framework 2's MVC system. It is a multi-tiered system allowing a variety of mechanisms for extension, substitution, and more.

The components of the view layer are as follows:

- **Variables Containers** hold variables and callbacks that you wish to represent in the view. Often-times, a Variables Container will also provide mechanisms for context-specific escaping of variables and more.
- **View Models** hold Variables Containers, specify the template to use (if any), and optionally provide rendering options (more on that below). View Models may be nested in order to represent complex structures.
- **Renderers** take View Models and provide a representation of them to return. Zend Framework 2 ships with three renderers by default: a `PhpRenderer` which utilizes PHP templates in order to generate markup, a `JsonRenderer`, and a `FeedRenderer` for generating RSS and Atom feeds.
- **Resolvers** utilizes Resolver Strategies to resolve a template name to a resource a Renderer may consume. As an example, a Resolver may take the name "blog/entry" and resolve it to a PHP view script.
- The **View** consists of strategies that map the current Request to a Renderer, and strategies for injecting the result of rendering to the Response.
- **Rendering Strategies** listen to the `Zend\View\ViewEvent::EVENT_RENDERER` event of the View and decide which Renderer should be selected based on the Request or other criteria.
- **Response Strategies** are used to inject the Response object with the results of rendering. That may also include taking actions such as setting Content-Type headers.

Additionally, Zend Framework 2 provides integration with the MVC via a number of event listeners in the `Zend\Mvc\View` namespace.

CHAPTER

FOUR

USAGE

This section of the manual is designed to show you typical usage patterns of the view layer when using it within the Zend Framework 2 MVC. The assumptions are that you are using *Dependency Injection* and the default MVC view strategies.

294.1 Configuration

The default configuration will typically work out-of-the-box. However, you will still need to select Resolver Strategies and configure them, as well as potentially indicate alternate template names for things like the site layout, 404 (not found) pages, and error pages. The code snippets below can be added to your configuration to accomplish this. We recommend adding it to a site-specific module, such as the "Application" module from the framework's ZendSkeletonApplication, or to one of your autoloaded configurations within the `config/autoload/` directory.

```
return array(
    'view_manager' => array(
        // The TemplateMapResolver allows you to directly map template names
        // to specific templates. The following map would provide locations
        // for a home page template ("application/index/index"), as well as for
        // the layout ("layout/layout"), error pages ("error/index"), and
        // 404 page ("error/404"), resolving them to view scripts.
        'template_map' => array(
            'application/index/index' => __DIR__ .  '/../view/application/index/index.phtml',
            'site/layout'             => __DIR__ . '/../view/layout/layout.phtml',
            'error/index'             => __DIR__ . '/../view/error/index.phtml',
            'error/404'               => __DIR__ . '/../view/error/404.phtml',
        ),

        // The TemplatePathStack takes an array of directories. Directories
        // are then searched in LIFO order (it's a stack) for the requested
        // view script. This is a nice solution for rapid application
        // development, but potentially introduces performance expense in
        // production due to the number of static calls necessary.
        //
        // The following adds an entry pointing to the view directory
        // of the current module. Make sure your keys differ between modules
        // to ensure that they are not overwritten -- or simply omit the key!
        'template_path_stack' => array(
            'application' => __DIR__ . '/../view',
        ),

        // Set the template name for the site's layout.
        //
```

```
        // By default, the MVC's default Rendering Strategy uses the
        // template name "layout/layout" for the site's layout.
        // Here, we tell it to use the "site/layout" template,
        // which we mapped via the TemplateMapResolver above.
        'layout' => 'site/layout',

        // By default, the MVC registers an "exception strategy", which is
        // triggered when a requested action raises an exception; it creates
        // a custom view model that wraps the exception, and selects a
        // template. We'll set it to "error/index".
        //
        // Additionally, we'll tell it that we want to display an exception
        // stack trace; you'll likely want to disable this by default.
        'display_exceptions' => true,
        'exception_template' => 'error/index',

       // Another strategy the MVC registers by default is a "route not
       // found" strategy. Basically, this gets triggered if (a) no route
       // matches the current request, (b) the controller specified in the
       // route match cannot be found in the service locator, (c) the controller
       // specified in the route match does not implement the DispatchableInterface
       // interface, or (d) if a response from a controller sets the
       // response status to 404.
       //
       // The default template used in such situations is "error", just
       // like the exception strategy. Here, we tell it to use the "error/404"
       // template (which we mapped via the TemplateMapResolver, above).
       //
       // You can opt in to inject the reason for a 404 situation; see the
       // various `Application\:\:ERROR_*`_ constants for a list of values.
       // Additionally, a number of 404 situations derive from exceptions
       // raised during routing or dispatching. You can opt-in to display
       // these.
       'display_not_found_reason' => true,
       'not_found_template'       => 'error/404',
    ),
);
```

294.2 Controllers and View Models

`Zend\View\View` consumes `ViewModels`, passing them to the selected renderer. Where do you create these, though?

The most explicit way is to create them in your controllers and return them.

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class BazBatController extends AbstractActionController
{
    public function doSomethingCrazyAction()
    {
        $view = new ViewModel(array(
            'message' => 'Hello world',
```

```
        ));
        $view->setTemplate('foo/baz-bat/do-something-crazy');
        return $view;
    }
}
```

This sets a "message" variable in the View Model, and sets the template name "foo/baz-bat/do-something-crazy". The View Model is then returned.

In most cases, you'll likely have a template name based on the module namespace, controller, and action. Considering that, and if you're simply passing some variables, could this be made simpler? Definitely.

The MVC registers a couple of listeners for controllers to automate this. The first will look to see if you returned an associative array from your controller; if so, it will create a View Model and make this associative array the Variables Container; this View Model then replaces the *MvcEvent*'s result. It will also look to see if you returned nothing or null; if so, it will create a View Model without any variables attached; this View Model also replaces the `MvcEvent`'s result.

The second listener checks to see if the `MvcEvent` result is a View Model, and, if so, if it has a template associated with it. If not, it will inspect the controller matched during routing to determine the module namespace and the controller class name, and, if available, it's "action" parameter in order to create a template name. This will be "module/controller/action", all normalized to lowercase, dash-separated words.

As an example, the controller `Foo\Controller\BazBatController` with action "doSomethingCrazyAction", would be mapped to the template `foo/baz-bat/do-something-crazy`. As you can see, the words "Controller" and "Action" are omitted.

In practice, that means our previous example could be re-written as follows:

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class BazBatController extends AbstractActionController
{
    public function doSomethingCrazyAction()
    {
        return array(
            'message' => 'Hello world',
        );
    }
}
```

The above method will likely work for the majority of use cases. When you need to specify a different template, explicitly create and return a View Model and specify the template manually, as in the first example.

294.3 Nesting View Models

The other use case you may have for setting explicit View Models is if you wish to **nest** them. In other words, you might want to render templates to be included within the main View you return.

As an example, you may want the View from an action to be one primary section that includes both an "article" and a couple of sidebars; one of the sidebars may include content from multiple Views as well:

```
namespace Content\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
```

```

class ArticleController extends AbstractActionController
{
    public function viewAction()
    {
        // get the article from the persistence layer, etc...

        $view = new ViewModel();

        // this is not needed since it matches "module/controller/action"
        $view->setTemplate('content/article/view');

        $articleView = new ViewModel(array('article' => $article));
        $articleView->setTemplate('content/article');

        $primarySidebarView = new ViewModel();
        $primarySidebarView->setTemplate('content/main-sidebar');

        $secondarySidebarView = new ViewModel();
        $secondarySidebarView->setTemplate('content/secondary-sidebar');

        $sidebarBlockView = new ViewModel();
        $sidebarBlockView->setTemplate('content/block');

        $secondarySidebarView->addChild($sidebarBlockView, 'block');

        $view->addChild($articleView, 'article')
             ->addChild($primarySidebarView, 'sidebar_primary')
             ->addChild($secondarySidebarView, 'sidebar_secondary');

        return $view;
    }
}
```

The above will create and return a View Model specifying the template "content/article/view". When the View is rendered, it will render three child Views, the `$articleView`, `$primarySidebarView`, and `$secondarySidebarView`; these will be captured to the `$view`'s "article", "sidebar_primary", and "sidebar_secondary" variables, respectively, so that when it renders, you may include that content. Additionally, the `$secondarySidebarView` will include an additional View Model, `$sidebarBlockView`, which will be captured to its "block" view variable.

To better visualize this, let's look at what the final content might look like, with comments detailing where each nested view model is injected.

Here are the templates, rendered based on a 12-column grid:

```
<?php // "content/article/view" template ?>
<!-- This is from the $view View Model, and the "content/article/view" template -->
<div class="row content">
    <?php echo $this->article ?>

    <?php echo $this->sidebar_primary ?>

    <?php echo $this->sidebar_secondary ?>
</div>
```

```
<?php // "content/article" template ?>
    <!-- This is from the $articleView View Model, and the "content/article"
         template -->
```

```
    <article class="span8">
        <?php echo $this->escapeHtml('article') ?>
    </article>
```

```
<?php // "content/main-sidebar" template ?>
    <!-- This is from the $primarySidebarView View Model, and the
         "content/main-sidebar" template -->
    <div class="span2 sidebar">
        sidebar content...
    </div>
```

```
<?php // "content/secondary-sidebar template ?>
    <!-- This is from the $secondarySidebarView View Model, and the
         "content/secondary-sidebar" template -->
    <div class="span2 sidebar pull-right">
        <?php echo $this->block ?>
    </div>
```

```
<?php // "content/block template ?>
        <!-- This is from the $sidebarBlockView View Model, and the
            "content/block" template -->
        <div class="block">
            block content...
        </div>
```

And here is the aggregate, generated content:

```
<!-- This is from the $view View Model, and the "content/article/view" template -->
<div class="row content">
    <!-- This is from the $articleView View Model, and the "content/article"
         template -->
    <article class="span8">
        Lorem ipsum ....
    </article>

    <!-- This is from the $primarySidebarView View Model, and the
         "content/main-sidebar" template -->
    <div class="span2 sidebar">
        sidebar content...
    </div>

    <!-- This is from the $secondarySidebarView View Model, and the
         "content/secondary-sidebar" template -->
    <div class="span2 sidebar pull-right">
        <!-- This is from the $sidebarBlockView View Model, and the
            "content/block" template -->
        <div class="block">
            block content...
        </div>
    </div>
</div>
```

As you can see, you can achieve very complex markup using nested Views, while simultaneously keeping the details of rendering isolated from the Request/Response lifecycle of the controller.

294.4 Dealing with Layouts

Most sites enforce a cohesive look-and-feel which we typically call the site's "layout". It includes the default stylesheets and JavaScript necessary, if any, as well as the basic markup structure into which all site content will be injected.

Within Zend Framework 2, layouts are handled via nesting of View Models (see the *previous example* for examples of View Model nesting). The `Zend\Mvc\View\Http\ViewManager` composes a View Model which acts as the "root" for nested View Models. As such, it should contain the skeleton (or layout) template for the site. All other content is then rendered and captured to view variables of this root View Model.

The `ViewManager` sets the layout template as "layout/layout" by default. To change this, you can add some configuration to the "view_manager" area of your *configuration*.

A listener on the controllers, `Zend\Mvc\View\Http\InjectViewModelListener`, will take a View Model returned from a controller and inject it as a child of the root (layout) View Model. By default, View Models will capture to the "content" variable of the root View Model. This means you can do the following in your layout view script:

```
<html>
    <head>
        <title><?php echo $this->headTitle() ?></title>
    </head>
    <body>
        <?php echo $this->content; ?>
    </body>
</html>
```

If you want to specify a different View variable for which to capture, explicitly create a view model in your controller, and set its "capture to" value:

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class BazBatController extends AbstractActionController
{
    public function doSomethingCrazyAction()
    {
        $view = new ViewModel(array(
            'message' => 'Hello world',
        ));

        // Capture to the layout view's "article" variable
        $view->setCaptureTo('article');

        return $view;
    }
}
```

There will be times you don't want to render a layout. For example, you might be answering an API call which expects JSON or an XML payload, or you might be answering an XHR request that expects a partial HTML payload. The simplest way to do this is to explicitly create and return a view model from your controller, and mark it as "terminal", which will hint to the MVC listener that normally injects the returned View Model into the layout View Model, to instead replace the layout view model.

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class BazBatController extends AbstractActionController
{
    public function doSomethingCrazyAction()
    {
        $view = new ViewModel(array(
            'message' => 'Hello world',
        ));

        // Disable layouts; `MvcEvent` will use this View Model instead
        $view->setTerminal(true);

        return $view;
    }
}
```

When discussing nesting View Models, we detailed a nested View Model which contained an article and sidebars. Sometimes, you may want to provide additional View Models to the layout, instead of nesting in the returned layout. This may be done by using the "layout" controller plugin, which returns the root View Model. You can then call the same `addChild()` method on it as we did in that previous example.

```
namespace Content\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class ArticleController extends AbstractActionController
{
    public function viewAction()
    {
        // get the article from the persistence layer, etc...

        // Get the "layout" view model and inject a sidebar
        $layout = $this->layout();
        $sidebarView = new ViewModel();
        $sidebarView->setTemplate('content/sidebar');
        $layout->addChild($sidebarView, 'sidebar');

        // Create and return a view model for the retrieved article
        $view = new ViewModel(array('article' => $article));
        $view->setTemplate('content/article');
        return $view;
    }
}
```

You could also use this technique to select a different layout, by simply calling the `setTemplate()` method of the layout View Model:

```
//In a controller
namespace Content\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class ArticleController extends AbstractActionController
```

```
{
    public function viewAction()
    {
        // get the article from the persistence layer, etc...

        // Get the "layout" view model and set an alternate template
        $layout = $this->layout();
        $layout->setTemplate('article/layout');

        // Create and return a view model for the retrieved article
        $view = new ViewModel(array('article' => $article));
        $view->setTemplate('content/article');
        return $view;
    }
}
```

Sometimes, you may want to access the layout from within your actual view scripts when using the `PhpRenderer`. Reasons might include wanting to change the layout template or wanting to either access or inject layout view variables. Similar to the "layout" controller plugin, you can use the "layout" View Helper. If you provide a string argument to it, you will change the template; if you provide no arguments, the root layout View Model is returned.

Commonly, you may want to alter the layout based on the current **module**. This requires (a) detecting if the controller matched in routing belongs to this module, and then (b) changing the template of the View Model.

The place to do these actions is in a listener. It should listen either to the "route" event at low (negative) priority, or on the "dispatch" event, at any priority. Typically, you will register this during the bootstrap event.

```
namespace Content;

class Module
{
    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function onBootstrap($e)
    {
        // Register a dispatch event
        $app = $e->getParam('application');
        $app->getEventManager()->attach('dispatch', array($this, 'setLayout'));
    }

    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function setLayout($e)
    {
        $matches    = $e->getRouteMatch();
        $controller = $matches->getParam('controller');
        if (false === strpos($controller, __NAMESPACE__)) {
            // not a controller from this module
            return;
        }

        // Set the layout template
        $viewModel = $e->getViewModel();
        $viewModel->setTemplate('content/layout');
    }
```

```
}
```

294.5 Creating and Registering Alternate Rendering and Response Strategies

`Zend\View\View` does very little. Its workflow is essentially to martial a `ViewEvent`, and then trigger two events, "renderer" and "response". You can attach "strategies" to these events, using the methods `addRenderingStrategy()` and `addResponseStrategy()`, respectively. A Rendering Strategy investigates the Request object (or any other criteria) in order to select a Renderer (or fail to select one). A Response Strategy determines how to populate the Response based on the result of rendering.

Zend Framework 2 ships with three Rendering and Response Strategies that you can use within your application.

- `Zend\View\Strategy\PhpRendererStrategy`. This strategy is a "catch-all" in that it will always return the `Zend\View\Renderer\PhpRenderer` and populate the Response body with the results of rendering.
- `Zend\View\Strategy\JsonStrategy`. This strategy inspects the Accept HTTP header, if present, and determines if the client has indicated it accepts an "application/json" response. If so, it will return the `Zend\View\Renderer\JsonRenderer`, and populate the Response body with the JSON value returned, as well as set a Content-Type header with a value of "application/json".
- `Zend\View\Strategy\FeedStrategy`. This strategy inspects the Accept HTTP header, if present, and determines if the client has indicated it accepts either an "application/rss+xml" or "application/atom+xml" response. If so, it will return the `Zend\View\Renderer\FeedRenderer`, setting the feed type to either "rss" or "atom", based on what was matched. Its Response strategy will populate the Response body with the generated feed, as well as set a Content-Type header with the appropriate value based on feed type.

By default, only the `PhpRendererStrategy` is registered, meaning you will need to register the other Strategies yourself if you want to use them. Additionally, it means that you will likely want to register these at higher priority to ensure they match before the `PhpRendererStrategy`. As an example, let's register the `JsonStrategy`:

```
namespace Application;

class Module
{
    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function onBootstrap($e)
    {
        // Register a "render" event, at high priority (so it executes prior
        // to the view attempting to render)
        $app = $e->getApplication();
        $app->getEventManager()->attach('render', array($this, 'registerJsonStrategy'), 100);
    }

    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function registerJsonStrategy($e)
    {
        $app          = $e->getTarget();
        $locator      = $app->getServiceManager();
```

```
        $view         = $locator->get('Zend\View\View');
        $jsonStrategy = $locator->get('ViewJsonStrategy');

        // Attach strategy, which is a listener aggregate, at high priority
        $view->getEventManager()->attach($jsonStrategy, 100);
    }
}
```

The above will register the `JsonStrategy` with the "render" event, such that it executes prior to the `PhpRendererStrategy`, and thus ensure that a JSON payload is created when requested.

What if you want this to happen only in specific modules, or specific controllers? One way is similar to the last example in the *previous section on layouts*, where we detailed changing the layout for a specific module:

```
namespace Content;

class Module
{
    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function onBootstrap($e)
    {
        // Register a render event
        $app = $e->getParam('application');
        $app->getEventManager()->attach('render', array($this, 'registerJsonStrategy'), 100);
    }

    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function registerJsonStrategy($e)
    {
        $matches    = $e->getRouteMatch();
        $controller = $matches->getParam('controller');
        if (false === strpos($controller, __NAMESPACE__)) {
            // not a controller from this module
            return;
        }

        // Potentially, you could be even more selective at this point, and test
        // for specific controller classes, and even specific actions or request
        // methods.

        // Set the JSON strategy when controllers from this module are selected
        $app          = $e->getTarget();
        $locator      = $app->getServiceManager();
        $view         = $locator->get('Zend\View\View');
        $jsonStrategy = $locator->get('ViewJsonStrategy');

        // Attach strategy, which is a listener aggregate, at high priority
        $view->getEventManager()->attach($jsonStrategy, 100);
    }
}
```

While the above examples detail using the `JsonStrategy`, the same could be done for the `FeedStrategy`.

What if you want to use a custom renderer? Or if your app might allow a combination of JSON, Atom feeds, and HTML? At this point, you'll need to create your own custom strategies. Below is an example that appropriately loops through the HTTP Accept header, and selects the appropriate Renderer based on what is matched first.

```
namespace Content\View;

use Zend\EventManager\EventManagerInterface;
use Zend\EventManager\ListenerAggregateInterface;
use Zend\Feed\Writer\Feed;
use Zend\View\Renderer\FeedRenderer;
use Zend\View\Renderer\JsonRenderer;
use Zend\View\Renderer\PhpRenderer;

class AcceptStrategy implements ListenerAggregateInterface
{
    protected $feedRenderer;
    protected $jsonRenderer;
    protected $listeners = array();
    protected $phpRenderer;

    public function __construct(
        PhpRenderer $phpRenderer,
        JsonRenderer $jsonRenderer,
        FeedRenderer $feedRenderer
    ) {
        $this->phpRenderer  = $phpRenderer;
        $this->jsonRenderer = $jsonRenderer;
        $this->feedRenderer = $feedRenderer;
    }

    public function attach(EventManagerInterface $events, $priority = null)
    {
        if (null === $priority) {
            $this->listeners[] = $events->attach('renderer', array($this, 'selectRenderer'));
            $this->listeners[] = $events->attach('response', array($this, 'injectResponse'));
        } else {
            $this->listeners[] = $events->attach('renderer', array($this, 'selectRenderer'), $priorit
            $this->listeners[] = $events->attach('response', array($this, 'injectResponse'), $priorit
        }
    }

    public function detach(EventManagerInterface $events)
    {
        foreach ($this->listeners as $index => $listener) {
            if ($events->detach($listener)) {
                unset($this->listeners[$index]);
            }
        }
    }

    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return \Zend\View\Renderer\RendererInterface
     */
    public function selectRenderer($e)
    {
        $request = $e->getRequest();
        $headers = $request->getHeaders();

```

```
        // No Accept header? return PhpRenderer
        if (!$headers->has('accept')) {
            return $this->phpRenderer;
        }

        $accept = $headers->get('accept');
        foreach ($accept->getPrioritized() as $mediaType) {
            if (0 === strpos($mediaType, 'application/json')) {
                return $this->jsonRenderer;
            }
            if (0 === strpos($mediaType, 'application/rss+xml')) {
                $this->feedRenderer->setFeedType('rss');
                return $this->feedRenderer;
            }
            if (0 === strpos($mediaType, 'application/atom+xml')) {
                $this->feedRenderer->setFeedType('atom');
                return $this->feedRenderer;
            }
        }

        // Nothing matched; return PhpRenderer. Technically, we should probably
        // return an HTTP 415 Unsupported response.
        return $this->phpRenderer;
    }

    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function injectResponse($e)
    {
        $renderer = $e->getRenderer();
        $response = $e->getResponse();
        $result   = $e->getResult();

        if ($renderer === $this->jsonRenderer) {
            // JSON Renderer; set content-type header
            $headers = $response->getHeaders();
            $headers->addHeaderLine('content-type', 'application/json');
        } elseif ($renderer === $this->feedRenderer) {
            // Feed Renderer; set content-type header, and export the feed if
            // necessary
            $feedType  = $this->feedRenderer->getFeedType();
            $headers   = $response->getHeaders();
            $mediatype = 'application/'
                       . (('rss' == $feedType) ? 'rss' : 'atom')
                       . '+xml';
            $headers->addHeaderLine('content-type', $mediatype);

            // If the $result is a feed, export it
            if ($result instanceof Feed) {
                $result = $result->export($feedType);
            }
        } elseif ($renderer !== $this->phpRenderer) {
            // Not a renderer we support, therefor not our strategy. Return
            return;
        }

```

```
        // Inject the content
        $response->setContent($result);
    }
}
```

This strategy would be registered just as we demonstrated registering the `JsonStrategy` earlier. You would also need to define DI configuration to ensure the various renderers are injected when you retrieve the strategy from the application's locator instance.

CHAPTER

FIVE

THE PHPRENDERER

`Zend\View\Renderer\PhpRenderer` "renders" view scripts written in PHP, capturing and returning the output. It composes Variable containers and/or View Models, a helper plugin manager for *helpers*, and optional filtering of the captured output.

The `PhpRenderer` is template system agnostic; you may use *PHP* as your template language, or create instances of other template systems and manipulate them within your view script. Anything you can do with PHP is available to you.

295.1 Usage

Basic usage consists of instantiating or otherwise obtaining an instance of the `PhpRenderer`, providing it with a resolver which will resolve templates to PHP view scripts, and then calling its `render()` method.

Instantiating a renderer is trivial:

```
use Zend\View\Renderer\PhpRenderer;

$renderer = new PhpRenderer();
```

Zend Framework ships with several types of "resolvers", which are used to resolve a template name to a resource a renderer can consume. The ones we will usually use with the `PhpRenderer` are:

- `Zend\View\Resolver\TemplateMapResolver`, which simply maps template names directly to view scripts.
- `Zend\View\Resolver\TemplatePathStack`, which creates a LIFO stack of script directories in which to search for a view script. By default, it appends the suffix ".phtml" to the requested template name, and then loops through the script directories; if it finds a file matching the requested template, it returns the full file path.
- `Zend\View\Resolver\AggregateResolver`, which allows attaching a FIFO queue of resolvers to consult.

We suggest using the `AggregateResolver`, as it allows you to create a multi-tiered strategy for resolving template names.

Programmatically, you would then do something like this:

```
use Zend\View\Renderer\PhpRenderer;
use Zend\View\Resolver;

$renderer = new PhpRenderer();

$resolver = new Resolver\AggregateResolver();
```

```

$renderer->setResolver($resolver);

$map = new Resolver\TemplateMapResolver(array(
    'layout'      => __DIR__ . '/view/layout.phtml',
    'index/index' => __DIR__ . '/view/index/index.phtml',
));
$stack = new Resolver\TemplatePathStack(array(
    'script_paths' => array(
        __DIR__ . '/view',
        $someOtherPath
    )
));

$resolver->attach($map)     // this will be consulted first
         ->attach($stack);
```

You can also specify a specific priority value when registering resolvers, with high, positive integers getting higher priority, and low, negative integers getting low priority, when resolving.

In an MVC application, you can configure this via DI quite easily:

```
return array(
    'di' => array(
        'instance' => array(
            'Zend\View\Resolver\AggregateResolver' => array(
                'injections' => array(
                    'Zend\View\Resolver\TemplateMapResolver',
                    'Zend\View\Resolver\TemplatePathStack',
                ),
            ),

            'Zend\View\Resolver\TemplateMapResolver' => array(
                'parameters' => array(
                    'map'  => array(
                        'layout'      => __DIR__ . '/view/layout.phtml',
                        'index/index' => __DIR__ . '/view/index/index.phtml',
                    ),
                ),
            ),
            'Zend\View\Resolver\TemplatePathStack' => array(
                'parameters' => array(
                    'paths'  => array(
                        'application' => __DIR__ . '/view',
                        'elsewhere'   => $someOtherPath,
                    ),
                ),
            ),
            'Zend\View\Renderer\PhpRenderer' => array(
                'parameters' => array(
                    'resolver' => 'Zend\View\Resolver\AggregateResolver',
                ),
            ),
        ),
    ),
);
```

Now that we have our `PhpRenderer` instance, and it can find templates, let's inject some variables. This can be done in 4 different ways.

- Pass an associative array (or `ArrayAccess` instance, or `Zend\View\Variables` instance) of items as the second argument to `render()`: *$renderer->render($templateName, array('foo' => 'bar))*
- Assign a `Zend\View\Variables` instance, associative array, or `ArrayAccess` instance to the `setVars()` method.
- Assign variables as instance properties of the renderer: *$renderer->foo = 'bar'*. This essentially proxies to an instance of `Variables` composed internally in the renderer by default.
- Create a ViewModel instance, assign variables to that, and pass the ViewModel to the `render()` method:

```
use Zend\View\Model\ViewModel;
use Zend\View\Renderer\PhpRenderer;

$renderer = new PhpRenderer();

$model    = new ViewModel();
$model->setVariable('foo', 'bar');
// or
$model = new ViewModel(array('foo' => 'bar'));

$model->setTemplate($templateName);
$renderer->render($model);
```

Now, let's render something. As a simple example, let us say you have a list of book data.

```
// use a model to get the data for book authors and titles.
$data = array(
    array(
        'author' => 'Hernando de Soto',
        'title' => 'The Mystery of Capitalism'
    ),
    array(
        'author' => 'Henry Hazlitt',
        'title' => 'Economics in One Lesson'
    ),
    array(
        'author' => 'Milton Friedman',
        'title' => 'Free to Choose'
    )
);

// now assign the book data to a renderer instance
$renderer->books = $data;

// and render the template "booklist"
echo $renderer->render('booklist');
```

More often than not, you'll likely be using the MVC layer. As such, you should be thinking in terms of view models. Let's consider the following code from within an action method of a controller.

```
namespace Bookstore\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class BookController extends AbstractActionController
{
    public function listAction()
    {
        // do some work...
```

```

        // Assume $data is the list of books from the previous example
        $model = new ViewModel(array('books' => $data));

        // Optionally specify a template; if we don't, by default it will be
        // auto-determined based on the module name, controller name and this action.
        // In this example, the template would resolve to "bookstore/book/list",
        // and thus the file "bookstore/book/list.phtml"; the following overrides
        // that to set the template to "booklist", and thus the file "booklist.phtml"
        // (note the lack of directory preceding the filename).
        $model->setTemplate('booklist');

        return $model
    }
}
```

This will then be rendered as if the following were executed:

```
$renderer->render($model);
```

Now we need the associated view script. At this point, we'll assume that the template "booklist" resolves to the file `booklist.phtml`. This is a *PHP* script like any other, with one exception: it executes inside the scope of the `PhpRenderer` instance, which means that references to `$this` point to the `PhpRenderer` instance properties and methods. Thus, a very basic view script could look like this:

```
<?php if ($this->books): ?>

    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escapeHtml($val['author']) ?></td>
            <td><?php echo $this->escapeHtml($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>

    </table>

<?php else: ?>

    <p>There are no books to display.</p>

<?php endif;?>
```

Note: Escape Output

The security mantra is "Filter input, escape output." If you are unsure of the source of a given variable – which is likely most of the time – you should escape it based on which HTML context it is being injected into. The primary contexts to be aware of are HTML Body, HTML Attribute, Javascript, CSS and URI. Each context has a dedicated helper available to apply the escaping strategy most appropriate to each context. You should be aware that escaping does vary significantly between contexts - there is no one single escaping strategy that can be globally applied.

In the example above, there are calls to an `escapeHtml()` method. The method is actually a *helper*, a plugin

available via method overloading. Additional escape helpers provide the `escapeHtmlAttr()`, `escapeJs()`, `escapeCss()`, and `escapeUrl()` methods for each of the HTML contexts you are most likely to encounter.

By using the provided helpers and being aware of your variables' contexts, you will prevent your templates from running afoul of Cross-Site Scripting (XSS) vulnerabilities.

We've now toured the basic usage of the `PhpRenderer`. By now you should know how to instantiate the renderer, provide it with a resolver, assign variables and/or create view models, create view scripts, and render view scripts.

295.2 Options and Configuration

`Zend\View\Renderer\PhpRenderer` utilizes several collaborators in order to do its work. use the following methods to configure the renderer.

setHelperPluginManager (*string|Zend\View\HelperPluginManager $helpers*)
Set the helper plugin manager instance used to load, register, and retrieve *helpers*.

Return type Zend\View\Renderer\PhpRenderer

setResolver (*Zend\View\Resolver\ResolverInterface $resolver*)
Set the resolver instance.

Return type Zend\View\Renderer\PhpRenderer

setFilterChain (*Zend\Filter\FilterChain $filters*)
Set a filter chain to use as an output filter on rendered content.

Return type Zend\View\Renderer\PhpRenderer

setVars (*array|ArrayAccess|Zend\View\Variables $variables*)
Set the variables to use when rendering a view script/template.

Return type mixed

setCanRenderTrees (*boolean $canRenderTrees*)
Set flag indicating whether or not we should render trees of view models. If set to true, the `Zend\View\View` instance will not attempt to render children separately, but instead pass the root view model directly to the `PhpRenderer`. It is then up to the developer to render the children from within the view script. This is typically done using the `RenderChildModel` helper: *$this->renderChildModel('child_name')*.

Return type Zend\View\Renderer\PhpRenderer

295.3 Additional Methods

Typically, you'll only ever access variables and *helpers* within your view scripts or when interacting with the `PhpRenderer`. However, there are a few additional methods you may be interested in.

render (*string|Zend\View\Model\ModelInterface $nameOrModel*, *array|Traversable $values = null*)
Render a template/view model.

If `$nameOrModel` is a string, it is assumed to be a template name. That template will be resolved using the current resolver, and then rendered. If `$values` is non-null, those values, and those values only, will be used during rendering, and will replace whatever variable container previously was in the renderer; however, the previous variable container will be reset when done. If `$values` is empty, the current variables container (see *setVars()*) will be injected when rendering.

If `$nameOrModel` is a `Model` instance, the template name will be retrieved from it and used. Additionally, if the model contains any variables, these will be used when rendering; otherwise, the variables container already present, if any, will be used.

It will return the script output.

Return type string

resolver()
Retrieves the `Resolver` instance.

Return type string|Zend\View\Resolver\ResolverInterface

vars(*string $key = null*)
Retrieve a single variable from the container if a key is provided, otherwise it will return the variables container.

Return type mixed

plugin(*string $name*, *array $options = null*)
Get a plugin/helper instance. Proxies to the plugin manager's `get()` method; as such, any `$options` you pass will be passed to the plugin's constructor if this is the first time the plugin has been retrieved. See the section on *helpers* for more information.

Return type Zend\View\Helper\HelperInterface

addTemplate(*string $template*)
Add a template to the stack. When used, the next call to `render()` will loop through all template added using this method, rendering them one by one; the output of the last will be returned.

Return type Zend\View\Renderer\PhpRenderer

CHAPTER

SIX

PHPRENDERER VIEW SCRIPTS

Once you call `render()`, `Zend\View\Renderer\PhpRenderer` then `include()`s the requested view script and executes it "inside" the scope of the `PhpRenderer` instance. Therefore, in your view scripts, references to `$this` actually point to the `PhpRenderer` instance itself.

Variables assigned to the view – either via a *View Model*, *Variables container*, or simply by passing an array of variables to `render()` – may be retrieved in three ways:

- Explicitly, by retrieving them from the `Variables` container composed in the `PhpRenderer`: *$this->vars()->varname*.
- As instance properties of the `PhpRenderer` instance: *$this->varname*. (In this situation, instance property access is simply proxying to the composed `Variables` instance.)
- As local PHP variables: *$varname*. The `PhpRenderer` extracts the members of the `Variables` container locally.

We generally recommend using the second notation, as it's less verbose than the first, but differentiates between variables in the view script scope and those assigned to the renderer from elsewhere.

By way of reminder, here is the example view script from the `PhpRenderer` introduction.

```
<?php if ($this->books): ?>

    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escapeHtml($val['author']) ?></td>
            <td><?php echo $this->escapeHtml($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>

    </table>

<?php else: ?>

    <p>There are no books to display.</p>

<?php endif;?>
```

296.1 Escaping Output

One of the most important tasks to perform in a view script is to make sure that output is escaped properly; among other things, this helps to avoid cross-site scripting attacks. Unless you are using a function, method, or helper that does escaping on its own, you should always escape variables when you output them and pay careful attention to applying the correct escaping strategy to each HTML context you use.

The `PhpRenderer` includes a selection of helpers you can use for this purpose: `EscapeHtml`, `EscapeHtmlAttr`, `EscapeJs`, `EscapeCss`, and `EscapeUrl`. Matching the correct helper (or combination of helpers) to the context into which you are injecting untrusted variables will ensure that you are protected against Cross-Site Scripting (XSS) vulnerabilities.

```
// bad view-script practice:
echo $this->variable;

// good view-script practice:
echo $this->escapeHtml($this->variable);

// and remember context is always relevant!
<script type="text/javascript">
    var foo = "<?php echo $this->escapeJs($variable) ?>";
</script>
```

CHAPTER

SEVEN

THE VIEWEVENT

The view layer of Zend Framework 2 incorporates and utilizes a custom `Zend\EventManager\Event` implementation - `Zend\View\ViewEvent`. This event is created during `Zend\View\View::getEvent()` and is passed directly to all the events that method triggers.

The `ViewEvent` adds accessors and mutators for the following:

- `Model` object, typically representing the layout view model.
- `Renderer` object.
- `Request` object.
- `Response` object.
- `Result` object.

The methods it defines are:

- `setModel(Model $model)`
- `getModel()`
- `setRequest($request)`
- `getRequest()`
- `setResponse($response)`
- `getResponse()`
- `setRenderer($renderer)`
- `getRenderer()`
- `setResult($result)`
- `getResult()`

297.1 Order of events

The following events are triggered, in the following order:

Table 297.1: ViewEvent Events

Name	Constant	Description
`renderer`	`ViewEvent::EVENT_RENDERER`	Render the view, with the help of renderers.
`renderer.post`	`ViewEvent::EVENT_RENDERER_POST`	Triggers after the view is rendered.
`response`	`ViewEvent::EVENT_RESPONSE`	Populate the response from the view.

Those events are extensively describe in the following sections.

297.2 ViewEvent::EVENT_RENDERER

297.2.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

For PhpStrategy

This listener is added when the strategy used for rendering is `PhpStrategy`:

Table 297.2: ViewEvent::EVENT_RENDERER Listeners for PhpStrategy

Class	Priority	Method Called	Description
`Zend\View\Strategy\PhpStrategy`	1	`selectRenderer`	Return a `PhpRenderer`

For JsonStrategy

This listener is added when the strategy used for rendering is `JsonStrategy`:

Table 297.3: ViewEvent::EVENT_RENDERER Listeners for JsonStrategy

Class	Priority	Method Called	Description
`Zend\View\Strategy\JsonStrategy`	1	`selectRenderer`	Return a `JsonRenderer`

For FeedStrategy

This listener is added when the strategy used for rendering is `FeedStrategy`:

Table 297.4: ViewEvent::EVENT_RENDERER Listeners for FeedStrategy

Class	Priority	Method Called	Description
`Zend\View\Strategy\FeedStrategy`	1	`selectRenderer`	Return a `FeedRenderer`

297.2.2 Triggerers

This event is triggered by the following classes:

Table 297.5: ViewEvent::EVENT_RENDERER Triggerers

Class	In Method	Description
`Zend\View\View`	`render`	It has a short circuit callback that stops propagation once one result return an instance of a Renderer.

297.3 ViewEvent::EVENT_RENDERER_POST

297.3.1 Listeners

There are currently no built-in listeners for this event.

297.3.2 Triggerers

This event is triggered by the following classes:

Table 297.6: ViewEvent::EVENT_RENDERER_POST Triggerers

Class	In Method	Description
`Zend\View\View`	`render`	This event is triggered after `ViewEvent::EVENT_RENDERER` and before `ViewEvent::EVENT_RESPONSE`.

297.4 ViewEvent::EVENT_RESPONSE

297.4.1 Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

For PhpStrategy

This listener is added when the strategy used for rendering is `PhpStrategy`:

Table 297.7: ViewEvent::EVENT_RESPONSE Listeners for PhpStrategy

Class	Priority	Method Called	Description
`Zend\View\Strategy\PhpStrategy`	1	`injectResponse`	Populate the `Response` object from the view.

For JsonStrategy

This listener is added when the strategy used for rendering is `JsonStrategy`:

Table 297.8: ViewEvent::EVENT_RESPONSE Listeners for JsonStrategy

Class	Priority	Method Called	Description
`Zend\View\Strategy\JsonStrategy`	1	`injectResponse`	Populate the `Response` object from the view.

For FeedStrategy

This listener is added when the strategy used for rendering is `FeedStrategy`:

Table 297.9: ViewEvent::EVENT_RESPONSE Listeners for FeedStrategy

Class	Priority	Method Called	Description
`Zend\View\Strategy\FeedStrategy`	1	`injectResponse`	Populate the `Response` object from the view.

297.4.2 Triggerers

This event is triggered by the following classes:

Table 297.10: ViewEvent::EVENT_RESPONSE Triggerers

Class	In Method	Description
`Zend\View\View`	`render`	This event is triggered after `ViewEvent::EVENT_RENDERER` and `ViewEvent::EVENT_RENDERER_POST`.

CHAPTER

EIGHT

VIEW HELPERS

In your view scripts, often it is necessary to perform certain complex functions over and over: e.g., formatting a date, generating form elements, or displaying action links. You can use helper, or plugin, classes to perform these behaviors for you.

A helper is simply a class that implements `Zend\View\Helper\HelperInterface` and it simply defines two methods, `setView()`, which accepts a `Zend\View\Renderer\RendererInterface` instance/implementation, and `getView()`, used to retrieve that instance. `Zend\View\Renderer\PhpRenderer` composes a *plugin manager*, allowing you to retrieve helpers, and also provides some method overloading capabilities that allow proxying method calls to helpers.

As an example, let's say we have a helper class named `MyModule\View\Helper\LowerCase`, which we register in our plugin manager with the name "lowercase". We can retrieve it in one of the following ways:

```
// $view is a PhpRenderer instance

// Via the plugin manager:
$pluginManager = $view->getHelperPluginManager();
$helper        = $pluginManager->get('lowercase');

// Retrieve the helper instance, via the method "plugin",
// which proxies to the plugin manager:
$helper = $view->plugin('lowercase');

// If the helper does not define __invoke(), the following also retrieves it:
$helper = $view->lowercase();

// If the helper DOES define __invoke, you can call the helper
// as if it is a method:
$filtered = $view->lowercase('some value');
```

The last two examples demonstrate how the `PhpRenderer` uses method overloading to retrieve and/or invoke helpers directly, offering a convenience API for end users.

A large number of helpers are provided in the standard distribution of Zend Framework. You can also register helpers by adding them to the *plugin manager*.

CHAPTER

NINE

INCLUDED HELPERS

Zend Framework comes with an initial set of helper classes. In particular, there are helpers for creating route-based *URL*s and *HTML* lists, as well as declaring variables. Additionally, there are a rich set of helpers for providing values for, and rendering, the various HTML <*head*> tags, such as `HeadTitle`, `HeadLink`, and `HeadScript`. The currently shipped helpers include:

- *BasePath*
- *Cycle*
- *Doctype*
- *HeadLink*
- *HeadMeta*
- *HeadScript*
- *HeadStyle*
- *HeadTitle*
- *HtmlList*
- *HTML Object Plugins*
- *Identity*
- *InlineScript*
- *JSON*
- *Partial*
- *Placeholder*

Note: View helpers related to **Internationalization** are documented in the *I18n View Helpers* chapter.

Note: View helpers related to **navigation** are documented in the *Navigation View Helpers* chapter.

299.1 BasePath Helper

While most *URL*s generated by the framework have the base *URL* prepended automatically, developers will need to prepend the base *URL* to their own *URL*s (usually inside an href attribute) in order for paths to resources to be correct.

If you're running on ZF2's MVC base, `basePath()` will point to the `public` folder of the application's root.

Usage of the `basePath()` helper is straightforward:

```
/*
 * The following assume that the base URL of the page/application is "/mypage".
 */

/*
 * Prints:
 * <base href="/mypage/" />
 */
<base href="<?php echo $this->basePath(); ?>" />

/*
 * Prints:
 * <link rel="stylesheet" type="text/css" href="/mypage/css/base.css" />
 */
<link rel="stylesheet" type="text/css"
     href="<?php echo $this->basePath('css/base.css'); ?>" />
```

Note: For simplicity's sake, we strip out the entry *PHP* file (e.g., "`index.php`") from the base *URL* . However, in some situations this may cause a problem. If one occurs, use `$this->plugin('basePath')->setBasePath()` to manually set the base path.

299.2 Cycle Helper

The `Cycle` helper is used to alternate a set of values.

Cycle Helper Basic Usage

To add elements to cycle just specify them in constructor or use `assign(array $data)` function

```
<?php foreach ($this->books as $book):?>
  <tr style="background-color:<?php echo $this->cycle(array("#F0F0F0",
                                                            "#FFFFFF"))
                                                 ->next()?>">
  <td><?php echo $this->escapeHtml($book['author']) ?></td>
</tr>
<?php endforeach;?>

// Moving in backwards order and assign function
$this->cycle()->assign(array("#F0F0F0","#FFFFFF"));
$this->cycle()->prev();
?>
```

The output

```
<tr style="background-color:'#F0F0F0'">
   <td>First</td>
</tr>
<tr style="background-color:'#FFFFFF'">
   <td>Second</td>
</tr>
```

Working with two or more cycles

To use two cycles you have to specify the names of cycles. Just set second parameter in cycle method. `$this->cycle(array("#F0F0F0","#FFFFFF"),'cycle2')`. You can also use setName($name) function.

```
<?php foreach ($this->books as $book):?>
  <tr style="background-color:<?php echo $this->cycle(array("#F0F0F0",
                                                            "#FFFFFF"))
                                                  ->next()?>">
  <td><?php echo $this->cycle(array(1,2,3),'number')->next()?></td>
  <td><?php echo $this->escapeHtml($book['author'])?></td>
</tr>
<?php endforeach;?>
```

299.3 Doctype Helper

Valid *HTML* and *XHTML* documents should include a `DOCTYPE` declaration. Besides being difficult to remember, these can also affect how certain elements in your document should be rendered (for instance, CDATA escaping in **<script>** and **<style>** elements.

The `Doctype` helper allows you to specify one of the following types:

- `XHTML11`
- `XHTML1_STRICT`
- `XHTML1_TRANSITIONAL`
- `XHTML1_FRAMESET`
- `XHTML1_RDFA`
- `XHTML_BASIC1`
- `HTML4_STRICT`
- `HTML4_LOOSE`
- `HTML4_FRAMESET`
- `HTML5`

You can also specify a custom doctype as long as it is well-formed.

The `Doctype` helper is a concrete implementation of the *Placeholder helper*.

Doctype Helper Basic Usage

You may specify the doctype at any time. However, helpers that depend on the doctype for their output will recognize it only after you have set it, so the easiest approach is to specify it in your bootstrap:

```
$doctypeHelper = new Zend\View\Helper\Doctype();
$doctypeHelper->doctype('XHTML1_STRICT');
```

And then print it out on top of your layout script:

```
<?php echo $this->doctype() ?>
```

Retrieving the Doctype

If you need to know the doctype, you can do so by calling `getDoctype()` on the object, which is returned by invoking the helper.

```
$doctype = $view->doctype()->getDoctype();
```

Typically, you'll simply want to know if the doctype is *XHTML* or not; for this, the `isXhtml()` method will suffice:

```
if ($view->doctype()->isXhtml()) {
    // do something differently
}
```

You can also check if the doctype represents an *HTML5* document.

```
if ($view->doctype()->isHtml5()) {
    // do something differently
}
```

Choosing a Doctype to Use with the Open Graph Protocol

To implement the Open Graph Protocol, you may specify the XHTML1_RDFA doctype. This doctype allows a developer to use the Resource Description Framework within an *XHTML* document.

```
$doctypeHelper = new Zend\View\Helper\Doctype();
$doctypeHelper->doctype('XHTML1_RDFA');
```

The RDFa doctype allows XHTML to validate when the 'property' meta tag attribute is used per the Open Graph Protocol spec. Example within a view script:

```
<?php echo $this->doctype('XHTML1_RDFA'); ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:og="http://opengraphprotocol.org/schema/">
<head>
   <meta property="og:type" content="musician" />
```

In the previous example, we set the property to og:type. The og references the Open Graph namespace we specified in the html tag. The content identifies the page as being about a musician. See the Open Graph Protocol documentation for supported properties. The *HeadMeta helper* may be used to programmatically set these Open Graph Protocol meta tags.

Here is how you check if the doctype is set to XHTML1_RDFA:

```
<?php echo $this->doctype() ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      <?php if ($view->doctype()->isRdfa()): ?>
      xmlns:og="http://opengraphprotocol.org/schema/"
      xmlns:fb="http://www.facebook.com/2008/fbml"
      <?php endif; ?>
>
```

299.4 HeadLink Helper

The *HTML* **<link>** element is increasingly used for linking a variety of resources for your site: stylesheets, feeds, favicons, trackbacks, and more. The `HeadLink` helper provides a simple interface for creating and aggregating these elements for later retrieval and output in your layout script.

The `HeadLink` helper has special methods for adding stylesheet links to its stack:

- `appendStylesheet($href, $media, $conditionalStylesheet, $extras)`
- `offsetSetStylesheet($index, $href, $media, $conditionalStylesheet, $extras)`
- `prependStylesheet($href, $media, $conditionalStylesheet, $extras)`
- `setStylesheet($href, $media, $conditionalStylesheet, $extras)`

The `$media` value defaults to 'screen', but may be any valid media value. `$conditionalStylesheet` is a string or boolean `FALSE`, and will be used at rendering time to determine if special comments should be included to prevent loading of the stylesheet on certain platforms. `$extras` is an array of any extra values that you want to be added to the tag.

Additionally, the `HeadLink` helper has special methods for adding 'alternate' links to its stack:

- `appendAlternate($href, $type, $title, $extras)`
- `offsetSetAlternate($index, $href, $type, $title, $extras)`
- `prependAlternate($href, $type, $title, $extras)`
- `setAlternate($href, $type, $title, $extras)`

The `headLink()` helper method allows specifying all attributes necessary for a **<link>** element, and allows you to also specify placement – whether the new element replaces all others, prepends (top of stack), or appends (end of stack).

The `HeadLink` helper is a concrete implementation of the *Placeholder helper*.

HeadLink Helper Basic Usage

You may specify a **headLink** at any time. Typically, you will specify global links in your layout script, and application specific links in your application view scripts. In your layout script, in the <head> section, you will then echo the helper to output it.

```
<?php // setting links in a view script:
$this->headLink(array(
    'rel'  => 'favicon',
    'href' => '/img/favicon.ico',
), 'PREPEND')
    ->appendStylesheet('/styles/basic.css')
    ->prependStylesheet(
        '/styles/moz.css',
        'screen',
        true,
        array('id' => 'my_stylesheet')
    );
?>
<?php // rendering the links: ?>
<?php echo $this->headLink() ?>
```

299.5 HeadMeta Helper

The *HTML* **<meta>** element is used to provide meta information about your *HTML* document – typically keywords, document character set, caching pragmas, etc. Meta tags may be either of the 'http-equiv' or 'name' types, must contain a 'content' attribute, and can also have either of the 'lang' or 'scheme' modifier attributes.

The `HeadMeta` helper supports the following methods for setting and adding meta tags:

- `appendName($keyValue, $content, $conditionalName)`
- `offsetSetName($index, $keyValue, $content, $conditionalName)`
- `prependName($keyValue, $content, $conditionalName)`
- `setName($keyValue, $content, $modifiers)`
- `appendHttpEquiv($keyValue, $content, $conditionalHttpEquiv)`
- `offsetSetHttpEquiv($index, $keyValue, $content, $conditionalHttpEquiv)`
- `prependHttpEquiv($keyValue, $content, $conditionalHttpEquiv)`
- `setHttpEquiv($keyValue, $content, $modifiers)`
- `setCharset($charset)`

The following methods are also supported with XHTML1_RDFA doctype set with the *Doctype helper*:

- `appendProperty($property, $content, $modifiers)`
- `offsetSetProperty($index, $property, $content, $modifiers)`
- `prependProperty($property, $content, $modifiers)`
- `setProperty($property, $content, $modifiers)`

The `$keyValue` item is used to define a value for the 'name' or 'http-equiv' key; `$content` is the value for the 'content' key, and `$modifiers` is an optional associative array that can contain keys for 'lang' and/or 'scheme'.

You may also set meta tags using the `headMeta()` helper method, which has the following signature: `headMeta($content, $keyValue, $keyType = 'name', $modifiers = array(), $placement = 'APPEND')`. `$keyValue` is the content for the key specified in `$keyType`, which should be either 'name' or 'http-equiv'. `$keyType` may also be specified as 'property' if the doctype has been set to XHTML1_RDFA. `$placement` can be 'SET' (overwrites all previously stored values), 'APPEND' (added to end of stack), or 'PREPEND' (added to top of stack).

`HeadMeta` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a `stdClass` token, which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadMeta` helper is a concrete implementation of the *Placeholder helper*.

HeadMeta Helper Basic Usage

You may specify a new meta tag at any time. Typically, you will specify client-side caching rules or SEO keywords.

For instance, if you wish to specify SEO keywords, you'd be creating a meta name tag with the name 'keywords' and the content the keywords you wish to associate with your page:

```
// setting meta keywords
$this->headMeta()->appendName('keywords', 'framework, PHP, productivity');
```

If you wished to set some client-side caching rules, you'd set http-equiv tags with the rules you wish to enforce:

```
// disabling client-side cache
$this->headMeta()->appendHttpEquiv('expires',
                                   'Wed, 26 Feb 1997 08:21:57 GMT')
                 ->appendHttpEquiv('pragma', 'no-cache')
                 ->appendHttpEquiv('Cache-Control', 'no-cache');
```

Another popular use for meta tags is setting the content type, character set, and language:

```
// setting content type and character set
$this->headMeta()->appendHttpEquiv('Content-Type',
                                   'text/html; charset=UTF-8')
                 ->appendHttpEquiv('Content-Language', 'en-US');
```

If you are serving an *HTML5* document, you should provide the character set like this:

```
// setting character set in HTML5
$this->headMeta()->setCharset('UTF-8'); // Will look like <meta charset="UTF-8">
```

As a final example, an easy way to display a transitional message before a redirect is using a "meta refresh":

```
// setting a meta refresh for 3 seconds to a new url:
$this->headMeta()->appendHttpEquiv('Refresh',
                                   '3;URL=http://www.some.org/some.html');
```

When you're ready to place your meta tags in the layout, simply echo the helper:

```
<?php echo $this->headMeta() ?>
```

HeadMeta Usage with XHTML1_RDFA doctype

Enabling the RDFa doctype with the *Doctype helper* enables the use of the 'property' attribute (in addition to the standard 'name' and 'http-equiv') with HeadMeta. This is commonly used with the Facebook Open Graph Protocol.

For instance, you may specify an open graph page title and type as follows:

```
$this->doctype(Zend\View\Helper\Doctype::XHTML_RDFA);
$this->headMeta()->setProperty('og:title', 'my article title');
$this->headMeta()->setProperty('og:type', 'article');
echo $this->headMeta();

// output is:
//   <meta property="og:title" content="my article title" />
//   <meta property="og:type" content="article" />
```

299.6 HeadScript Helper

The *HTML* **<script>** element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `HeadScript` helper allows you to manage both.

The `HeadScript` helper supports the following methods for setting and adding scripts:

- `appendFile($src, $type = 'text/javascript', $attrs = array())`
- `offsetSetFile($index, $src, $type = 'text/javascript', $attrs = array())`
- `prependFile($src, $type = 'text/javascript', $attrs = array())`
- `setFile($src, $type = 'text/javascript', $attrs = array())`
- `appendScript($script, $type = 'text/javascript', $attrs = array())`
- `offsetSetScript($index, $script, $type = 'text/javascript', $attrs = array())`
- `prependScript($script, $type = 'text/javascript', $attrs = array())`

- `setScript($script, $type = 'text/javascript', $attrs = array())`

In the case of the * `File()` methods, `$src` is the remote location of the script to load; this is usually in the form of a *URL* or a path. For the * `Script()` methods, `$script` is the client-side scripting directives you wish to use in the element.

Note: Setting Conditional Comments

`HeadScript` allows you to wrap the script tag in conditional comments, which allows you to hide it from specific browsers. To add the conditional tags, pass the conditional value as part of the `$attrs` parameter in the method calls.

Headscript With Conditional Comments

```
// adding scripts
$this->headScript()->appendFile(
    '/js/prototype.js',
    'text/javascript',
    array('conditional' => 'lt IE 7')
);
```

Note: Preventing HTML style comments or CDATA wrapping of scripts

By default `HeadScript` will wrap scripts with HTML comments or it wraps scripts with XHTML cdata. This behavior can be problematic when you intend to use the script tag in an alternative way by setting the type to something other then 'text/javascript'. To prevent such escaping, pass an `noescape` with a value of true as part of the `$attrs` parameter in the method calls.

Create an jQuery template with the headScript

```
// jquery template
$template = '<div class="book">{{:title}}</div>';
$this->headScript()->appendScript(
    $template,
    'text/x-jquery-tmpl',
    array('id='tmpl-book', 'noescape' => true)
);
```

`HeadScript` also allows capturing scripts; this can be useful if you want to create the client-side script programmatically, and then place it elsewhere. The usage for this will be showed in an example below.

Finally, you can also use the `headScript()` method to quickly add script elements; the signature for this is `headScript($mode = 'FILE', $spec = null, $placement = 'APPEND', array $attrs = array(), $type = 'text/javascript')`. The `$mode` is either 'FILE' or 'SCRIPT', depending on if you're linking a script or defining one. `$spec` is either the script file to link or the script source itself. `$placement` should be either 'APPEND', 'PREPEND', or 'SET'. `$attrs` is an array of script attributes. `$type` is the script type attribute.

`HeadScript` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a `stdClass` token, which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadScript` helper is a concrete implementation of the *Placeholder helper*.

Note: Use InlineScript for HTML Body Scripts

`HeadScript`'s sibling helper, *InlineScript*, should be used when you wish to include scripts inline in the *HTML* **body**. Placing scripts at the end of your document is a good practice for speeding up delivery of your page, particularly when using 3rd party analytics scripts.

Note: Arbitrary Attributes are Disabled by Default

By default, `HeadScript` only will render **<script>** attributes that are blessed by the W3C. These include 'type', 'charset', 'defer', 'language', and 'src'. However, some JavaScript frameworks, notably Dojo, utilize custom attributes in order to modify behavior. To allow such attributes, you can enable them via the `setAllowArbitraryAttributes()` method:

```
$this->headScript()->setAllowArbitraryAttributes(true);
```

HeadScript Helper Basic Usage

You may specify a new script tag at any time. As noted above, these may be links to outside resource files or scripts themselves.

```
// adding scripts
$this->headScript()->appendFile('/js/prototype.js')
                    ->appendScript($onloadScript);
```

Order is often important with client-side scripting; you may need to ensure that libraries are loaded in a specific order due to dependencies each have; use the various append, prepend, and offsetSet directives to aid in this task:

```
// Putting scripts in order

// place at a particular offset to ensure loaded last
$this->headScript()->offsetSetFile(100, '/js/myfuncs.js');

// use scriptaculous effects (append uses next index, 101)
$this->headScript()->appendFile('/js/scriptaculous.js');

// but always have base prototype script load first:
$this->headScript()->prependFile('/js/prototype.js');
```

When you're finally ready to output all scripts in your layout script, simply echo the helper:

```
<?php echo $this->headScript() ?>
```

Capturing Scripts Using the HeadScript Helper

Sometimes you need to generate client-side scripts programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the script and sprinkling in *PHP* tags. `HeadScript` lets you do just that, capturing it to the stack:

```
<?php $this->headScript()->captureStart() ?>
var action = '<?php echo $this->baseUrl ?>';
$('foo_form').action = action;
<?php $this->headScript()->captureEnd() ?>
```

The following assumptions are made:

- The script will be appended to the stack. If you wish for it to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- The script *MIME* type is assumed to be 'text/javascript'; if you wish to specify a different type, you will need to pass it as the second argument to `captureStart()`.
- If you wish to specify any additional attributes for the **<script>** tag, pass them in an array as the third argument to `captureStart()`.

299.7 HeadStyle Helper

The *HTML* **<style>** element is used to include *CSS* stylesheets inline in the *HTML* **<head>** element.

Note: Use HeadLink to link CSS files

HeadLink should be used to create **<link>** elements for including external stylesheets. `HeadStyle` is used when you wish to define your stylesheets inline.

The `HeadStyle` helper supports the following methods for setting and adding stylesheet declarations:

- `appendStyle($content, $attributes = array())`
- `offsetSetStyle($index, $content, $attributes = array())`
- `prependStyle($content, $attributes = array())`
- `setStyle($content, $attributes = array())`

In all cases, `$content` is the actual *CSS* declarations. `$attributes` are any additional attributes you wish to provide to the `style` tag: lang, title, media, or dir are all permissible.

Note: Setting Conditional Comments

`HeadStyle` allows you to wrap the style tag in conditional comments, which allows you to hide it from specific browsers. To add the conditional tags, pass the conditional value as part of the `$attributes` parameter in the method calls.

Headstyle With Conditional Comments

```
// adding scripts
$this->headStyle()->appendStyle($styles, array('conditional' => 'lt IE 7'));
```

`HeadStyle` also allows capturing style declarations; this can be useful if you want to create the declarations programmatically, and then place them elsewhere. The usage for this will be showed in an example below.

Finally, you can also use the `headStyle()` method to quickly add declarations elements; the signature for this is `headStyle($content = null, $placement = 'APPEND', $attributes = array())`. `$placement` should be either 'APPEND', 'PREPEND', or 'SET'.

`HeadStyle` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a `stdClass` token, which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadStyle` helper is a concrete implementation of the *Placeholder helper*.

Note: UTF-8 encoding used by default

By default, Zend Framework uses *UTF-8* as its default encoding, and, specific to this case, `Zend\View` does as well. So if you want to use another encoding with `headStyle`, you will have do three things:

1. Create a custom renderer and implement a `getEncoding()` method;
2. Create a custom rendering strategy that will return an instance of your custom renderer;
3. Attach the custom strategy in the `ViewEvent`;

See the example below.

First we have to write the custom renderer:

```
// module/MyModule/View/Renderer/MyRenderer.php
<?php

namespace MyModule\View\Renderer;

// Since we just want to implement the getEncoding() method, we can extend the Zend native renderer
use Zend\View\Renderer\PhpRenderer;

class MyRenderer extends PhpRenderer
{
    /**
     * @var string
     */
    protected $encoding;

    /**
     * Constructor
     *
     * @param  string $encoding The encoding to be used
     */
    public function __construct($encoding)
    {
        parent::__construct();
        $this->encoding = $encoding;
    }

    /**
     * Sets the encoding
     *
     * @param string $encoding The encoding to be used
     */
    public function setEncoding($encoding)
    {
        $this->encoding = $encoding;
    }

    /**
     * Gets the encoding
     *
     * @return string The encoding being used
     */
    public function getEncoding()
```

```
    {
        return $this->encoding;
    }
}
```

Now we make some configuration in the module class:

```
// module/MyModule.php
<?php

namespace MyModule;

use MyModule\View\Renderer\MyRenderer;
use Zend\Mvc\MvcEvent;
use Zend\View\Strategy\PhpRendererStrategy;

class Module
{
    public function getConfig(){/* ... */}

    public function getAutoloaderConfig(){/* ... */}

    public function getServiceConfig()
    {
        return array(
            'factories' => array(
                // Register our custom renderer in the service manager
                'MyCustomRenderer' => function ($serviceManager) {
                    $myRenderer = new MyRenderer('ISO-8859-1');
                    return $myRenderer;
                },
                'MyCustomStrategy' => function ($serviceManager) {
                    // As stated before, we just want to implement the getEncoding() method, so we can use
                    // Zend\View\Strategy\PhpRendererStrategy and just provide our custom renderer to it.
                    $myRenderer = $serviceManager->get('MyCustomRenderer');
                    $strategy = new PhpRendererStrategy($myRenderer);
                    return $strategy;
                }
            ),
        );
    }

    public function onBootstrap(MvcEvent $e)
    {
        // Register a render event
        $app = $e->getParam('application');
        $app->getEventManager()->attach('render', array($this, 'registerMyStrategy'), 100);
    }

     public function registerMyStrategy(MvcEvent $e)
     {
         $app          = $e->getTarget();
         $locator      = $app->getServiceManager();
         $view         = $locator->get('Zend\View\View');
         $myStrategy = $locator->get('MyCustomStrategy');

         // Attach strategy, which is a listener aggregate, at high priority
         $view->getEventManager()->attach($myStrategy, 100);
     }
```

```
}
```

See the quick start *Creating and Registering Alternate Rendering and Response Strategies* chapter for more information on how to create and register custom strategies to your view.

HeadStyle Helper Basic Usage

You may specify a new style tag at any time:

```
// adding styles
$this->headStyle()->appendStyle($styles);
```

Order is very important with *CSS*; you may need to ensure that declarations are loaded in a specific order due to the order of the cascade; use the various append, prepend, and offsetSet directives to aid in this task:

```
// Putting styles in order

// place at a particular offset:
$this->headStyle()->offsetSetStyle(100, $customStyles);

// place at end:
$this->headStyle()->appendStyle($finalStyles);

// place at beginning
$this->headStyle()->prependStyle($firstStyles);
```

When you're finally ready to output all style declarations in your layout script, simply echo the helper:

```
<?php echo $this->headStyle() ?>
```

Capturing Style Declarations Using the HeadStyle Helper

Sometimes you need to generate *CSS* style declarations programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the styles and sprinkling in *PHP* tags. `HeadStyle` lets you do just that, capturing it to the stack:

```
<?php $this->headStyle()->captureStart() ?>
body {
    background-color: <?php echo $this->bgColor ?>;
}
<?php $this->headStyle()->captureEnd() ?>
```

The following assumptions are made:

- The style declarations will be appended to the stack. If you wish for them to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- If you wish to specify any additional attributes for the **<style>** tag, pass them in an array as the second argument to `captureStart()`.

299.8 HeadTitle Helper

The *HTML* **<title>** element is used to provide a title for an *HTML* document. The `HeadTitle` helper allows you to programmatically create and store the title for later retrieval and output.

The `HeadTitle` helper is a concrete implementation of the *Placeholder helper*. It overrides the `toString()` method to enforce generating a **<title>** element, and adds a `headTitle()` method for quick and easy setting and aggregation of title elements. The signature for that method is `headTitle($title, $setType = null)`; by default, the value is appended to the stack (aggregating title segments) if left at null, but you may also specify either 'PREPEND' (place at top of stack) or 'SET' (overwrite stack).

Since setting the aggregating (attach) order on each call to `headTitle` can be cumbersome, you can set a default attach order by calling `setDefaultAttachOrder()` which is applied to all `headTitle()` calls unless you explicitly pass a different attach order as the second parameter.

HeadTitle Helper Basic Usage

You may specify a title tag at any time. A typical usage would have you setting title segments for each level of depth in your application: site, module, controller, action, and potentially resource. This could be achieved in the module class.

```
// module/MyModule/Module.php
<?php

namespace MyModule;

class Module
{
    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function onBootstrap($e)
    {
        // Register a render event
        $app = $e->getParam('application');
        $app->getEventManager()->attach('render', array($this, 'setLayoutTitle'));
    }

    /**
     * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
     * @return void
     */
    public function setLayoutTitle($e)
    {
        $matches    = $e->getRouteMatch();
        $action     = $matches->getParam('action');
        $controller = $matches->getParam('controller');
        $module     = __NAMESPACE__;
        $siteName   = 'Zend Framework';

        // Getting the view helper manager from the application service manager
        $viewHelperManager = $e->getApplication()->getServiceManager()->get('viewHelperManager');

        // Getting the headTitle helper from the view helper manager
        $headTitleHelper   = $viewHelperManager->get('headTitle');

        // Setting a separator string for segments
        $headTitleHelper->setSeparator(' - ');

        // Setting the action, controller, module and site name as title segments
        $headTitleHelper->append($action);
```

```
        $headTitleHelper->append($controller);
        $headTitleHelper->append($module);
        $headTitleHelper->append($siteName);
    }
}
```

When you're finally ready to render the title in your layout script, simply echo the helper:

```
<!-- renders <action> - <controller> - <module> - Zend Framework -->
<?php echo $this->headTitle() ?>
```

299.9 HtmlList Helper

- `htmlList($items, $ordered, $attribs, $escape)`: generates unordered and ordered lists based on the `$items` passed to it. If `$items` is a multidimensional array, a nested list will be built. If the `$escape` flag is `TRUE` (default), individual items will be escaped using the view objects registered escaping mechanisms; pass a `FALSE` value if you want to allow markup in your lists.

299.9.1 Unordered list

```
$items = array(
    'Level one, number one',
    array(
        'Level two, number one',
        'Level two, number two',
        array(
            'Level three, number one'
        ),
        'Level two, number three',
    ),
    'Level one, number two',
 );

echo $this->htmlList($items);
```

Output:

```
<ul>
    <li>Level one, number one
        <ul>
            <li>Level two, number one</li>
            <li>Level two, number two
                <ul>
                    <li>Level three, number one</li>
                </ul>
            </li>
            <li>Level two, number three</li>
        </ul>
    </li>
    <li>Level one, number two</li>
</ul>
```

299.9.2 Ordered list

```
echo $this->htmlList($items, true);
```

Output:

```
<ol>
    <li>Level one, number one
        <ol>
            <li>Level two, number one</li>
            <li>Level two, number two
                <ol>
                    <li>Level three, number one</li>
                </ol>
            </li>
            <li>Level two, number three</li>
        </ol>
    </li>
    <li>Level one, number two</li>
</ol>
```

299.9.3 HTML attributes

```
$attribs = array(
    'class' => 'foo',
);

echo $this->htmlList($items, false, $attribs);
```

Output:

```
<ul class="foo">
    <li>Level one, number one
        <ul class="foo">
            <li>Level two, number one</li>
            <li>Level two, number two
                <ul class="foo">
                    <li>Level three, number one</li>
                </ul>
            </li>
            <li>Level two, number three</li>
        </ul>
    </li>
    <li>Level one, number two</li>
</ul>
```

299.9.4 Escape Output

```
$items = array(
    'Level one, number <strong>one</strong>',
    'Level one, number <em>two</em>',
 );

// Escape output (default)
echo $this->htmlList($items);
```

```

// Don't escape output
echo $this->htmlList($items, false, false, false);
```

Output:

```
<!-- Escape output (default) -->
<ul class="foo">
    <li>Level one, number &lt;strong&gt;one&lt;/strong&gt;</li>
    <li>Level one, number &lt;em&gt;two&lt;/em&gt;</li>
</ul>

<!-- Don't escape output -->
<ul class="foo">
    <li>Level one, number <strong>one</strong></li>
    <li>Level one, number <em>two</em></li>
</ul>
```

299.10 HTML Object Helpers

The *HTML* **<object>** element is used for embedding media like Flash or QuickTime in web pages. The object view helpers take care of embedding media with minimum effort.

There are four initial Object helpers:

- `htmlFlash()` Generates markup for embedding Flash files.
- `htmlObject()` Generates markup for embedding a custom Object.
- `htmlPage()` Generates markup for embedding other (X)HTML pages.
- `htmlQuicktime()` Generates markup for embedding QuickTime files.

All of these helpers share a similar interface. For this reason, this documentation will only contain examples of two of these helpers.

Flash helper

Embedding Flash in your page using the helper is pretty straight-forward. The only required argument is the resource *URI*.

```
<?php echo $this->htmlFlash('/path/to/flash.swf'); ?>
```

This outputs the following *HTML*:

```
<object data="/path/to/flash.swf"
        type="application/x-shockwave-flash"
        classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab">
</object>
```

Additionally you can specify attributes, parameters and content that can be rendered along with the **<object>**. This will be demonstrated using the `htmlObject()` helper.

Customizing the object by passing additional arguments

The first argument in the object helpers is always required. It is the *URI* to the resource you want to embed. The second argument is only required in the `htmlObject()` helper. The other helpers already contain the correct value for this argument. The third argument is used for passing along attributes to the object element. It only accepts an array with key-value pairs. `classid` and `codebase` are examples of such attributes. The fourth argument also only takes a key-value array and uses them to create **<param>** elements. You will see an example of this shortly. Lastly, there is the option of providing additional content to the object. Now for an example which utilizes all arguments.

```
echo $this->htmlObject(
    '/path/to/file.ext',
    'mime/type',
    array(
        'attr1' => 'aval1',
        'attr2' => 'aval2'
    ),
    array(
        'param1' => 'pval1',
        'param2' => 'pval2'
    ),
    'some content'
);

/*
This would output:

<object data="/path/to/file.ext" type="mime/type"
    attr1="aval1" attr2="aval2">
    <param name="param1" value="pval1" />
    <param name="param2" value="pval2" />
    some content
</object>
*/
```

299.11 Identity Helper

The `Identity` helper allows for getting the identity from the `AuthenticationService`.

For the `Identity` helper to work, a `Zend\Authentication\AuthenticationService` name or alias must be defined and recognized by the `ServiceManager`.

`Identity` returns the identity in the `AuthenticationService` or *null* if no identity is available.

As an example:

```
<?php
    if ($user = $this->identity()) {
        echo 'Logged in as ' . $this->escapeHtml($user->getUsername());
    } else {
        echo 'Not logged in';
    }
?>
```

When invoked, the `Identity` plugin will look for a service by the name or alias `Zend\Authentication\AuthenticationService` in the `ServiceManager`. You can provide this service to the `ServiceManager` in a configuration file:

```
// In a configuration file...
return array(
    'service_manager' => array(
        'alias' => array(
            'Zend\Authentication\AuthenticationService' => 'my_auth_service',
        ),
        'invokables' => array(
            'my_auth_service' => 'Zend\Authentication\AuthenticationService';
            },
        ),
    ),
);
```

299.12 InlineScript Helper

The *HTML* **<script>** element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `InlineScript` helper allows you to manage both. It is derived from *HeadScript*, and any method of that helper is available; however, use the `inlineScript()` method in place of `headScript()`.

Note: Use InlineScript for HTML Body Scripts

`InlineScript`, should be used when you wish to include scripts inline in the *HTML* **body**. Placing scripts at the end of your document is a good practice for speeding up delivery of your page, particularly when using 3rd party analytics scripts.

Some JS libraries need to be included in the *HTML* **head**; use *HeadScript* for those scripts.

299.13 JSON Helper

When creating views that return *JSON*, it's important to also set the appropriate response header. The *JSON* view helper does exactly that. In addition, by default, it disables layouts (if currently enabled), as layouts generally aren't used with *JSON* responses.

The *JSON* helper sets the following header:

```
Content-Type: application/json
```

Most *AJAX* libraries look for this header when parsing responses to determine how to handle the content.

Usage of the *JSON* helper is very straightforward:

```
<?php echo $this->json($this->data) ?>
```

Note: Keeping layouts and enabling encoding using Zend\Json\Expr

Each method in the *JSON* helper accepts a second, optional argument. This second argument can be a boolean flag to enable or disable layouts, or an array of options that will be passed to `Zend\Json\Json::encode()` and used internally to encode data.

To keep layouts, the second parameter needs to be boolean `TRUE`. When the second parameter is an array, keeping layouts can be achieved by including a `keepLayouts` key with a value of a boolean `TRUE`.

```
// Boolean true as second argument enables layouts:
echo $this->json($this->data, true);

// Or boolean true as "keepLayouts" key:
echo $this->json($this->data, array('keepLayouts' => true));
```

`Zend\Json\Json::encode` allows the encoding of native *JSON* expressions using `Zend\Json\Expr` objects. This option is disabled by default. To enable this option, pass a boolean `TRUE` to the `enableJsonExprFinder` key of the options array:

```
<?php echo $this->json($this->data, array(
    'enableJsonExprFinder' => true,
    'keepLayouts'          => true,
)) ?>
```

299.14 Partial Helper

The `Partial` view helper is used to render a specified template within its own variable scope. The primary use is for reusable template fragments with which you do not need to worry about variable name clashes. Additionally, they allow you to specify partial view scripts from specific modules.

A sibling to the `Partial`, the `PartialLoop` view helper allows you to pass iterable data, and render a partial for each item.

Note: PartialLoop Counter

The `PartialLoop` view helper assigns a variable to the view named **partialCounter** which passes the current position of the array to the view script. This provides an easy way to have alternating colors on table rows for example.

Basic Usage of Partials

Basic usage of partials is to render a template fragment in its own view scope. Consider the following partial script:

```
<?php // partial.phtml ?>
<ul>
    <li>From: <?php echo $this->escapeHtml($this->from) ?></li>
    <li>Subject: <?php echo $this->escapeHtml($this->subject) ?></li>
</ul>
```

You would then call it from your view script using the following:

```
<?php echo $this->partial('partial.phtml', array(
    'from' => 'Team Framework',
    'subject' => 'view partials')); ?>
```

Which would then render:

```
<ul>
    <li>From: Team Framework</li>
    <li>Subject: view partials</li>
</ul>
```

Note: What is a model?

A model used with the `Partial` view helper can be one of the following:

- **Array**. If an array is passed, it should be associative, as its key/value pairs are assigned to the view with keys as view variables.
- **Object implementing toArray() method**. If an object is passed an has a `toArray()` method, the results of `toArray()` will be assigned to the view object as view variables.
- **Standard object**. Any other object will assign the results of `get_object_vars()` (essentially all public properties of the object) to the view object.

If your model is an object, you may want to have it passed **as an object** to the partial script, instead of serializing it to an array of variables. You can do this by setting the 'objectKey' property of the appropriate helper:

```
// Tell partial to pass objects as 'model' variable
$view->partial()->setObjectKey('model');

// Tell partial to pass objects from partialLoop as 'model' variable
// in final partial view script:
$view->partialLoop()->setObjectKey('model');
```

This technique is particularly useful when passing `Zend\Db\ResultSet\ResultSets` to `partialLoop()`, as you then have full access to your row objects within the view scripts, allowing you to call methods on them (such as retrieving values from parent or dependent rows).

Using PartialLoop to Render Iterable Models

Typically, you'll want to use partials in a loop, to render the same content fragment many times; this way you can put large blocks of repeated content or complex display logic into a single location. However this has a performance impact, as the partial helper needs to be invoked once for each iteration.

The `PartialLoop` view helper helps solve this issue. It allows you to pass an iterable item (array or object implementing **Iterator**) as the model. It then iterates over this, passing, the items to the partial script as the model. Items in the iterator may be any model the `Partial` view helper allows.

Let's assume the following partial view script:

```
<?php // partialLoop.phtml ?>
    <dt><?php echo $this->key ?></dt>
    <dd><?php echo $this->value ?></dd>
```

And the following "model":

```
$model = array(
    array('key' => 'Mammal', 'value' => 'Camel'),
    array('key' => 'Bird', 'value' => 'Penguin'),
    array('key' => 'Reptile', 'value' => 'Asp'),
    array('key' => 'Fish', 'value' => 'Flounder'),
);
```

In your view script, you could then invoke the `PartialLoop` helper:

```
<dl>
<?php echo $this->partialLoop('partialLoop.phtml', $model) ?>
</dl>
```

```
<dl>
    <dt>Mammal</dt>
    <dd>Camel</dd>
```

```

    <dt>Bird</dt>
    <dd>Penguin</dd>

    <dt>Reptile</dt>
    <dd>Asp</dd>

    <dt>Fish</dt>
    <dd>Flounder</dd>
</dl>
```

Rendering Partials in Other Modules

Sometime a partial will exist in a different module. If you know the name of the module, you can pass it as the second argument to either `partial()` or `partialLoop()`, moving the `$model` argument to third position.

For instance, if there's a pager partial you wish to use that's in the 'list' module, you could grab it as follows:

```
<?php echo $this->partial('pager.phtml', 'list', $pagerData) ?>
```

In this way, you can re-use partials created specifically for other modules. That said, it's likely a better practice to put re-usable partials in shared view script paths.

299.15 Placeholder Helper

The `Placeholder` view helper is used to persist content between view scripts and view instances. It also offers some useful features such as aggregating content, capturing view script content for later use, and adding pre- and post-text to content (and custom separators for aggregated content).

Basic Usage of Placeholders

Basic usage of placeholders is to persist view data. Each invocation of the `Placeholder` helper expects a placeholder name; the helper then returns a placeholder container object that you can either manipulate or simply echo out.

```
<?php $this->placeholder('foo')->set("Some text for later") ?>

<?php
    echo $this->placeholder('foo');
    // outputs "Some text for later"
?>
```

Using Placeholders to Aggregate Content

Aggregating content via placeholders can be useful at times as well. For instance, your view script may have a variable array from which you wish to retrieve messages to display later; a later view script can then determine how those will be rendered.

The `Placeholder` view helper uses containers that extend `ArrayObject`, providing a rich feature set for manipulating arrays. In addition, it offers a variety of methods for formatting the content stored in the container:

- `setPrefix($prefix)` sets text with which to prefix the content. Use `getPrefix()` at any time to determine what the current setting is.

- `setPostfix($prefix)` sets text with which to append the content. Use `getPostfix()` at any time to determine what the current setting is.
- `setSeparator($prefix)` sets text with which to separate aggregated content. Use `getSeparator()` at any time to determine what the current setting is.
- `setIndent($prefix)` can be used to set an indentation value for content. If an integer is passed, that number of spaces will be used; if a string is passed, the string will be used. Use `getIndent()` at any time to determine what the current setting is.

```
<!-- first view script -->
<?php $this->placeholder('foo')->exchangeArray($this->data) ?>
```

```
<!-- later view script -->
<?php
$this->placeholder('foo')->setPrefix("<ul>\n    <li>")
                         ->setSeparator("</li><li>\n")
                         ->setIndent(4)
                         ->setPostfix("</li></ul>\n");
?>

<?php
    echo $this->placeholder('foo');
    // outputs as unordered list with pretty indentation
?>
```

Because the `Placeholder` container objects extend `ArrayObject`, you can also assign content to a specific key in the container easily, instead of simply pushing it into the container. Keys may be accessed either as object properties or as array keys.

```
<?php $this->placeholder('foo')->bar = $this->data ?>
<?php echo $this->placeholder('foo')->bar ?>

<?php
$foo = $this->placeholder('foo');
echo $foo['bar'];
?>
```

Using Placeholders to Capture Content

Occasionally you may have content for a placeholder in a view script that is easiest to template; the `Placeholder` view helper allows you to capture arbitrary content for later rendering using the following *API*.

- `captureStart($type, $key)` begins capturing content.

  `$type` should be one of the `Placeholder` constants `APPEND` or `SET`. If `APPEND`, captured content is appended to the list of current content in the placeholder; if `SET`, captured content is used as the sole value of the placeholder (potentially replacing any previous content). By default, `$type` is `APPEND`.

  `$key` can be used to specify a specific key in the placeholder container to which you want content captured.

  `captureStart()` locks capturing until `captureEnd()` is called; you cannot nest capturing with the same placeholder container. Doing so will raise an exception.

- `captureEnd()` stops capturing content, and places it in the container object according to how `captureStart()` was called.

```
<!-- Default capture: append -->
<?php $this->placeholder('foo')->captureStart();
```

```
foreach ($this->data as $datum): ?>
<div class="foo">
    <h2><?php echo $datum->title ?></h2>
    <p><?php echo $datum->content ?></p>
</div>
<?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>

<?php echo $this->placeholder('foo') ?>
```

```
<!-- Capture to key -->
<?php $this->placeholder('foo')->captureStart('SET', 'data');
foreach ($this->data as $datum): ?>
<div class="foo">
    <h2><?php echo $datum->title ?></h2>
    <p><?php echo $datum->content ?></p>
</div>
 <?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>

<?php echo $this->placeholder('foo')->data ?>
```

299.15.1 Concrete Placeholder Implementations

Zend Framework ships with a number of "concrete" placeholder implementations. These are for commonly used placeholders: doctype, page title, and various <head> elements. In all cases, calling the placeholder with no arguments returns the element itself.

Documentation for each element is covered separately, as linked below:

- *Doctype*
- *HeadLink*
- *HeadMeta*
- *HeadScript*
- *HeadStyle*
- *HeadTitle*
- *InlineScript*

299.16 URL Helper

- `url($name, $urlParams, $routeOptions, $reuseMatchedParams)`: Creates a *URL* string based on a named route. `$urlParams` should be an associative array of key/value pairs used by the particular route.

```
// In a configuration array (e.g. returned by some module's module.config.php)
'router' => array(
    'routes' => array(
        'auth' => array(
            'type'    => 'segment',
            'options' => array(
                'route'       => '/auth[/:action][/:id]',
```

```
                'constraints' => array(
                    'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                ),
                'defaults' => array(
                    'controller' => 'auth',
                    'action'     => 'index',
                ),
            )
        )
    )
),

// In a view script:
<a href="<?php echo $this->url('auth', array('action' => 'logout', 'id' => 100)); ?>">Logout</a>
```

Output:

```
<a href="/auth/logout/100">Logout</a>
```

CHAPTER

ADVANCED USAGE OF HELPERS

300.1 Registering Helpers

`Zend\View\Renderer\PhpRenderer` composes a *plugin manager* for managing helpers, specifically an instance of `Zend\View\HelperPluginManager`, which extends `Zend\ServiceManager\AbstractPluginManager`, and this extends `Zend\ServiceManager\ServiceManager`. As you can see, the *HelperPluginManager* is a specialized service manager, so you can register a helper/plugin like any other service (see *the Service Manager documentation* for more information).

Programmatically, this is done as follows:

```
// $view is an instance of PhpRenderer
$pluginManager = $view->getHelperPluginManager();

// Register as a invokable class:
$pluginManager->setInvokableClass('lowercase', 'MyModule\View\Helper\LowerCase');

// Register as a factory:
$pluginManager->setFactory('lowercase', function ($pluginManager) {
   $lowercaseHelper = new MyModule\View\Helper\LowerCase;

   // ...do some configuration or dependency injection...

   return $lowercaseHelper;
});
```

Within an MVC application, you will typically simply pass a map of plugins to the class via your configuration.

```
// From within a configuration file
return array(
   'view_helpers' => array(
      'invokables' => array(
         'lowercase' => 'MyModule\View\Helper\LowerCase',
         'uppercase' => 'MyModule\View\Helper\UpperCase',
      ),
   ),
);
```

If your module class implements `Zend\ModuleManager\Feature\ViewHelperProviderInterface`, or just the method `getViewHelperConfig()`, you could do the following (it's the same as the previous example).

```
namespace MyModule;

class Module
{
```

```
    public function getAutoloaderConfig(){ /*common code*/ }
    public function getConfig(){ /*common code*/ }

    public function getViewHelperConfig()
    {
        return array(
           'invokables' => array(
              'lowercase' => 'MyModule\View\Helper\LowerCase',
              'uppercase' => 'MyModule\View\Helper\UpperCase',
           ),
        );
   }
}
```

The two latter examples can be done in each module that needs to register helpers with the `PhpRenderer`; however, be aware that another module can register helpers with the same name, so order of modules can impact which helper class will actually be registered!

300.2 Writing Custom Helpers

Writing custom helpers is easy. We recommend extending `Zend\View\Helper\AbstractHelper`, but at the minimum, you need only implement the `Zend\View\Helper\HelperInterface` interface:

```
namespace Zend\View\Helper;

use Zend\View\Renderer\RendererInterface as Renderer;

interface HelperInterface
{
    /**
     * Set the View object
     *
     * @param  Renderer $view
     * @return HelperInterface
     */
    public function setView(Renderer $view);

    /**
     * Get the View object
     *
     * @return Renderer
     */
    public function getView();
}
```

If you want your helper to be capable of being invoked as if it were a method call of the `PhpRenderer`, you should also implement an `__invoke()` method within your helper.

As previously noted, we recommend extending `Zend\View\Helper\AbstractHelper`, as it implements the methods defined in `HelperInterface`, giving you a headstart in your development.

Once you have defined your helper class, make sure you can autoload it, and then *register it with the plugin manager*.

Here is an example helper, which we're titling "SpecialPurpose"

```
// /module/src/MyModule/View/Helper/SpecialPurpose.php
namespace MyModule\View\Helper;
```

```

use Zend\View\Helper\AbstractHelper;

class SpecialPurpose extends AbstractHelper
{
    protected $count = 0;

    public function __invoke()
    {
        $this->count++;
        $output = sprintf("I have seen 'The Jerk' %d time(s).", $this->count);
        return htmlspecialchars($output, ENT_QUOTES, 'UTF-8');
    }
}
```

Then assume that we *register it with the plugin manager*, by the name "specialpurpose".

Within a view script, you can call the `SpecialPurpose` helper as many times as you like; it will be instantiated once, and then it persists for the life of that `PhpRenderer` instance.

```
// remember, in a view script, $this refers to the Zend\View\Renderer\PhpRenderer instance.
echo $this->specialPurpose();
echo $this->specialPurpose();
echo $this->specialPurpose();
```

The output would look something like this:

```
I have seen 'The Jerk' 1 time(s).
I have seen 'The Jerk' 2 time(s).
I have seen 'The Jerk' 3 time(s).
```

Sometimes you will need access to the calling `PhpRenderer` object – for instance, if you need to use the registered encoding, or want to render another view script as part of your helper. This is why we define the `setView()` and `getView()` methods. As an example, we could rewrite the `SpecialPurpose` helper as follows to take advantage of the `EscapeHtml` helper:

```
namespace MyModule\View\Helper;

use Zend\View\Helper\AbstractHelper;

class SpecialPurpose extends AbstractHelper
{
    protected $count = 0;

    public function __invoke()
    {
        $this->count++;
        $output  = sprintf("I have seen 'The Jerk' %d time(s).", $this->count);
        $escaper = $this->getView()->plugin('escapehtml');
        return $escaper($output);
    }
}
```

300.3 Registering Concrete Helpers

Sometimes it is convenient to instantiate a view helper, and then register it with the renderer. This can be done by injecting it directly into the plugin manager.

```
// $view is a PhpRenderer instance

$helper = new MyModule\View\Helper\LowerCase;
// ...do some configuration or dependency injection...

$view->getHelperPluginManager()->setService('lowercase', $helper);
```

The plugin manager will validate the helper/plugin, and if the validation passes, the helper/plugin will be registered.

CHAPTER

ONE

INTRODUCTION TO ZEND\XMLRPC

From its home page, *XML-RPC* is described as a "...remote procedure calling using *HTTP* as the transport and *XML* as the encoding. *XML-RPC* is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned."

Zend Framework provides support for both consuming remote *XML-RPC* services and building new *XML-RPC* servers.

301.1 Quick Start

To show how easy is to create *XML-RPC* services with `Zend\XmlRpc\Server`, take a look at the following example:

```
class Greeter
{

    /**
    * Say hello to someone.
    *
    * @param string $name Who to greet
    * @return string
    */
    public function sayHello($name='Stranger')
    {
        return sprintf("Hello %s!", $name);
    }
}

$server = new Zend\XmlRpc\Server;
// Our Greeter class will be called
// greeter from the client
$server->setClass('Greeter', 'greeter');
$server->handle();
```

Note: It is necessary to write function and method docblocks for the services which are to be exposed via `Zend\XmlRpc\Server`, as it will be used to validate parameters provided to the methods, and also to determine the method help text and method signatures.

An example of a client consuming this *XML-RPC* service would be something like this:

```
$client = new Zend\XmlRpc\Client('http://example.com/xmlrpcserver.php');

echo $client->call('greeter.sayHello');
// will output "Hello Stranger!"

echo $client->call('greeter.sayHello', array('Dude'));
// will output "Hello Dude!"
```

CHAPTER

TWO

ZEND\XMLRPC\CLIENT

302.1 Introduction

Zend Framework provides support for consuming remote *XML-RPC* services as a client in the `Zend\XmlRpc\Client` package. Its major features include automatic type conversion between *PHP* and *XML-RPC*, a server proxy object, and access to server introspection capabilities.

302.2 Method Calls

The constructor of `Zend\XmlRpc\Client` receives the *URL* of the remote *XML-RPC* server endpoint as its first parameter. The new instance returned may be used to call any number of remote methods at that endpoint.

To call a remote method with the *XML-RPC* client, instantiate it and use the `call()` instance method. The code sample below uses a demonstration *XML-RPC* server on the Zend Framework website. You can use it for testing or exploring the `Zend\XmlRpc` components.

XML-RPC Method Call

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

echo $client->call('test.sayHello');

// hello
```

The *XML-RPC* value returned from the remote method call will be automatically unmarshaled and cast to the equivalent *PHP* native type. In the example above, a *PHP* `String` is returned and is immediately ready to be used.

The first parameter of the `call()` method receives the name of the remote method to call. If the remote method requires any parameters, these can be sent by supplying a second, optional parameter to `call()` with an `Array` of values to pass to the remote method:

XML-RPC Method Call with Parameters

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$arg1 = 1.1;
$arg2 = 'foo';

```

```
$result = $client->call('test.sayHello', array($arg1, $arg2));

// $result is a native PHP type
```

If the remote method doesn't require parameters, this optional parameter may either be left out or an empty `array()` passed to it. The array of parameters for the remote method can contain native *PHP* types, `Zend\XmlRpc\Value` objects, or a mix of each.

The `call()` method will automatically convert the *XML-RPC* response and return its equivalent *PHP* native type. A `Zend\XmlRpc\Response` object for the return value will also be available by calling the `getLastResponse()` method after the call.

302.3 Types and Conversions

Some remote method calls require parameters. These are given to the `call()` method of `Zend\XmlRpc\Client` as an array in the second parameter. Each parameter may be given as either a native *PHP* type which will be automatically converted, or as an object representing a specific *XML-RPC* type (one of the `Zend\XmlRpc\Value` objects).

302.3.1 PHP Native Types as Parameters

Parameters may be passed to `call()` as native *PHP* variables, meaning as a `String`, `Integer`, `Float`, `Boolean`, `Array`, or an `Object`. In this case, each *PHP* native type will be auto-detected and converted into one of the *XML-RPC* types according to this table:

Table 302.1: PHP and XML-RPC Type Conversions

PHP Native Type	XML-RPC Type
integer	int
Zend\Math\BigInteger\BigInteger	i8
double	double
boolean	boolean
string	string
null	nil
array	array
associative array	struct
object	array
DateTime	dateTime.iso8601
DateTime	dateTime.iso8601

Note: What type do empty arrays get cast to?

Passing an empty array to an *XML-RPC* method is problematic, as it could represent either an array or a struct. `Zend\XmlRpc\Client` detects such conditions and makes a request to the server's `system.methodSignature` method to determine the appropriate *XML-RPC* type to cast to.

However, this in itself can lead to issues. First off, servers that do not support `system.methodSignature` will log failed requests, and `Zend\XmlRpc\Client` will resort to casting the value to an *XML-RPC* array type. Additionally, this means that any call with array arguments will result in an additional call to the remote server.

To disable the lookup entirely, you can call the `setSkipSystemLookup()` method prior to making your *XML-RPC* call:

```
$client->setSkipSystemLookup(true);
$result = $client->call('foo.bar', array(array()));
```

302.3.2 Zend\XmlRpc\Value Objects as Parameters

Parameters may also be created as `Zend\XmlRpc\Value` instances to specify an exact *XML-RPC* type. The primary reasons for doing this are:

- When you want to make sure the correct parameter type is passed to the procedure (i.e. the procedure requires an integer and you may get it from a database as a string)
- When the procedure requires `base64` or `dateTime.iso8601` type (which doesn't exists as a *PHP* native type)
- When auto-conversion may fail (i.e. you want to pass an empty *XML-RPC* struct as a parameter. Empty structs are represented as empty arrays in *PHP* but, if you give an empty array as a parameter it will be auto-converted to an *XML-RPC* array since it's not an associative array)

There are two ways to create a `Zend\XmlRpc\Value` object: instantiate one of the `Zend\XmlRpc\Value` subclasses directly, or use the static factory method `Zend\XmlRpc\AbstractValue::getXmlRpcValue()`.

Table 302.2: Zend\XmlRpc\Value Objects for XML-RPC Types

XML-RPC Type	Zend\XmlRpc\AbstractValue Constant	Zend\XmlRpc\Value Object
int	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_INTEGER	Zend\XmlRpc\Value\Integer
i4	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_I4	Zend\XmlRpc\Value\Integer
i8	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_I8	Zend\XmlRpc\Value\BigInteger
ex:i8	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_APACHEI8	Zend\XmlRpc\Value\BigInteger
double	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_DOUBLE	Zend\XmlRpc\Value_Double
boolean	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_BOOLEAN	Zend\XmlRpc\Value\Boolean
string	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_STRING	Zend\XmlRpc\Value\String
nil	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_NIL	Zend\XmlRpc\Value\Nil
ex:nil	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_APACHENIL	Zend\XmlRpc\Value\Nil
base64	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_BASE64	Zend\XmlRpc\Value\Base64
dateTime.iso8601	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_DATETIME	Zend\XmlRpc\Value\DateTime
array	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_ARRAY	Zend\XmlRpc\Value\Array
struct	Zend\XmlRpc\AbstractValue::XMLRPC_TYPE_STRUCT	Zend\XmlRpc\Value\Struct

Note: Automatic Conversion

When building a new `Zend\XmlRpc\Value` object, its value is set by a *PHP* type. The *PHP* type will be converted to the specified type using *PHP* casting. For example, if a string is given as a value to the `Zend\XmlRpc\Value\Integer` object, it will be converted using `(int)$value`.

302.4 Server Proxy Object

Another way to call remote methods with the *XML-RPC* client is to use the server proxy. This is a *PHP* object that proxies a remote *XML-RPC* namespace, making it work as close to a native *PHP* object as possible.

To instantiate a server proxy, call the `getProxy()` instance method of `Zend\XmlRpc\Client`. This will return an instance of `Zend\XmlRpc\Client\ServerProxy`. Any method call on the server proxy object will be forwarded to the remote, and parameters may be passed like any other *PHP* method.

Proxy the Default Namespace

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$service = $client->getProxy();           // Proxy the default namespace

$hello = $service->test->sayHello(1, 2);  // test.Hello(1, 2) returns "hello"
```

The `getProxy()` method receives an optional argument specifying which namespace of the remote server to proxy. If it does not receive a namespace, the default namespace will be proxied. In the next example, the 'test' namespace will be proxied:

Proxy Any Namespace

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$test  = $client->getProxy('test');     // Proxy the "test" namespace

$hello = $test->sayHello(1, 2);         // test.Hello(1,2) returns "hello"
```

If the remote server supports nested namespaces of any depth, these can also be used through the server proxy. For example, if the server in the example above had a method `test.foo.bar()`, it could be called as `$test->foo->bar()`.

302.5 Error Handling

Two kinds of errors can occur during an *XML-RPC* method call: *HTTP* errors and *XML-RPC* faults. The `Zend\XmlRpc\Client` recognizes each and provides the ability to detect and trap them independently.

302.5.1 HTTP Errors

If any *HTTP* error occurs, such as the remote *HTTP* server returns a **404 Not Found**, a `Zend\XmlRpc\Client\Exception\HttpException` will be thrown.

Handling HTTP Errors

```
$client = new Zend\XmlRpc\Client('http://foo/404');

try {

    $client->call('bar', array($arg1, $arg2));

} catch (Zend\XmlRpc\Client\Exception\HttpException $e) {

    // $e->getCode() returns 404
    // $e->getMessage() returns "Not Found"

}
```

Regardless of how the *XML-RPC* client is used, the `Zend\XmlRpc\Client\Exception\HttpException` will be thrown whenever an *HTTP* error occurs.

302.5.2 XML-RPC Faults

An *XML-RPC* fault is analogous to a *PHP* exception. It is a special type returned from an *XML-RPC* method call that has both an error code and an error message. *XML-RPC* faults are handled differently depending on the context of how the `Zend\XmlRpc\Client` is used.

When the `call()` method or the server proxy object is used, an *XML-RPC* fault will result in a `Zend\XmlRpc\Client\Exception\FaultException` being thrown. The code and message of the exception will map directly to their respective values in the original *XML-RPC* fault response.

Handling XML-RPC Faults

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

try {

    $client->call('badMethod');

} catch (Zend\XmlRpc\Client\Exception\FaultException $e) {

    // $e->getCode() returns 1
    // $e->getMessage() returns "Unknown method"

}
```

When the `call()` method is used to make the request, the `Zend\XmlRpc\Client\Exception\FaultException` will be thrown on fault. A `Zend\XmlRpc\Response` object containing the fault will also be available by calling `getLastResponse()`.

When the `doRequest()` method is used to make the request, it will not throw the exception. Instead, it will return a `Zend\XmlRpc\Response` object returned will containing the fault. This can be checked with `isFault()` instance method of `Zend\XmlRpc\Response`.

302.6 Server Introspection

Some *XML-RPC* servers support the de facto introspection methods under the *XML-RPC* **system.** namespace. `Zend\XmlRpc\Client` provides special support for servers with these capabilities.

A `Zend\XmlRpc\Client\ServerIntrospection` instance may be retrieved by calling the `getIntrospector()` method of `Zend\XmlRpc\Client`. It can then be used to perform introspection operations on the server.

```
$client = new Zend\XmlRpc\Client('http://example.com/xmlrpcserver.php');
$introspector = $client->getIntrospector();
foreach ($introspector->listMethods() as $method) {
    echo "Method: " . $method . "\n";
}
```

The following methods are available for introspection:

- `getSignatureForEachMethod`: Returns the signature for each method on the server
- `getSignatureForEachMethodByMulticall($methods=null)`: Attempt to get the method signatures in one request via system.multicall(). Optionally pass an array of method names.

- `getSignatureForEachMethodByLooping($methods=null)`: Get the method signatures for every method by successively calling system.methodSignature. Optionally pass an array of method names
- `getMethodSignature($method)`: Get the method's signature for $method
- `listMethods`: List all methods on the server

302.7 From Request to Response

Under the hood, the `call()` instance method of `Zend\XmlRpc\Client` builds a request object (`Zend\XmlRpc\Request`) and sends it to another method, `doRequest()`, that returns a response object (`Zend\XmlRpc\Response`).

The `doRequest()` method is also available for use directly:

Processing Request to Response

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$request = new Zend\XmlRpc\Request();
$request->setMethod('test.sayHello');
$request->setParams(array('foo', 'bar'));

$client->doRequest($request);

// $client->getLastRequest() returns instanceof Zend\XmlRpc\Request
// $client->getLastResponse() returns instanceof Zend\XmlRpc\Response
```

Whenever an *XML-RPC* method call is made by the client through any means, either the `call()` method, `doRequest()` method, or server proxy, the last request object and its resultant response object will always be available through the methods `getLastRequest()` and `getLastResponse()` respectively.

302.8 HTTP Client and Testing

In all of the prior examples, an *HTTP* client was never specified. When this is the case, a new instance of `Zend\Http\Client` will be created with its default options and used by `Zend\XmlRpc\Client` automatically.

The *HTTP* client can be retrieved at any time with the `getHttpClient()` method. For most cases, the default *HTTP* client will be sufficient. However, the `setHttpClient()` method allows for a different *HTTP* client instance to be injected.

The `setHttpClient()` is particularly useful for unit testing. When combined with the `Zend\Http\Client\Adapter\Test`, remote services can be mocked out for testing. See the unit tests for `Zend\XmlRpc\Client` for examples of how to do this.

CHAPTER

THREE

ZEND\XMLRPC\SERVER

303.1 Introduction

`Zend\XmlRpc\Server` is intended as a fully-featured *XML-RPC* server, following the specifications outlined at www.xmlrpc.com. Additionally, it implements the `system.multicall()` method, allowing boxcarring of requests.

303.2 Basic Usage

An example of the most basic use case:

```
$server = new Zend\XmlRpc\Server();
$server->setClass('My\Service\Class');
echo $server->handle();
```

303.3 Server Structure

`Zend\XmlRpc\Server` is composed of a variety of components, ranging from the server itself to request, response, and fault objects.

To bootstrap `Zend\XmlRpc\Server`, the developer must attach one or more classes or functions to the server, via the `setClass()` and `addFunction()` methods.

Once done, you may either pass a `Zend\XmlRpc\Request` object to `Zend\XmlRpc\Server::handle()`, or it will instantiate a `Zend\XmlRpc\Request\Http` object if none is provided – thus grabbing the request from `php://input`.

`Zend\XmlRpc\Server::handle()` then attempts to dispatch to the appropriate handler based on the method requested. It then returns either a `Zend\XmlRpc\Response`-based object or a `Zend\XmlRpc\Server\Fault`object. These objects both have `__toString()` methods that create valid *XML-RPC XML* responses, allowing them to be directly echoed.

303.4 Anatomy of a webservice

303.4.1 General considerations

For maximum performance it is recommended to use a simple bootstrap file for the server component. Using `Zend\XmlRpc\Server` inside a *Zend\Controller* is strongly discouraged to avoid the overhead.

Services change over time and while webservices are generally less change intense as code-native *APIs*, it is recommended to version your service. Do so to lay grounds to provide compatibility for clients using older versions of your service and manage your service lifecycle including deprecation timeframes. To do so just include a version number into your *URI*. It is also recommended to include the remote protocol name in the *URI* to allow easy integration of upcoming remoting technologies. http://myservice.ws/**1.0/XMLRPC/**.

303.4.2 What to expose?

Most of the time it is not sensible to expose business objects directly. Business objects are usually small and under heavy change, because change is cheap in this layer of your application. Once deployed and adopted, web services are hard to change. Another concern is *I/O* and latency: the best webservice calls are those not happening. Therefore service calls need to be more coarse-grained than usual business logic is. Often an additional layer in front of your business objects makes sense. This layer is sometimes referred to as Remote Facade. Such a service layer adds a coarse grained interface on top of your business logic and groups verbose operations into smaller ones.

303.5 Conventions

`Zend\XmlRpc\Server` allows the developer to attach functions and class method calls as dispatchable *XML-RPC* methods. Via `Zend\Server\Reflection`, it does introspection on all attached methods, using the function and method docblocks to determine the method help text and method signatures.

XML-RPC types do not necessarily map one-to-one to *PHP* types. However, the code will do its best to guess the appropriate type based on the values listed in @param and @return lines. Some *XML-RPC* types have no immediate *PHP* equivalent, however, and should be hinted using the *XML-RPC* type in the PHPDoc. These include:

- **dateTime.iso8601**, a string formatted as '`YYYYMMDDTHH:mm:ss`'
- **base64**, base64 encoded data
- **struct**, any associative array

An example of how to hint follows:

```
/**
* This is a sample function
*
* @param base64 $val1 Base64-encoded data
* @param dateTime.iso8601 $val2 An ISO date
* @param struct $val3 An associative array
* @return struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

PhpDocumentor does no validation of the types specified for params or return values, so this will have no impact on your *API* documentation. Providing the hinting is necessary, however, when the server is validating the parameters provided to the method call.

It is perfectly valid to specify multiple types for both params and return values; the *XML-RPC* specification even suggests that system.methodSignature should return an array of all possible method signatures (i.e., all possible combinations of param and return values). You may do so just as you normally would with PhpDocumentor, using the '|' operator:

```
/**
* This is a sample function
*
* @param string|base64 $val1 String or base64-encoded data
* @param string|dateTime.iso8601 $val2 String or an ISO date
* @param array|struct $val3 Normal indexed array or an associative array
* @return boolean|struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

Note: Allowing multiple signatures can lead to confusion for developers using the services; to keep things simple, a *XML-RPC* service method should only have a single signature.

303.6 Utilizing Namespaces

XML-RPC has a concept of namespacing; basically, it allows grouping *XML-RPC* methods by dot-delimited namespaces. This helps prevent naming collisions between methods served by different classes. As an example, the *XML-RPC* server is expected to server several methods in the 'system' namespace:

- system.listMethods
- system.methodHelp
- system.methodSignature

Internally, these map to the methods of the same name in `Zend\XmlRpc\Server`.

If you want to add namespaces to the methods you serve, simply provide a namespace to the appropriate method when attaching a function or class:

```
// All public methods in My_Service_Class will be accessible as
// myservice.METHODNAME
$server->setClass('My\Service\Class', 'myservice');

// Function 'somefunc' will be accessible as funcs.somefunc
$server->addFunction('somefunc', 'funcs');
```

303.7 Custom Request Objects

Most of the time, you'll simply use the default request type included with `Zend\XmlRpc\Server`, `Zend\XmlRpc\Request\Http`. However, there may be times when you need *XML-RPC* to be available via the *CLI*, a *GUI*, or other environment, or want to log incoming requests. To do so, you may create a custom request object that extends `Zend\XmlRpc\Request`. The most important thing to remember is to ensure that the `getMethod()` and `getParams()` methods are implemented so that the *XML-RPC* server can retrieve that information in order to dispatch the request.

303.8 Custom Responses

Similar to request objects, `Zend\XmlRpc\Server` can return custom response objects; by default, a `Zend\XmlRpc\Response\Http` object is returned, which sends an appropriate Content-Type *HTTP* header for use with *XML-RPC*. Possible uses of a custom object would be to log responses, or to send responses back to `STDOUT`.

To use a custom response class, use `Zend\XmlRpc\Server::setResponseClass()` prior to calling `handle()`.

303.9 Handling Exceptions via Faults

`Zend\XmlRpc\Server` catches Exceptions generated by a dispatched method, and generates an *XML-RPC* fault response when such an exception is caught. By default, however, the exception messages and codes are not used in a fault response. This is an intentional decision to protect your code; many exceptions expose more information about the code or environment than a developer would necessarily intend (a prime example includes database abstraction or access layer exceptions).

Exception classes can be whitelisted to be used as fault responses, however. To do so, simply utilize `Zend\XmlRpc\Server\Fault::attachFaultException()` to pass an exception class to whitelist:

```
Zend\XmlRpc\Server\Fault::attachFaultException('My\Project\Exception');
```

If you utilize an exception class that your other project exceptions inherit, you can then whitelist a whole family of exceptions at a time. `Zend\XmlRpc\Server\Exceptions` are always whitelisted, to allow reporting specific internal errors (undefined methods, etc.).

Any exception not specifically whitelisted will generate a fault response with a code of '404' and a message of 'Unknown error'.

303.10 Caching Server Definitions Between Requests

Attaching many classes to an *XML-RPC* server instance can utilize a lot of resources; each class must introspect using the Reflection *API* (via `Zend\Server\Reflection`), which in turn generates a list of all possible method signatures to provide to the server class.

To reduce this performance hit somewhat, `Zend\XmlRpc\Server\Cache` can be used to cache the server definition between requests. When combined with `__autoload()`, this can greatly increase performance.

An sample usage follows:

```
use Zend\XmlRpc\Server as XmlRpcServer;

// Register the "My\Services" namespace
$loader = new Zend\Loader\StandardAutoloader();
$loader->registerNamespace('My\Services', 'path to My/Services');
$loader->register();

$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
$server = new XmlRpcServer();

if (!XmlRpcServer\Cache::get($cacheFile, $server)) {

    $server->setClass('My\Services\Glue', 'glue');   // glue. namespace
    $server->setClass('My\Services\Paste', 'paste'); // paste. namespace
```

```
    $server->setClass('My\Services\Tape', 'tape');    // tape. namespace

    XmlRpcServer\Cache::save($cacheFile, $server);
}

echo $server->handle();
```

The above example attempts to retrieve a server definition from `xmlrpc.cache` in the same directory as the script. If unsuccessful, it loads the service classes it needs, attaches them to the server instance, and then attempts to create a new cache file with the server definition.

303.11 Usage Examples

Below are several usage examples, showing the full spectrum of options available to developers. Usage examples will each build on the previous example provided.

Basic Usage

The example below attaches a function as a dispatchable *XML-RPC* method and handles incoming calls.

```
/**
 * Return the MD5 sum of a value
 *
 * @param string $value Value to md5sum
 * @return string MD5 sum of value
 */
function md5Value($value)
{
    return md5($value);
}

$server = new Zend\XmlRpc\Server();
$server->addFunction('md5Value');
echo $server->handle();
```

Attaching a class

The example below illustrates attaching a class' public methods as dispatchable *XML-RPC* methods.

```
require_once 'Services/Comb.php';

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb');
echo $server->handle();
```

Attaching a class with arguments

The following example illustrates how to attach a class' public methods and passing arguments to its methods. This can be used to specify certain defaults when registering service classes.

```
namespace Services;

class PricingService
{
    /**
     * Calculate current price of product with $productId
     *
     * @param ProductRepository $productRepository
     * @param PurchaseRepository $purchaseRepository
     * @param integer $productId
     */
    public function calculate(ProductRepository $productRepository,
                              PurchaseRepository $purchaseRepository,
                              $productId)
    {
        ...
    }
}

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\PricingService',
                  'pricing',
                  new ProductRepository(),
                  new PurchaseRepository());
```

The arguments passed at `setClass()` at server construction time are injected into the method call `pricing.calculate()` on remote invokation. In the example above, only the argument `$purchaseId` is expected from the client.

Passing arguments only to constructor

`Zend\XmlRpc\Server` allows to restrict argument passing to constructors only. This can be used for constructor dependency injection. To limit injection to constructors, call `sendArgumentsToAllMethods` and pass `FALSE` as an argument. This disables the default behavior of all arguments being injected into the remote method. In the example below the instance of `ProductRepository` and `PurchaseRepository` is only injected into the constructor of `Services_PricingService2`.

```
class Services\PricingService2
{
    /**
     * @param ProductRepository $productRepository
     * @param PurchaseRepository $purchaseRepository
     */
    public function __construct(ProductRepository $productRepository,
                                PurchaseRepository $purchaseRepository)
    {
        ...
    }

    /**
     * Calculate current price of product with $productId
     *
     * @param integer $productId
     * @return double
     */
    public function calculate($productId)
    {
```

```
        ...
    }
}

$server = new Zend\XmlRpc\Server();
$server->sendArgumentsToAllMethods(false);
$server->setClass('Services\PricingService2',
                  'pricing',
                  new ProductRepository(),
                  new PurchaseRepository());
```

Attaching a class instance

`setClass()` allows to register a previously instantiated class at the server. Just pass an instance instead of the class name. Obviously passing arguments to the constructor is not possible with pre-instantiated classes.

Attaching several classes using namespaces

The example below illustrates attaching several classes, each with their own namespace.

```
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb', 'comb');   // methods called as comb.*
$server->setClass('Services\Brush', 'brush'); // methods called as brush.*
$server->setClass('Services\Pick', 'pick');   // methods called as pick.*
echo $server->handle();
```

Specifying exceptions to use as valid fault responses

The example below allows any `Services\Exception`-derived class to report its code and message in the fault response.

```
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb', 'comb');   // methods called as comb.*
$server->setClass('Services\Brush', 'brush'); // methods called as brush.*
$server->setClass('Services\Pick', 'pick');   // methods called as pick.*
echo $server->handle();
```

Utilizing custom request and response objects

Some use cases require to utilize a custom request object. For example, *XML/RPC* is not bound to *HTTP* as a transfer protocol. It is possible to use other transfer protocols like *SSH* or telnet to send the request and response data over the

wire. Another use case is authentication and authorization. In case of a different transfer protocol, one need to change the implementation to read request data.

The example below instantiates a custom request class and passes it to the server to handle.

```
require_once 'Services/Request.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb', 'comb');   // methods called as comb.*
$server->setClass('Services\Brush', 'brush'); // methods called as brush.*
$server->setClass('Services\Pick', 'pick');   // methods called as pick.*

// Create a request object
$request = new Services\Request();

echo $server->handle($request);
```

Specifying a custom response class

The example below illustrates specifying a custom response class for the returned response.

```
require_once 'Services/Request.php';
require_once 'Services/Response.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb', 'comb');   // methods called as comb.*
$server->setClass('Services\Brush', 'brush'); // methods called as brush.*
$server->setClass('Services\Pick', 'pick');   // methods called as pick.*

// Create a request object
$request = new Services\Request();

// Utilize a custom response
$server->setResponseClass('Services\Response');

echo $server->handle($request);
```

303.12 Performance optimization

Cache server definitions between requests

The example below illustrates caching server definitions between requests.

```
use Zend\XmlRpc\Server as XmlRpcServer;

// Register the "Services" namespace
$loader = new Zend\Loader\StandardAutoloader();
$loader->registerNamespace('Services', 'path to Services');
$loader->register();

// Specify a cache file
$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';

// Allow Services\Exceptions to report as fault responses
XmlRpcServer\Fault::attachFaultException('Services\Exception');

$server = new XmlRpcServer();

// Attempt to retrieve server definition from cache
if (!XmlRpcServer\Cache::get($cacheFile, $server)) {
    $server->setClass('Services\Comb', 'comb');   // methods called as comb.*
    $server->setClass('Services\Brush', 'brush'); // methods called as brush.*
    $server->setClass('Services\Pick', 'pick');   // methods called as pick.*

    // Save cache
    XmlRpcServer\Cache::save($cacheFile, $server);
}

// Create a request object
$request = new Services\Request();

// Utilize a custom response
$server->setResponseClass('Services\Response');

echo $server->handle($request);
```

Note: The server cache file should be located outside the document root.

Optimizing XML generation

`Zend\XmlRpc\Server` uses `DOMDocument` of *PHP* extension **ext/dom** to generate it's *XML* output. While **ext/dom** is available on a lot of hosts it is not exactly the fastest. Benchmarks have shown, that `XmlWriter` from **ext/xmlwriter** performs better.

If **ext/xmlwriter** is available on your host, you can select a the `XmlWriter`-based generator to leverage the performance differences.

```
use Zend\XmlRpc;

XmlRpc\AbstractValue::setGenerator(new XmlRpc\Generator\XmlWriter());

```

```
5 $server = new XmlRpc\Server();
6 ...
```

Note: Benchmark your application

Performance is determined by a lot of parameters and benchmarks only apply for the specific test case. Differences come from *PHP* version, installed extensions, webserver and operating system just to name a few. Please make sure to benchmark your application on your own and decide which generator to use based on **your** numbers.

Note: Benchmark your client

This optimization makes sense for the client side too. Just select the alternate *XML* generator before doing any work with `Zend\XmlRpc\Client`.

CHAPTER

FOUR

ZENDSERVICEAKISMET

304.1 Introduction

`ZendService\Akismet` provides a client for the Akismet API. The Akismet service is used to determine if incoming data is potentially spam. It also exposes methods for submitting data as known spam or as false positives (ham). It was originally intended to help categorize and identify spam for Wordpress, but it can be used for any type of data.

Akismet requires an *API* key for usage. You can get one by signing up for a WordPress.com account. You do not need to activate a blog. Simply acquiring the account will provide you with the *API* key.

Akismet requires that all requests contain a *URL* to the resource for which data is being filtered. Because of Akismet's origins in WordPress, this resource is called the blog *URL*. This value should be passed as the second argument to the constructor, but may be reset at any time using the `setBlogUrl()` method, or overridden by specifying a 'blog' key in the various method calls.

304.2 Verify an API key

`ZendService\Akismet\Akismet::verifyKey($key)` is used to verify that an Akismet *API* key is valid. In most cases, you will not need to check, but if you need a sanity check, or to determine if a newly acquired key is active, you may do so with this method.

```
// Instantiate with the API key and a URL to the application or
// resource being used
$akismet = new ZendService\Akismet\Akismet($apiKey,
                                   'http://framework.zend.com/wiki/');
if ($akismet->verifyKey($apiKey)) {
    echo "Key is valid.\n";
} else {
    echo "Key is not valid\n";
}
```

If called with no arguments, `verifyKey()` uses the *API* key provided to the constructor.

`verifyKey()` implements Akismet's *verify-key* REST method.

304.3 Check for spam

`ZendService\Akismet\Akismet::isSpam($data)` is used to determine if the data provided is considered spam by Akismet. It accepts an associative array as the sole argument. That array requires the following keys be set:

- *user_ip*, the IP address of the user submitting the data (not your IP address, but that of a user on your site).
- *user_agent*, the reported UserAgent string (browser and version) of the user submitting the data.

The following keys are also recognized specifically by the *API*:

- *blog*, the fully qualified *URL* to the resource or application. If not specified, the *URL* provided to the constructor will be used.
- *referrer*, the content of the HTTP_REFERER header at the time of submission. (Note spelling; it does not follow the header name.)
- *permalink*, the permalink location, if any, of the entry the data was submitted to.
- *comment_type*, the type of data provided. Values specified in the *API* include 'comment', 'trackback', 'pingback', and an empty string (''), but it may be any value.
- *comment_author*, the name of the person submitting the data.
- *comment_author_email*, the email of the person submitting the data.
- *comment_author_url*, the *URL* or home page of the person submitting the data.
- *comment_content*, the actual data content submitted.

You may also submit any other environmental variables you feel might be a factor in determining if data is spam. Akismet suggests the contents of the entire $_SERVER array.

The `isSpam()` method will return either `TRUE` or `FALSE`, or throw an exception if the *API* key is invalid.

isSpam() Usage

```
$data = array(
    'user_ip'              => '111.222.111.222',
    'user_agent'           => 'Mozilla/5.0 ' . '(Windows; U; Windows NT ' .
                              '5.2; en-GB; rv:1.8.1) Gecko/20061010 ' .
                              'Firefox/2.0',
    'comment_type'         => 'contact',
    'comment_author'       => 'John Doe',
    'comment_author_email' => 'nospam@myhaus.net',
    'comment_content'      => "I'm not a spammer, honest!"
);
if ($akismet->isSpam($data)) {
    echo "Sorry, but we think you're a spammer.";
} else {
    echo "Welcome to our site!";
}
```

`isSpam()` implements the *comment-check* Akismet *API* method.

304.4 Submitting known spam

Spam data will occasionally get through the filter. If you discover spam that you feel should have been caught, you can submit it to Akismet to help improve their filter.

`ZendService\Akismet\Akismet::submitSpam()` takes the same data array as passed to `isSpam()`, but does not return a value. An exception will be raised if the *API* key used is invalid.

submitSpam() Usage

```
$data = array(
    'user_ip'               => '111.222.111.222',
    'user_agent'            => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
                               'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
    'comment_type'          => 'contact',
    'comment_author'        => 'John Doe',
    'comment_author_email'  => 'nospam@myhaus.net',
    'comment_content'       => "I'm not a spammer, honest!"
);
$akismet->submitSpam($data));
```

`submitSpam()` implements the *submit-spam* Akismet *API* method.

304.5 Submitting false positives (ham)

Data will occasionally be trapped erroneously as spam by Akismet. For this reason, you should probably keep a log of all data trapped as spam by Akismet and review it periodically. If you find such occurrences, you can submit the data to Akismet as "ham", or a false positive (ham is good, spam is not).

`ZendService\Akismet\Akismet::submitHam()` takes the same data array as passed to `isSpam()` or `submitSpam()`, and, like `submitSpam()`, does not return a value. An exception will be raised if the *API* key used is invalid.

submitHam() Usage

```
$data = array(
    'user_ip'               => '111.222.111.222',
    'user_agent'            => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
                               'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
    'comment_type'          => 'contact',
    'comment_author'        => 'John Doe',
    'comment_author_email'  => 'nospam@myhaus.net',
    'comment_content'       => "I'm not a spammer, honest!"
);
$akismet->submitHam($data));
```

`submitHam()` implements the *submit-ham* Akismet *API* method.

304.6 Zend-specific Methods

While the Akismet *API* only specifies four methods, `ZendService\Akismet\Akismet` has several additional methods that may be used for retrieving and modifying internal properties.

- `getBlogUrl()` and `setBlogUrl()` allow you to retrieve and modify the blog *URL* used in requests.
- `getApiKey()` and `setApiKey()` allow you to retrieve and modify the *API* key used in requests.
- `getCharset()` and `setCharset()` allow you to retrieve and modify the character set used to make the request.
- `getPort()` and `setPort()` allow you to retrieve and modify the *TCP* port used to make the request.

- `getUserAgent()` and `setUserAgent()` allow you to retrieve and modify the *HTTP* user agent used to make the request. Note: this is not the user_agent used in data submitted to the service, but rather the value provided in the *HTTP* User-Agent header when making a request to the service.

  The value used to set the user agent should be of the form *some user agent/version | Akismet/version*. The default is *Zend Framework/ZF-VERSION | Akismet/1.11*, where *ZF-VERSION* is the current Zend Framework version as stored in the `Zend\Version\Version::VERSION` constant.

CHAPTER

FIVE

ZENDSERVICEAMAZON

305.1 Introduction

`ZendService\Amazon` is a simple *API* for using Amazon web services. `ZendService\Amazon` has two *API*s: a more traditional one that follows Amazon's own *API*, and a simpler "Query *API*" for constructing even complex search queries easily.

`ZendService\Amazon` enables developers to retrieve information appearing throughout Amazon.com web sites directly through the Amazon Web Services *API*. Examples include:

- Store item information, such as images, descriptions, pricing, and more
- Customer and editorial reviews
- Similar products and accessories
- Amazon.com offers
- ListMania lists

In order to use `ZendService\Amazon`, you should already have an Amazon developer *API* key as well as a secret key. To get a key and for more information, please visit the Amazon Web Services web site. As of August 15th, 2009 you can only use the Amazon Product Advertising *API* through `ZendService\Amazon`, when specifying the additional secret key.

Note: Attention

Your Amazon developer *API* and secret keys are linked to your Amazon identity, so take appropriate measures to keep them private.

Search Amazon Using the Traditional API

In this example, we search for *PHP* books at Amazon and loop through the results, printing them.

```
$amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$results = $amazon->itemSearch(array('SearchIndex' => 'Books',
                                     'Keywords' => 'php'));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

Search Amazon Using the Query API

Here, we also search for *PHP* books at Amazon, but we instead use the Query *API*, which resembles the Fluent Interface design pattern.

```
$query = new ZendService\Amazon\Query('AMAZON_API_KEY',
                                      'US',
                                      'AMAZON_SECRET_KEY');
$query->category('Books')->Keywords('PHP');
$results = $query->search();
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

305.2 Country Codes

By default, `ZendService\Amazon` connects to the United States ("*US*") Amazon web service. To connect from a different country, simply specify the appropriate country code string as the second parameter to the constructor:

Choosing an Amazon Web Service Country

```
// Connect to Amazon in Japan
$amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'JP', 'AMAZON_SECRET_KEY');
```

Note: Country codes

Valid country codes are: *CA*, *DE*, *FR*, *JP*, *UK*, and *US*.

305.3 Looking up a Specific Amazon Item by ASIN

The `itemLookup()` method provides the ability to fetch a particular Amazon item when the *ASIN* is known.

Looking up a Specific Amazon Item by ASIN

```
$amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$item = $amazon->itemLookup('B0000A432X');
```

The `itemLookup()` method also accepts an optional second parameter for handling search options. For full details, including a list of available options, please see the relevant Amazon documentation.

Note: Image information

To retrieve images information for your search results, you must set *ResponseGroup* option to *Medium* or *Large*.

305.4 Performing Amazon Item Searches

Searching for items based on any of various available criteria are made simple using the `itemSearch()` method, as in the following example:

Performing Amazon Item Searches

```
$amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$results = $amazon->itemSearch(array('SearchIndex' => 'Books',
                                     'Keywords' => 'php'));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

Using the ResponseGroup Option

The *ResponseGroup* option is used to control the specific information that will be returned in the response.

```
$amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$results = $amazon->itemSearch(array(
    'SearchIndex'   => 'Books',
    'Keywords'      => 'php',
    'ResponseGroup' => 'Small,ItemAttributes,Images,SalesRank,Reviews,' .
                       'EditorialReview,Similarities,ListmaniaLists'
    ));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

The `itemSearch()` method accepts a single array parameter for handling search options. For full details, including a list of available options, please see the relevant Amazon documentation

Tip: The *ZendServiceAmazonQuery* class is an easy to use wrapper around this method.

305.5 Using the Alternative Query API

305.5.1 Introduction

`ZendService\Amazon\Query` provides an alternative *API* for using the Amazon Web Service. The alternative *API* uses the Fluent Interface pattern. That is, all calls can be made using chained method calls. (e.g., *$obj->method()->method2($arg)*)

The `ZendService\Amazon\Query` *API* uses overloading to easily set up an item search and then allows you to search based upon the criteria specified. Each of the options is provided as a method call, and each method's argument corresponds to the named option's value:

Search Amazon Using the Alternative Query API

In this example, the alternative query *API* is used as a fluent interface to specify options and their respective values:

```
$query = new ZendService\Amazon\Query('MY_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$query->Category('Books')->Keywords('PHP');
$results = $query->search();
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

This sets the option *Category* to "Books" and *Keywords* to "PHP".

For more information on the available options, please refer to the relevant Amazon documentation.

305.6 ZendServiceAmazon Classes

The following classes are all returned by *ZendServiceAmazon::itemLookup()* and *ZendServiceAmazon::itemSearch()*:

- *ZendServiceAmazonItem*
- *ZendServiceAmazonImage*
- *ZendServiceAmazonResultSet*
- *ZendServiceAmazonOfferSet*
- *ZendServiceAmazonOffer*
- *ZendServiceAmazonSimilarProduct*
- *ZendServiceAmazonAccessories*
- *ZendServiceAmazonCustomerReview*
- *ZendServiceAmazonEditorialReview*
- *ZendServiceAmazonListMania*

305.6.1 ZendServiceAmazonItem

`ZendService\Amazon\Item` is the class type used to represent an Amazon item returned by the web service. It encompasses all of the items attributes, including title, description, reviews, etc.

305.6.2 ZendServiceAmazonItem::asXML()

string:`asXML()`

Return the original *XML* for the item

305.6.3 Properties

`ZendService\Amazon\Item` has a number of properties directly related to their standard Amazon *API* counterparts. *Back to Class List*

305.6.4 ZendServiceAmazonImage

`ZendService\Amazon\Image` represents a remote Image for a product.

305.6.5 Properties

Table 305.1: ZendServiceAmazonImage Properties

Name	Type	Description
Url	ZendUriUri	Remote URL for the Image
Height	int	The Height of the image in pixels
Width	int	The Width of the image in pixels

Back to Class List

305.6.6 ZendServiceAmazonResultSet

`ZendService\Amazon\ResultSet` objects are returned by *ZendServiceAmazon::itemSearch()* and allow you to easily handle the multiple results returned.

Note: SeekableIterator

Implements the *SeekableIterator* for easy iteration (e.g. using *foreach*), as well as direct access to a specific result using `seek()`.

305.6.7 ZendServiceAmazonResultSet::totalResults()

int:`totalResults()` Returns the total number of results returned by the search

Back to Class List

305.6.8 ZendServiceAmazonOfferSet

Each result returned by *ZendServiceAmazon::itemSearch()* and *ZendServiceAmazon::itemLookup()* contains a `ZendService\Amazon\OfferSet` object through which pricing information for the item can be retrieved.

305.6.9 Properties

Back to Class List

305.6.10 ZendServiceAmazonOffer

Each offer for an item is returned as an `ZendService\Amazon\Offer` object.

305.6.11 ZendServiceAmazonOffer Properties

Table 305.2: Properties

Name	Type	Description
MerchantId	string	Merchants Amazon ID
MerchantName	string	Merchants Amazon Name. Requires setting the ResponseGroup option to OfferFull to retrieve.
GlancePage	string	URL for a page with a summary of the Merchant
Condition	string	Condition of the item
OfferListingId	string	ID of the Offer Listing
Price	int	Price for the item
CurrencyCode	string	Currency Code for the price of the item
Availability	string	Availability of the item
IsEligibleForSuperSaver-Shipping	boolean	Whether the item is eligible for Super Saver Shipping or not

Back to Class List

305.6.12 ZendServiceAmazonSimilarProduct

When searching for items, Amazon also returns a list of similar products that the searcher may find to their liking. Each of these is returned as a `ZendService\Amazon\SimilarProduct` object.

Each object contains the information to allow you to make sub-sequent requests to get the full information on the item.

305.6.13 Properties

Table 305.3: ZendServiceAmazonSimilarProduct Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

Back to Class List

305.6.14 ZendServiceAmazonAccessories

Accessories for the returned item are represented as `ZendService\Amazon\Accessories` objects

305.6.15 Properties

Table 305.4: ZendServiceAmazonAccessories Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

Back to Class List

305.6.16 ZendServiceAmazonCustomerReview

Each Customer Review is returned as a `ZendService\Amazon\CustomerReview` object.

305.6.17 Properties

Table 305.5: ZendServiceAmazonCustomerReview Properties

Name	Type	Description
Rating	string	Item Rating
HelpfulVotes	string	Votes on how helpful the review is
CustomerId	string	Customer ID
TotalVotes	string	Total Votes
Date	string	Date of the Review
Summary	string	Review Summary
Content	string	Review Content

Back to Class List

305.6.18 ZendServiceAmazonEditorialReview

Each items Editorial Reviews are returned as a `ZendService\Amazon\EditorialReview` object

305.6.19 Properties

Table 305.6: ZendServiceAmazonEditorialReview Properties

Name	Type	Description
Source	string	Source of the Editorial Review
Content	string	Review Content

Back to Class List

305.6.20 ZendServiceAmazonListmania

Each results List Mania List items are returned as `ZendService\Amazon\Listmania` objects.

305.6.21 Properties

Table 305.7: ZendServiceAmazonListmania Properties

Name	Type	Description
ListId	string	List ID
ListName	string	List Name

Back to Class List

CHAPTER

SIX

ZENDSERVICE\APPLE\APNS

306.1 Introduction

`ZendService\Apple\Apns` provides a client for the Apple Push Notification Service. `ZendService\Apple\Apns\Client` allows you to send data from servers to your iOS Applications.

In order to leverage APNS you **must** follow the Provisioning and deployments steps outlined by Apple.

The service is composed of 3 distinct parts:

- The Clients: * Feedback: `ZendService\Apple\Apns\Client\Feedback` * Message: `ZendService\Apple\Apns\Client\Message`
- The Message: `ZendService\Apple\Apns\Message\Alert`
- The Responses: * Feedback: `ZendService\Apple\Apns\Response\Feedback` * Message: `ZendService\Apple\Apns\Response\Message`

The Clients is the broker that sends the message to the APNS server and returns the response. The Message is where you define all of the message specific data that you would like to send for the alert. The Response is the feedback given back from the APNS server.

306.2 Quick Start

In order to send messages; you must have completed the provisioning and deployment steps mentioned above. Once you have your certificates in place you will be able to prepare to send messages to your iOS application. Here we will setup the client and prepare to send out messages.

```
use ZendService\Apple\Apns\Client\Message as Client;
use ZendService\Apple\Apns\Message;
use ZendService\Apple\Apns\Message\Alert;
use ZendService\Apple\Apns\Response\Message as Response;
use ZendService\Apple\Apns\Exception\RuntimeException;

$client = new Client();
$client->open(Client::SANDBOX_URI, '/path/to/push-certificate.pem', 'optionalPassPhrase');
```

So now that we have the client setup and available, it is time to define out the message that we intend to send to our iOS tokens that have registered for push notifications on our server. Note that many of the methods specified are not required but are here to give an inclusive look into the message.

```
$message = new Message();
$message->setId('my_unique_id');
$message->setToken('DEVICE_TOKEN');
$message->setBadge(5);
$message->setSound('bingbong.aiff');

// simple alert:
$message->setAlert('Bob wants to play poker');
// complex alert:
$alert = new Alert();
$alert->setBody('Bob wants to play poker');
$alert->setActionLocKey('PLAY');
$alert->setLocKey('GAME_PLAY_REQUEST_FORMAT');
$alert->setLocArgs(array('Jenna', 'Frank'));
$alert->setLaunchImage('Play.png');
$message->setAlert($alert);
```

Now that we have the message taken care of, all we need to do next is send out the message. Each message comes back with a set of data that allows us to understand what happened with our push notification as well as throwing exceptions in the cases of server failures.

```
try {
    $response = $client->send($message);
} catch (RuntimeException $e) {
    echo $e->getMessage() . PHP_EOL;
    exit(1);
}
$client->close();

if ($response->getCode() != Response::RESULT_OK) {
     switch ($response->getCode() {
         case Response::RESULT_PROCESSING_ERROR:
             // you may want to retry
             break;
         case Response::RESULT_MISSING_TOKEN:
             // you were missing a token
             break;
         case Response::RESULT_MISSING_TOPIC:
             // you are missing a message id
             break;
         case Response::RESULT_MISSING_PAYLOAD:
             // you need to send a payload
             break;
         case Response::RESULT_INVALID_TOKEN_SIZE:
             // the token provided was not of the proper size
             break;
         case Response::RESULT_INVALID_TOPIC_SIZE:
             // the topic was too long
             break;
         case Response::RESULT_INVALID_PAYLOAD_SIZE:
             // the payload was too large
             break;
         case Response::RESULT_INVALID_TOKEN:
             // the token was invalid; remove it from your system
             break;
         case Response::RESULT_UNKNOWN_ERROR:
             // apple didn't tell us what happened
             break;
```

```
        }
}
```

306.3 Feedback Service

APNS has a feedback service that you *must* listen to. Apple states that they monitor providers to ensure that they are listening to this service.

The feedback service simply returns an array of Feedback responses. All tokens provided in the feedback should not be sent to again; unless the device re-registers for push notification. You can use the time in the Feedback response to ensure that the device has not re-registered for push notifications since the last send.

```
use ZendService\Apple\Apns\Client\Message as Client;
use ZendService\Apple\Apns\Message;
use ZendService\Apple\Apns\Message\Alert;
use ZendService\Apple\Apns\Response\Message as Response;
use ZendService\Apple\Apns\Exception\RuntimeException;

$client = new Client();
$client->open(Client::SANDBOX_URI, '/path/to/push-certificate.pem', 'optionalPassPhrase');
$reponses = $client->feedback();
$client->close();

foreach ($responses as $response) {
    echo $response->getTime() . ': ' . $response->getToken();
}
```

CHAPTER

SEVEN

ZENDSERVICEAUDIOSCROBBLER

307.1 Introduction

`ZendService\Audioscrobbler` is a simple *API* for using the Audioscrobbler REST Web Service. The Audioscrobbler Web Service provides access to its database of Users, Artists, Albums, Tracks, Tags, Groups, and Forums. The methods of the `ZendService\Audioscrobbler` class begin with one of these terms. The syntax and namespaces of the Audioscrobbler Web Service are mirrored in `ZendService\Audioscrobbler`. For more information about the Audioscrobbler REST Web Service, please visit the Audioscrobbler Web Service site.

307.2 Users

In order to retrieve information for a specific user, the `setUser()` method is first used to select the user for which data are to be retrieved. `ZendService\Audioscrobbler` provides several methods for retrieving data specific to a single user:

- `userGetProfileInformation()`: Returns a SimpleXML object containing the current user's profile information.
- `userGetTopArtists()`: Returns a SimpleXML object containing a list of the current user's most listened to artists.
- `userGetTopAlbums()`: Returns a SimpleXML object containing a list of the current user's most listened to albums.
- `userGetTopTracks()`: Returns a SimpleXML object containing a list of the current user's most listened to tracks.
- `userGetTopTags()`: Returns a SimpleXML object containing a list of tags most applied by the current user.
- `userGetTopTagsForArtist()`: Requires that an artist be set via `setArtist()`. Returns a SimpleXML object containing the tags most applied to the current artist by the current user.
- `userGetTopTagsForAlbum()`: Requires that an album be set via `setAlbum()`. Returns a SimpleXML object containing the tags most applied to the current album by the current user.
- `userGetTopTagsForTrack()`: Requires that a track be set via `setTrack()`. Returns a SimpleXML object containing the tags most applied to the current track by the current user.
- `userGetFriends()`: Returns a SimpleXML object containing the user names of the current user's friends.
- `userGetNeighbours()`: Returns a SimpleXML object containing the user names of people with similar listening habits to the current user.

- `userGetRecentTracks()`: Returns a SimpleXML object containing the 10 tracks most recently played by the current user.
- `userGetRecentBannedTracks()`: Returns a SimpleXML object containing a list of the 10 tracks most recently banned by the current user.
- `userGetRecentLovedTracks()`: Returns a SimpleXML object containing a list of the 10 tracks most recently loved by the current user.
- `userGetRecentJournals()`: Returns a SimpleXML object containing a list of the current user's most recent journal entries.
- `userGetWeeklyChartList()`: Returns a SimpleXML object containing a list of weeks for which there exist Weekly Charts for the current user.
- `userGetRecentWeeklyArtistChart()`: Returns a SimpleXML object containing the most recent Weekly Artist Chart for the current user.
- `userGetRecentWeeklyAlbumChart()`: Returns a SimpleXML object containing the most recent Weekly Album Chart for the current user.
- `userGetRecentWeeklyTrackChart()`: Returns a SimpleXML object containing the most recent Weekly Track Chart for the current user.
- `userGetPreviousWeeklyArtistChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Artist Chart from `$fromDate` to `$toDate` for the current user.
- `userGetPreviousWeeklyAlbumChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Album Chart from `$fromDate` to `$toDate` for the current user.
- `userGetPreviousWeeklyTrackChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Track Chart from `$fromDate` to `$toDate` for the current user.

Retrieving User Profile Information

In this example, we use the `setUser()` and `userGetProfileInformation()` methods to retrieve a specific user's profile information:

```
$as = new ZendService\Audioscrobbler\Audioscrobbler();
// Set the user whose profile information we want to retrieve
$as->setUser('BigDaddy71');
// Retrieve BigDaddy71's profile information
$profileInfo = $as->userGetProfileInformation();
// Display some of it
print "Information for $profileInfo->realname "
    . "can be found at $profileInfo->url";
```

Retrieving a User's Weekly Artist Chart

```
$as = new ZendService\Audioscrobbler\Audioscrobbler();
// Set the user whose profile weekly artist chart we want to retrieve
$as->setUser('lo_fye');
// Retrieves a list of previous weeks for which there are chart data
$weeks = $as->userGetWeeklyChartList();
if (count($weeks) < 1) {
    echo 'No data available';
}
sort($weeks); // Order the list of weeks
```

```

$as->setFromDate($weeks[0]); // Set the starting date
$as->setToDate($weeks[0]); // Set the ending date

$previousWeeklyArtists = $as->userGetPreviousWeeklyArtistChart();

echo 'Artist Chart For Week Of '
   . date('Y-m-d h:i:s', $as->from_date)
   . '<br />';

foreach ($previousWeeklyArtists as $artist) {
    // Display the artists' names with links to their profiles
    print '<a href="' . $artist->url . '">' . $artist->name . '</a><br />';
}
```

307.3 Artists

`ZendService\Audioscrobbler\Audioscrobbler` provides several methods for retrieving data about a specific artist, specified via the `setArtist()` method:

- `artistGetRelatedArtists()`: Returns a SimpleXML object containing a list of Artists similar to the current Artist.
- `artistGetTopFans()`: Returns a SimpleXML object containing a list of Users who listen most to the current Artist.
- `artistGetTopTracks()`: Returns a SimpleXML object containing a list of the current Artist's top-rated Tracks.
- `artistGetTopAlbums()`: Returns a SimpleXML object containing a list of the current Artist's top-rated Albums.
- `artistGetTopTags()`: Returns a SimpleXML object containing a list of the Tags most frequently applied to current Artist.

Retrieving Related Artists

```
$as = new ZendService\Audioscrobbler\Audioscrobbler();
// Set the artist for whom you would like to retrieve related artists
$as->setArtist('LCD Soundsystem');
// Retrieve the related artists
$relatedArtists = $as->artistGetRelatedArtists();
foreach ($relatedArtists as $artist) {
    // Display the related artists
    print '<a href="' . $artist->url . '">' . $artist->name . '</a><br />';
}
```

307.4 Tracks

`ZendService\Audioscrobbler\Audioscrobbler` provides two methods for retrieving data specific to a single track, specified via the `setTrack()` method:

- `trackGetTopFans()`: Returns a SimpleXML object containing a list of Users who listen most to the current Track.

- `trackGetTopTags()`: Returns a SimpleXML object containing a list of the Tags most frequently applied to the current Track.

307.5 Tags

`ZendService\Audioscrobbler\Audioscrobbler` provides several methods for retrieving data specific to a single tag, specified via the `setTag()` method:

- `tagGetOverallTopTags()`: Returns a SimpleXML object containing a list of Tags most frequently used on Audioscrobbler.
- `tagGetTopArtists()`: Returns a SimpleXML object containing a list of Artists to whom the current Tag was most frequently applied.
- `tagGetTopAlbums()`: Returns a SimpleXML object containing a list of Albums to which the current Tag was most frequently applied.
- `tagGetTopTracks()`: Returns a SimpleXML object containing a list of Tracks to which the current Tag was most frequently applied.

307.6 Groups

`ZendService\Audioscrobbler\Audioscrobbler` provides several methods for retrieving data specific to a single group, specified via the `setGroup()` method:

- `groupGetRecentJournals()`: Returns a SimpleXML object containing a list of recent journal posts by Users in the current Group.
- `groupGetWeeklyChart()`: Returns a SimpleXML object containing a list of weeks for which there exist Weekly Charts for the current Group.
- `groupGetRecentWeeklyArtistChart()`: Returns a SimpleXML object containing the most recent Weekly Artist Chart for the current Group.
- `groupGetRecentWeeklyAlbumChart()`: Returns a SimpleXML object containing the most recent Weekly Album Chart for the current Group.
- `groupGetRecentWeeklyTrackChart()`: Returns a SimpleXML object containing the most recent Weekly Track Chart for the current Group.
- `groupGetPreviousWeeklyArtistChart($fromDate, $toDate)`: Requires `setFromDate()` and `setToDate()`. Returns a SimpleXML object containing the Weekly Artist Chart from the current fromDate to the current toDate for the current Group.
- `groupGetPreviousWeeklyAlbumChart($fromDate, $toDate)`: Requires `setFromDate()` and `setToDate()`. Returns a SimpleXML object containing the Weekly Album Chart from the current fromDate to the current toDate for the current Group.
- `groupGetPreviousWeeklyTrackChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Track Chart from the current fromDate to the current toDate for the current Group.

307.7 Forums

`ZendService\Audioscrobbler\Audioscrobbler` provides a method for retrieving data specific to a single forum, specified via the `setForum()` method:

- `forumGetRecentPosts()`: Returns a SimpleXML object containing a list of recent posts in the current forum.

CHAPTER

EIGHT

ZENDSERVICEDELICIOUS

308.1 Introduction

`ZendService\Delicious` is simple *API* for using del.icio.us *XML* and *JSON* web services. This component gives you read-write access to posts at del.icio.us if you provide credentials. It also allows read-only access to public data of all users.

Get all posts

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');
$posts = $delicious->getAllPosts();

foreach ($posts as $post) {
    echo "--\n";
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

308.2 Retrieving posts

`ZendService\Delicious\Delicious` provides three methods for retrieving posts: `getPosts()`, `getRecentPosts()` and `getAllPosts()`. All of these methods return an instance of `ZendService\Delicious\PostList`, which holds all retrieved posts.

```
/**
 * Get posts matching the arguments. If no date or url is given,
 * most recent date will be used.
 *
 * @param string $tag Optional filtering by tag
 * @param DateTime $dt Optional filtering by date
 * @param string $url Optional filtering by url
 * @return ZendService\Delicious\PostList
 */
public function getPosts($tag = null, $dt = null, $url = null);

/**
 * Get recent posts
 *
 * @param string $tag   Optional filtering by tag
```

```
 * @param string $count Maximal number of posts to be returned
 *                      (default 15)
 * @return ZendService\Delicious\PostList
 */
public function getRecentPosts($tag = null, $count = 15);

/**
 * Get all posts
 *
 * @param string $tag Optional filtering by tag
 * @return ZendService\Delicious\PostList
 */
public function getAllPosts($tag = null);
```

308.3 ZendServiceDeliciousPostList

Instances of this class are returned by the `getPosts()`, `getAllPosts()`, `getRecentPosts()`, and `getUserPosts()` methods of `ZendService\Delicious`.

For easier data access this class implements the *Countable*, *Iterator*, and *ArrayAccess* interfaces.

Accessing post lists

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// count posts
echo count($posts);

// iterate over posts
foreach ($posts as $post) {
    echo "--\n";
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}

// get post using array access
echo $posts[0]->getTitle();
```

Note: The `ArrayAccess::offsetSet()` and `ArrayAccess::offsetUnset()` methods throw exceptions in this implementation. Thus, code like *unset($posts[0]);* and *$posts[0] = 'A';* will throw exceptions because these properties are read-only.

Post list objects have two built-in filtering capabilities. Post lists may be filtered by tags and by *URL*.

Filtering a Post List with Specific Tags

Posts may be filtered by specific tags using `withTags()`. As a convenience, `withTag()` is also provided for when only a single tag needs to be specified.

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// Print posts having "php" and "zend" tags
foreach ($posts->withTags(array('php', 'zend')) as $post) {
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

Filtering a Post List by URL

Posts may be filtered by *URL* matching a specified regular expression using the `withUrl()` method:

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// Print posts having "help" in the URL
foreach ($posts->withUrl('/help/') as $post) {
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

308.4 Editing posts

Post editing

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');
$posts = $delicious->getPosts();

// set title
$posts[0]->setTitle('New title');
// save changes
$posts[0]->save();
```

Method call chaining

Every setter method returns the post object so that you can chain method calls using a fluent interface.

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');
$posts = $delicious->getPosts();

$posts[0]->setTitle('New title')
         ->setNotes('New notes')
         ->save();
```

308.5 Deleting posts

There are two ways to delete a post, by specifying the post *URL* or by calling the `delete()` method upon a post object.

Deleting posts

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');

// by specifying URL
$delicious->deletePost('http://framework.zend.com');

// or by calling the method upon a post object
$posts = $delicious->getPosts();
$posts[0]->delete();

// another way of using deletePost()
$delicious->deletePost($posts[0]->getUrl());
```

308.6 Adding new posts

To add a post you first need to call the `createNewPost()` method, which returns a `ZendService\Delicious\Post` object. When you edit the post, you need to save it to the del.icio.us database by calling the `save()` method.

Adding a post

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');

// create a new post and save it (with method call chaining)
$delicious->createNewPost('Zend Framework', 'http://framework.zend.com')
          ->setNotes('Zend Framework Homepage')
          ->save();

// create a new post and save it  (without method call chaining)
$newPost = $delicious->createNewPost('Zend Framework',
                                     'http://framework.zend.com');
$newPost->setNotes('Zend Framework Homepage');
$newPost->save();
```

308.7 Tags

Tags

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');

// get all tags
print_r($delicious->getTags());

// rename tag ZF to zendFramework
$delicious->renameTag('ZF', 'zendFramework');
```

308.8 Bundles

Bundles

```
$delicious = new ZendService\Delicious\Delicious('username', 'password');

// get all bundles
print_r($delicious->getBundles());

// delete bundle someBundle
$delicious->deleteBundle('someBundle');

// add bundle
$delicious->addBundle('newBundle', array('tag1', 'tag2'));
```

308.9 Public data

The del.icio.us web *API* allows access to the public data of all users.

Note: When using only these methods, a username and password combination is not required when constructing a new `ZendService\Delicious` object.

Retrieving public data

```
// username and password are not required
$delicious = new ZendService\Delicious\Delicious();

// get fans of user someUser
print_r($delicious->getUserFans('someUser'));

// get network of user someUser
print_r($delicious->getUserNetwork('someUser'));

// get tags of user someUser
print_r($delicious->getUserTags('someUser'));
```

308.9.1 Public posts

When retrieving public posts with the `getUserPosts()` method, a `ZendService\Delicious\PostList` object is returned, and it contains `ZendService\Delicious\SimplePost` objects, which contain basic information about the posts, including *URL*, title, notes, and tags.

Table 308.1: Methods of the ZendServiceDeliciousSimplePost class

Name	Description	Return type
getNotes()	Returns notes of a post	String
getTags()	Returns tags of a post	Array
getTitle()	Returns title of a post	String
getUrl()	Returns URL of a post	String

308.10 HTTP client

`ZendService\Delicious` uses `Zend\Rest\Client` for making *HTTP* requests to the del.icio.us web service. To change which *HTTP* client `ZendService\Delicious` uses, you need to change the *HTTP* client of `Zend\Rest\Client`.

Changing the HTTP client of ZendRestClient

```
$myHttpClient = new My_Http_Client();
Zend\Rest\Client::setHttpClient($myHttpClient);
```

When you are making more than one request with `ZendService\Delicious` to speed your requests, it's better to configure your *HTTP* client to keep connections alive.

Configuring your HTTP client to keep connections alive

```
Zend\Rest\Client::getHttpClient()->setConfig(array(
        'keepalive' => true
));
```

Note: When a `ZendService\Delicious` object is constructed, the *SSL* transport of `Zend\Rest\Client` is set to *'ssl'* rather than the default of *'ssl2'*. This is because del.icio.us has some problems with *'ssl2'*, such as requests taking a long time to complete (around 2 seconds).

CHAPTER

NINE

ZEND_SERVICE_DEVELOPERGARDEN

309.1 Introduction to DeveloperGarden

Developer Garden is the name of Deutsche Telekom's developer community. Developer Garden offers you access to core services of Deutsche Telekom, such as voice connections (Voice Call) or sending text messages (Send SMS) via open interfaces (Open *API*s). You can access the Developer Garden services directly via *SOAP* or *REST*.

The family of `Zend_Service_DeveloperGarden` components provides a clean and simple interface to the Developer Garden APIs and additionally offers functionality to improve handling and performance.

- *BaseUserService*: Class to manage *API* quota and user accounting details.
- *IPLocation*: Locale the given IP and returns geo coordinates. Works only with IPs allocated in the network of the Deutsche Telekom.
- *LocalSearch*: Allows you to search with options nearby or around a given geo coordinate or city.
- *SendSMS*: Send a SMS or Flash SMS to a given number.
- *SMSValidation*: You can validate a number to use it with SendSMS for also supply a back channel.
- *VoiceCall*: Initiates a call between two participants.
- *ConferenceCall*: You can configure a whole conference room with participants for an adhoc conference or you can also schedule your conference.

The backend *SOAP API* is documented here.

309.1.1 Sign Up for an Account

Before you can start using the DeveloperGarden *API*, you first have to sign up for an account.

309.1.2 The Environment

With the DeveloperGarden *API* you have the possibility to choose between 3 different development environments.

- **production**: In Production environment there are no usage limitations. You have to pay for calls, sms and other services with costs.
- **sandbox**: In the Sandbox mode you can use the same features (with limitations) as in the production without to paying for them. This environment is suitable for testing your prototype.
- **mock**: The Mock environment allows you to build your application and have results but you do not initiate any action on the *API* side. This environment is intended for testing during development.

For every environment and service, there are some special features (options) available for testing. Please look here for details.

309.1.3 Your configuration

You can pass to all classes an array of configuration values. Possible values are:

- **username**: Your DeveloperGarden *API* username.
- **password**: Your DeveloperGarden *API* password.
- **environment**: The environment that you selected.

Configuration Example

```
require_once 'Zend/Service/DeveloperGarden/SendSms.php';
$config = array(
    'username'    => 'yourUsername',
    'password'    => 'yourPassword',
    'environment' => Zend_Service_DeveloperGarden_SendSms::ENV_PRODUCTION,
);
$service = new Zend_Service_DeveloperGarden_SendSms($config);
```

309.2 BaseUserService

The class can be used to set and get quota values for the services and to fetch account details.

The `getAccountBalance()` method fetches an array of account id's with the current balance status (credits).

Get account balance example

```
$service = new Zend_Service_DeveloperGarden_BaseUserService($config);
print_r($service->getAccountBalance());
```

309.2.1 Get quota information

You can fetch quota informations for a specific service module with the provided methods.

Get quota information example

```
$service = new Zend_Service_DeveloperGarden_BaseUserService($config);
$result = $service->getSmsQuotaInformation(
    Zend_Service_DeveloperGarden_BaseUserService::ENV_PRODUCTION
);
echo 'Sms Quota:<br />';
echo 'Max Quota: ', $result->getMaxQuota(), '<br />';
echo 'Max User Quota: ', $result->getMaxUserQuota(), '<br />';
echo 'Quota Level: ', $result->getQuotaLevel(), '<br />';
```

You get a `result` object that contains all the information you need, optional you can pass to the `QuotaInformation` method the environment constant to fetch the quota for the specific environment.

Here a list of all `getQuotaInformation` methods:

- `getConferenceCallQuotaInformation()`
- `getIPLocationQuotaInformation()`
- `getLocalSearchQuotaInformation()`
- `getSmsQuotaInformation()`
- `getVoiceCallQuotaInformation()`

309.2.2 Change quota information

To change the current quota use one of the `changeQuotaPool` methods. First parameter is the new pool value and the second one is the environment.

Change quota information example

```
$service = new Zend_Service_DeveloperGarden_BaseUserService($config);
$result = $service->changeSmsQuotaPool(
    1000,
    Zend_Service_DeveloperGarden_BaseUserService::ENV_PRODUCTION
);
if (!$result->hasError()) {
    echo 'updated Quota Pool';
}
```

Here a list of all `changeQuotaPool` methods:

- `changeConferenceCallQuotaPool()`
- `changeIPLocationQuotaPool()`
- `changeLocalSearchQuotaPool()`
- `changeSmsQuotaPool()`
- `changeVoiceCallQuotaPool()`

309.3 IP Location

This service allows you to retrieve location information for a given IP address.

There are some limitations:

- The IP address must be in the T-Home network
- Just the next big city will be resolved
- IPv6 is not supported yet

Locate a given IP

```
$service = new Zend_Service_DeveloperGarden_IpLocation($config);
$service->setEnvironment(
    Zend_Service_DeveloperGarden_IpLocation::ENV_MOCK
);
$ip = new Zend_Service_DeveloperGarden_IpLocation_IpAddress('127.0.0.1');
print_r($service->locateIp($ip));
```

309.4 Local Search

The Local Search service provides the location based search machine suchen.de via web service interface. For more details, refer to the documentation.

Locate a Restaurant

```
$service = new Zend_Service_DeveloperGarden_LocalSearch($config);
$search  = new Zend_Service_DeveloperGarden_LocalSearch_SearchParameters();
/**
 * @see http://www.developergarden.com/static/docu/en/ch04s02s06s04.html
 */
$search->setWhat('pizza')
       ->setWhere('jena');
print_r($service->localSearch($search));
```

309.5 Send SMS

The Send SMS service is used to send normal and Flash SMS to any number.

The following restrictions apply to the use of the SMS service:

- An SMS or Flash SMS in the production environment must not be longer than 765 characters and must not be sent to more than 10 recipients.
- An SMS or Flash SMS in the sandbox environment is shortened and enhanced by a note from the DeveloperGarden. The maximum length of the message is 160 characters.
- In the sandbox environment, a maximum of 10 SMS can be sent per day.
- The following characters are counted twice: `|` `^` `€` `{` `}` `[` `]` `~` `\` `LF` (line break)
- If a SMS or Flash SMS is longer than 160 characters, one message is charged for each 153 characters (quota and credit).
- Delivery cannot be guaranteed for SMS or Flash SMS to landline numbers.
- The sender can be a maximum of 11 characters. Permitted characters are letters and numbers.
- The specification of a phone number as the sender is only permitted if the phone number has been validated. (See: *SMS Validation*)

Sending an SMS

```
$service = new Zend_Service_DeveloperGarden_SendSms($config);
$sms = $service->createSms(
    '+49-172-123456; +49-177-789012',
    'your test message',
    'yourname'
);
print_r($service->send($sms));
```

309.6 SMS Validation

The SMS Validation service allows the validation of physical phone number to be used as the sender of an SMS.

First, call `setValidationKeyword()` to receive an SMS with a keyword.

After you get your keyword, you have to use the `validate()` to validate your number with the keyword against the service.

With the method `getValidatedNumbers()`, you will get a list of all already validated numbers and the status of each.

Request validation keyword

```
$service = new Zend_Service_DeveloperGarden_SmsValidation($config);
print_r($service->sendValidationKeyword('+49-172-123456'));
```

Validate a number with a keyword

```
$service = new Zend_Service_DeveloperGarden_SmsValidation($config);
print_r($service->validate('TheKeyWord', '+49-172-123456'));
```

To invalidate a validated number, call the method `inValidate()`.

309.7 Voice Call

The Voice Call service can be used to set up a voice connection between two telephone connections. For specific details please read the API Documentation.

Normally the Service works as followed:

- Call the first participant.
- If the connection is successful, call the second participant.
- If second participant connects successfully, both participants are connected.
- The call is open until one of the participants hangs up or the expire mechanism intercepts.

Call two numbers

```
$service = new Zend_Service_DeveloperGarden_VoiceCall($config);
$aNumber = '+49-30-000001';
$bNumber = '+49-30-000002';
$expiration  = 30;  // seconds
$maxDuration = 300; // 5 mins
$newCall = $service->newCall($aNumber, $bNumber, $expiration, $maxDuration);
echo $newCall->getSessionId();
```

If the call is initiated, you can ask the result object for the session ID and use this session ID for an additional call to the `callStatus` or `tearDownCall()` methods. The second parameter on the `callStatus()` method call extends the expiration for this call.

Call two numbers, ask for status, and cancel

```
$service = new Zend_Service_DeveloperGarden_VoiceCall($config);
$aNumber = '+49-30-000001';
$bNumber = '+49-30-000002';
$expiration  = 30; // seconds
$maxDuration = 300; // 5 mins

$newCall = $service->newCall($aNumber, $bNumber, $expiration, $maxDuration);

$sessionId = $newCall->getSessionId();

$service->callStatus($sessionId, true); // extend the call

sleep(10); // sleep 10s and then tearDown

$service->tearDownCall($sessionId);
```

309.8 ConferenceCall

Conference Call allows you to setup and start a phone conference.

The following features are available:

- Conferences with an immediate start
- Conferences with a defined start date
- Recurring conference series
- Adding, removing, and muting of participants from a conference
- Templates for conferences

Here is a list of currently implemented *API* methods:

- `createConference()` creates a new conference
- `updateConference()` updates an existing conference
- `commitConference()` saves the conference, and, if no date is configured, immediately starts the conference
- `removeConference()` removes a conference

- `getConferenceList()` returns a list of all configured conferences
- `getConferenceStatus()` displays information for an existing conference
- `getParticipantStatus()` displays status information about a conference participant
- `newParticipant()` creates a new participant
- `addParticipant()` adds a participant to a conference
- `updateParticipant()` updates a participant, usually to mute or redial the participant
- `removeParticipant()` removes a participant from a conference
- `getRunningConference()` requests the running instance of a planned conference
- `createConferenceTemplate()` creates a new conference template
- `getConferenceTemplate()` requests an existing conference template
- `updateConferenceTemplate()` updates existing conference template details
- `removeConferenceTemplate()` removes a conference template
- `getConferenceTemplateList()` requests all conference templates of an owner
- `addConferenceTemplateParticipant()` adds a conference participant to conference template
- `getConferenceTemplateParticipant()` displays details of a participant of a conference template
- `updateConferenceTemplateParticipant()` updates participant details within a conference template
- `removeConferenceTemplateParticipant()` removes a participant from a conference template

Ad-Hoc conference

```
$client = new Zend_Service_DeveloperGarden_ConferenceCall($config);

$conferenceDetails =
    new Zend_Service_DeveloperGarden_ConferenceCall_ConferenceDetail(
        'Zend-Conference',                    // name for the conference
        'this is my private zend conference', // description
        60                                    // duration in seconds
    );

$conference = $client->createConference('MyName', $conferenceDetails);

$part1 = new Zend_Service_DeveloperGarden_ConferenceCall_ParticipantDetail(
    'Jon',
    'Doe',
    '+49-123-4321',
    'your.name@example.com',
    true
);

$client->newParticipant($conference->getConferenceId(), $part1);
// add a second, third ... participant

$client->commitConference($conference->getConferenceId());
```

309.9 Performance and Caching

You can setup various caching options to improve the performance for resolving WSDL and authentication tokens.

First of all, you can setup the internal SoapClient (PHP) caching values.

WSDL cache options

```
Zend_Service_DeveloperGarden_SecurityTokenServer_Cache::setWsdlCache(
    [PHP CONSTANT]
);
```

The `[PHP CONSTANT]` can be one of the following values:

- `WSDL_CACHE_DISC`: enabled disc caching
- `WSDL_CACHE_MEMORY`: enabled memory caching
- `WSDL_CACHE_BOTH`: enabled disc and memory caching
- `WSDL_CACHE_NONE`: disabled both caching

If you also want to cache the result for calls to the SecurityTokenServer you can setup a `Zend_Cache` instance and pass it to the `setCache()`.

SecurityTokenServer cache option

```
$cache = Zend_Cache::factory('Core', ...);
Zend_Service_DeveloperGarden_SecurityTokenServer_Cache::setCache($cache);
```

CHAPTER

ZENDSERVICEFLICKR

310.1 Introduction

`ZendService\Flickr` is a simple *API* for using the Flickr REST Web Service. In order to use the Flickr web services, you must have an *API* key. To obtain a key and for more information about the Flickr REST Web Service, please visit the Flickr API Documentation.

In the following example, we use the `tagSearch()` method to search for photos having "php" in the tags.

Simple Flickr Photo Search

```
$flickr = new ZendService\Flickr\Flickr('MY_API_KEY');

$results = $flickr->tagSearch("php");

foreach ($results as $result) {
    echo $result->title . '<br />';
}
```

Note: Optional parameter

`tagSearch()` accepts an optional second parameter as an array of options.

310.2 Finding Flickr Users' Photos and Information

`ZendService\Flickr\Flickr` provides several ways to get information about Flickr users:

- `userSearch()`: Accepts a string query of space-delimited tags and an optional second parameter as an array of search options, and returns a set of photos as a `ZendService\Flickr\ResultSet` object.
- `getIdByUsername()`: Returns a string user ID associated with the given username string.
- `getIdByEmail()`: Returns a string user ID associated with the given email address string.

Finding a Flickr User's Public Photos by E-Mail Address

In this example, we have a Flickr user's e-mail address, and we search for the user's public photos by using the `userSearch()` method:

```
$flickr = new ZendService\Flickr('MY_API_KEY');

$results = $flickr->userSearch($userEmail);

foreach ($results as $result) {
    echo $result->title . '<br />';
}
```

310.3 Finding photos From a Group Pool

`ZendService\Flickr\Flickr` allows to retrieve a group's pool photos based on the group ID. Use the `groupPoolGetPhotos()` method:

Retrieving a Group's Pool Photos by Group ID

```
$flickr = new ZendService\Flickr\Flickr('MY_API_KEY');

    $results = $flickr->groupPoolGetPhotos($groupId);

    foreach ($results as $result) {
        echo $result->title . '<br />';
    }
```

Note: Optional parameter

`groupPoolGetPhotos()` accepts an optional second parameter as an array of options.

310.4 Retrieving Flickr Image Details

`ZendService\Flickr\Flickr` makes it quick and easy to get an image's details based on a given image ID. Just use the `getImageDetails()` method, as in the following example:

Retrieving Flickr Image Details

Once you have a Flickr image ID, it is a simple matter to fetch information about the image:

```
$flickr = new ZendService\Flickr\Flickr('MY_API_KEY');

$image = $flickr->getImageDetails($imageId);

echo "Image ID $imageId is $image->width x $image->height pixels.<br />\n";
echo "<a href=\"$image->clickUri\">Click for Image</a>\n";
```

310.5 ZendServiceFlickr Result Classes

The following classes are all returned by `tagSearch()` and `userSearch()`:

- *ZendServiceFlickrResultSet*
- *ZendServiceFlickrResult*
- *ZendServiceFlickrImage*

310.5.1 ZendServiceFlickrResultSet

Represents a set of Results from a Flickr search.

Note: Implements the `SeekableIterator` interface for easy iteration (e.g., using `foreach()`), as well as direct access to a specific result using `seek()`.

310.5.2 Properties

Table 310.1: ZendServiceFlickrResultSet Properties

Name	Type	Description
totalResultsAvailable	int	Total Number of Results available
totalResultsReturned	int	Total Number of Results returned
firstResultPosition	int	The offset in the total result set of this result set

310.5.3 ZendServiceFlickrResultSet::totalResults()

int:`totalResults()`

Returns the total number of results in this result set.

Back to Class List

310.5.4 ZendServiceFlickrResult

A single Image result from a Flickr query

310.5.5 Properties

Back to Class List

310.5.6 ZendServiceFlickrImage

Represents an Image returned by a Flickr search.

310.5.7 Properties

Table 310.2: ZendServiceFlickrImage Properties

Name	Type	Description
uri	string	URI for the original image
clickUri	string	Clickable URI (i.e. the Flickr page) for the image
width	int	Width of the Image
height	int	Height of the Image

Back to Class List

CHAPTER

ONE

ZENDSERVICE\LIVEDOCX

311.1 Introduction to LiveDocx

LiveDocx is a *SOAP* service that allows developers to generate word processing documents by combining structured textual or image data from *PHP* with a template, created in a word processor. The resulting document can be saved as a *PDF*, *DOCX*, *DOC*, *HTML* or *RTF* file. LiveDocx implements mail-merge in *PHP*.

The family of `ZendService\LiveDocx` components provides a clean and simple interface to *LiveDocx Free*, *LiveDocx Premium* and *LiveDocx Fully Licensed*, authored by *Text Control GmbH*, and additionally offers functionality to improve network performance.

`ZendService\LiveDocx` is part of the official Zend Framework family, but has to be downloaded and installed in addition to the core components of the Zend Framework, as do all other service components. Please refer to GitHub (ZendServiceLiveDocx) for download and installation instructions.

In addition to this section of the manual, to learn more about `ZendService\LiveDocx` and the backend *SOAP* service LiveDocx, please take a look at the following resources:

- **Shipped demonstration applications**. There is a large number of demonstration applications in the directory `/demos`. They illustrate all functionality offered by LiveDocx. Where appropriate this part of the user manual references the demonstration applications at the end of each section. It is **highly recommended** to read all the code in the `/demos` directory. It is well commented and explains all you need to know about LiveDocx and `ZendService\LiveDocx`.
- LiveDocx in PHP.
- LiveDocx SOAP API documentation.
- LiveDocx WSDL.
- LiveDocx blog and web site.

311.1.1 Sign Up for an Account

Before you can start using LiveDocx, you must first sign up for an account. The account is completely free of charge and you only need to specify a **username**, **password** and **e-mail address**. Your login credentials will be dispatched to the e-mail address you supply, so please type carefully. If, or when, your application gets really popular and you require high performance, or additional features only supplied in the premium service, you can upgrade from the *LiveDocx Free* to *LiveDocx Premium* for a minimal monthly charge. For details of the various services, please refer to LiveDocx pricing.

311.1.2 Templates and Documents

LiveDocx differentiates between the following terms: 1) **template** and 2) **document**. In order to fully understand the documentation and indeed LiveDocx itself, it is important that any programmer deploying LiveDocx understands the difference.

The term **template** is used to refer to the input file, created in a word processor, containing formatting and text fields. You can download an example template, stored as a *DOCX* file. The term **document** is used to refer to the output file that contains the template file, populated with data - i.e. the finished document. You can download an example document, stored as a *PDF* file.

311.1.3 Supported File Formats

LiveDocx supports the following file formats:

311.1.4 Template File Formats (input)

Templates can be saved in any of the following file formats:

- DOCX- Office Open *XML* format
- DOC- Microsoft Word *DOC* format
- RTF- Rich text file format
- TXD- TX Text Control format

311.1.5 Document File Formats (output):

The resulting document can be saved in any of the following file formats:

- DOCX- Office Open *XML* format
- DOC- Microsoft Word *DOC* format
- HTML-*XHTML* 1.0 transitional format
- RTF- Rich text file format
- PDF- Acrobat Portable Document Format
- PDF/A- Acrobat Portable Document Format (ISO-standardized version)
- TXD- TX Text Control format
- TXT-*ANSI* plain text

311.1.6 Image File Formats (output):

The resulting document can be saved in any of the following graphical file formats:

- BMP- Bitmap image format
- GIF- Graphics Interchange Format
- JPG- Joint Photographic Experts Group format
- PNG- Portable Network Graphics format

- TIFF- Tagged Image File Format
- WMF- Windows Meta File format

311.2 ZendService\LiveDocx\MailMerge

`MailMerge` is the mail-merge object in the `ZendService\LiveDocx` family.

311.2.1 Document Generation Process

The document generation process can be simplified with the following equation:

Template + Data = Document

Or expressed by the following diagram:

Data is inserted into template to create a document.

A template, created in a word processing application, such as Microsoft Word, is loaded into LiveDocx. Data is then inserted into the template and the resulting document is saved to any supported format.

311.2.2 Creating Templates in Microsoft Word 2007

Start off by launching Microsoft Word and creating a new document. Next, open up the **Field** dialog box. This looks as follows:

Microsoft Word 2007 Field dialog box.

Using this dialog, you can insert the required merge fields into your document. Below is a screenshot of a license agreement in Microsoft Word 2007. The merge fields are marked as `{ MERGEFIELD FieldName }`:

Template in Microsoft Word 2007.

Now, save the template as **template.docx**.

In the next step, we are going to populate the merge fields with textual data from *PHP*.

Cropped template in Microsoft Word 2007.

To populate the merge fields in the above cropped screenshot of the template in Microsoft Word, all we have to code is as follows:

```
use ZendService\LiveDocx\MailMerge;

$locale    = Locale::getDefault();
$timestamp = time();

$intlTimeFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::NONE, IntlDateFormatter::SHORT);

$intlDateFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_
```

```
17
18  $mailMerge->setLocalTemplate('license-agreement-template.docx');
19
20  $mailMerge->assign('software', 'Magic Graphical Compression Suite v1.9')
21            ->assign('licensee', 'Henry Döner-Meyer')
22            ->assign('company',  'Co-Operation')
23            ->assign('date',     $intlDateFormatter->format($timestamp))
24            ->assign('time',     $intlTimeFormatter->format($timestamp))
25            ->assign('city',     'Lyon')
26            ->assign('country',  'France');
27
28  $mailMerge->createDocument();
29
30  $document = $mailMerge->retrieveDocument('pdf');
31
32  file_put_contents('license-agreement-document.pdf', $document);
33
34  unset($mailMerge);
```

The resulting document is written to disk in the file **license-agreement-document.pdf**. This file can now be post-processed, sent via e-mail or simply displayed, as is illustrated below in **Document Viewer 2.26.1** on **Ubuntu 9.04**:

Resulting document as *PDF* in Document Viewer 2.26.1. For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/license-agreement`.

311.2.3 Advanced Mail-Merge

`ZendService\LiveDocx\MailMerge` allows designers to insert any number of text fields into a template. These text fields are populated with data when **createDocument()** is called.

In addition to text fields, it is also possible specify regions of a document, which should be repeated.

For example, in a telephone bill it is necessary to print out a list of all connections, including the destination number, duration and cost of each call. This repeating row functionality can be achieved with so called blocks.

Blocks are simply regions of a document, which are repeated when `createDocument()` is called. In a block any number of **block fields** can be specified.

Blocks consist of two consecutive document targets with a unique name. The following screenshot illustrates these targets and their names in red:

The format of a block is as follows:

```
blockStart_ + unique name
blockEnd_ + unique name
```

For example:

```
blockStart_block1
blockEnd_block1
```

The content of a block is repeated, until all data assigned in the block fields has been injected into the template. The data for block fields is specified in *PHP* as a multi-assoc array.

The following screenshot of a template in Microsoft Word 2007 shows how block fields are used:

Template, illustrating blocks in Microsoft Word 2007.

The following code populates the above template with data.

```
use ZendService\LiveDocx\MailMerge;

$locale    = Locale::getDefault();
$timestamp = time();

$intlDateFormatter1 = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$intlDateFormatter2 = new IntlDateFormatter($locale,
        null, null, null, null, 'LLLL yyyy');

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_P

$mailMerge->setLocalTemplate('telephone-bill-template.doc');

$mailMerge->assign('customer_number', sprintf("#%'10s", rand(0,1000000000)))
          ->assign('invoice_number',  sprintf("#%'10s", rand(0,1000000000)))
          ->assign('account_number',  sprintf("#%'10s", rand(0,1000000000)));

$billData = array (
    'phone'         => '+22 (0)333 444 555',
    'date'          => $intlDateFormatter1->format($timestamp),
    'name'          => 'James Henry Brown',
    'service_phone' => '+22 (0)333 444 559',
    'service_fax'   => '+22 (0)333 444 558',
    'month'         => $intlDateFormatter2->format($timestamp),
    'monthly_fee'   => '15.00',
    'total_net'     => '19.60',
    'tax'           => '19.00',
    'tax_value'     =>  '3.72',
    'total'         => '23.32'
);

$mailMerge->assign($billData);

$billConnections = array(
    array(
        'connection_number'   => '+11 (0)222 333 441',
        'connection_duration' => '00:01:01',
        'fee'                 => '1.15'
    ),
    array(
        'connection_number'   => '+11 (0)222 333 442',
        'connection_duration' => '00:01:02',
        'fee'                 => '1.15'
    ),
    array(
        'connection_number'   => '+11 (0)222 333 443',
        'connection_duration' => '00:01:03',
        'fee'                 => '1.15'
    ),
    array(
        'connection_number'   => '+11 (0)222 333 444',
        'connection_duration' => '00:01:04',
```

```
        'fee'                   => '1.15'
    )
);

$mailMerge->assign('connection', $billConnections);

$mailMerge->createDocument();

$document = $mailMerge->retrieveDocument('pdf');

file_put_contents('telephone-bill-document.pdf', $document);

unset($mailMerge);
```

The data, which is specified in the array `$billConnections` is repeated in the template in the block connection. The keys of the array (`connection_number`, `connection_duration` and `fee`) are the block field names - their data is inserted, one row per iteration.

The resulting document is written to disk in the file **telephone-bill-document.pdf**. This file can now be post-processed, sent via e-mail or simply displayed, as is illustrated below in **Document Viewer 2.26.1** on **Ubuntu 9.04**:

Resulting document as *PDF* in Document Viewer 2.26.1.

You can download the *DOC* template file and the resulting PDF document.

NOTE: blocks may not be nested.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/telephone-bill`.

311.2.4 Merging Image Data into a Template

In addition to assigning textual data, it is also possible to merge image data into a template. The following code populates a conference badge template with the photo `dailemaitre.jpg`, in addition to some textual data.

The first step is to upload the image to the backend service. Once you have done this, you can assign the filename of the image to the template just as you would any other textual data. Note the syntax of the field name containing an image - it must start with `image`:

```
use ZendService\LiveDocx\MailMerge;

$locale    = Locale::getDefault();
$timestamp = time();

$intlDateFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

$photoFilename = __DIR__ . '/dailemaitre.jpg';
$photoFile     = basename($photoFilename);

if (!$mailMerge->imageExists($photoFile)) {          // pass image file *without* path
    $mailMerge->uploadImage($photoFilename);         // pass image file *with* path
}
```

```

$mailMerge->setLocalTemplate('conference-pass-template.docx');

$mailMerge->assign('name',        'Daï Lemaitre')
          ->assign('company',     'Megasoft Co-operation')
          ->assign('date',        $intlDateFormatter->format($timestamp))
          ->assign('image:photo', $photoFile);      // pass image file *without* path

$mailMerge->createDocument();

$document = $mailMerge->retrieveDocument('pdf');

file_put_contents('conference-pass-document.pdf', $document);

$mailMerge->deleteImage($photoFilename);

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/conference-pass`.

311.2.5 Generating Bitmaps Image Files

In addition to document file formats, `MailMerge` also allows documents to be saved to a number of image file formats (*BMP*, *GIF*, *JPG*, *PNG* and *TIFF*). Each page of the document is saved to one file.

The following sample illustrates the use of `getBitmaps($fromPage, $toPage, $zoomFactor, $format)` and `getAllBitmaps($zoomFactor, $format)`.

`$fromPage` is the lower-bound page number of the page range that should be returned as an image and `$toPage` the upper-bound page number. `$zoomFactor` is the size of the images, as a percent, relative to the original page size. The range of this parameter is 10 to 400. `$format` is the format of the images returned by this method. The supported formats can be obtained by calling `getImageExportFormats()`.

```
use ZendService\LiveDocx\MailMerge;

$locale    = Locale::getDefault();
$timestamp = time();

$intlTimeFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::NONE, IntlDateFormatter::SHORT);

$intlDateFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

$mailMerge->setLocalTemplate('license-agreement-template.docx');

$mailMerge->assign('software', 'Magic Graphical Compression Suite v1.9')
          ->assign('licensee', 'Henry Döner-Meyer')
          ->assign('company',  'Co-Operation')
          ->assign('date',     $intlDateFormatter->format($timestamp))
          ->assign('time',     $intlTimeFormatter->format($timestamp))
```

```
            ->assign('city',     'Lyon')
            ->assign('country',  'France');

$mailMerge->createDocument();

// Get all bitmaps
// (zoomFactor, format)
$bitmaps = $mailMerge->getAllBitmaps(100, 'png');

// Get just bitmaps in specified range
// (fromPage, toPage, zoomFactor, format)
//$bitmaps = $mailMerge->getBitmaps(2, 2, 100, 'png');

foreach ($bitmaps as $pageNumber => $bitmapData) {
    $filename = sprintf('license-agreement-page-%d.png', $pageNumber);
    file_put_contents($filename, $bitmapData);
}

unset($mailMerge);
```

This produces two files (`license-agreement-page-1.png` and `license-agreement-page-2.png`) and writes them to disk in the same directory as the executable *PHP* file.

license-agreement-page-1.png.

license-agreement-page-2.png. For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/bitmaps`.

311.2.6 Local vs. Remote Templates

Templates can be stored **locally**, on the client machine, or **remotely**, by LiveDocx. There are advantages and disadvantages to each approach.

In the case that a template is stored locally, it must be transferred from the client to LiveDocx on every request. If the content of the template rarely changes, this approach is inefficient. Similarly, if the template is several megabytes in size, it may take considerable time to transfer it to LiveDocx. Local template are useful in situations in which the content of the template is constantly changing.

The following code illustrates how to use a local template.

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

$mailMerge->setLocalTemplate('template.docx');

// assign data and create document

unset($mailMerge);
```

In the case that a template is stored remotely, it is uploaded once to LiveDocx and then simply referenced on all subsequent requests. Obviously, this is much quicker than using a local template, as the template does not have to be transferred on every request. For speed critical applications, it is recommended to use the remote template method.

The following code illustrates how to upload a template to the server:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

$mailMerge->uploadTemplate('template.docx');

unset($mailMerge);
```

The following code illustrates how to reference the remotely stored template on all subsequent requests:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

$mailMerge->setRemoteTemplate('template.docx');

// assign data and create document

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/templates`.

311.2.7 Getting Information

`ZendService\LiveDocx\MailMerge` provides a number of methods to get information on field names, available fonts and supported formats.

Get Array of Field Names in Template

The following code returns and displays an array of all field names in the specified template. This functionality is useful, in the case that you create an application, in which an end-user can update a template.

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

$templateName = 'template-1-text-field.docx';
$mailMerge->setLocalTemplate($templateName);

$fieldNames = $mailMerge->getFieldNames();
foreach ($fieldNames as $fieldName) {
    printf('- %s%s', $fieldName, PHP_EOL);
}
```

```

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/template-info`.

Get Array of Block Field Names in Template

The following code returns and displays an array of all block field names in the specified template. This functionality is useful, in the case that you create an application, in which an end-user can update a template. Before such templates can be populated, it is necessary to find out the names of the contained block fields.

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

$templateName = 'template-block-fields.doc';
$mailMerge->setLocalTemplate($templateName);

$blockNames = $mailMerge->getBlockNames();
foreach ($blockNames as $blockName) {
    $blockFieldNames = $mailMerge->getBlockFieldNames($blockName);
    foreach ($blockFieldNames as $blockFieldName) {
        printf('- %s::%s%s', $blockName, $blockFieldName, PHP_EOL);
    }
}

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/template-info`.

Get Array of Fonts Installed on Server

The following code returns and displays an array of all fonts installed on the server. You can use this method to present a list of fonts which may be used in a template. It is important to inform the end-user about the fonts installed on the server, as only these fonts may be used in a template. In the case that a template contains fonts, which are not available on the server, font-substitution will take place. This may lead to undesirable results.

```
use ZendService\LiveDocx\MailMerge;
use Zend\Debug\Debug;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

Debug::dump($mailMerge->getFontNames());

unset($mailMerge);
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend\Cache\Cache`- this will considerably speed up your application.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-fonts`.

Get Array of Supported Template File Formats

The following code returns and displays an array of all supported template file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the input format of the documentation generation process.

```
use ZendService\LiveDocx\MailMerge;
use Zend\Debug\Debug;

$mailMerge = new MailMerge()

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

Debug::dump($mailMerge->getTemplateFormats());

unset($mailMerge);
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend\Cache\Cache`- this will considerably speed up your application.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-formats`.

Get Array of Supported Document File Formats

The following code returns and displays an array of all supported document file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the output format of the documentation generation process.

```
use ZendService\LiveDocx\MailMerge;
use Zend\Debug\Debug;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

Debug::dump($mailMerge->getDocumentFormats());

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-formats`.

Get Array of Supported Image File Formats

The following code returns and displays an array of all supported image file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the output format of the documentation generation process.

```
use ZendService\LiveDocx\MailMerge;
use Zend\Debug\Debug;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_

Debug::dump($mailMerge->getImageExportFormats());

unset($mailMerge);
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend\Cache\Cache`- this will considerably speed up your application.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-formats`.

311.2.8 Upgrading From LiveDocx Free to LiveDocx Premium

LiveDocx Free is provided by *Text Control GmbH* completely free for charge. It is free for all to use in an unlimited number of applications. However, there are times when you may like to update to LiveDocx Premium. For example, you need to generate a very large number of documents concurrently, or your application requires documents to be created faster than LiveDocx Free permits. For such scenarios, *Text Control GmbH* offers LiveDocx Premium, a paid service with a number of benefits. For an overview of the benefits, please take a look at LiveDocx pricing.

This section of the manual offers a technical overview of how to upgrade from LiveDocx Free to LiveDocx Premium.

All you have to do, is make a very small change to the code that runs with LiveDocx Free. Your instantiation and initialization of LiveDocx Free probably looks as follows:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge()

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);

// rest of your application here

unset($mailMerge);
```

To use LiveDocx Premium, you simply need to change the service value from `MailMerge::SERVICE_FREE` to `MailMerge::SERVICE_PREMIUM`, and set the username and password assigned to you for Livedocx Premium. This may, or may not be the same as the credentials for LiveDocx Free. For example:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge()

```

```
$mailMerge->setUsername('myPremiumUsername')
          ->setPassword('myPremiumPassword')
          ->setService (MailMerge::SERVICE_PREMIUM);

// rest of your application here

unset($mailMerge);
```

And that is all there is to it. The assignment of the premium WSDL to the component is handled internally and automatically. You are now using LiveDocx Premium.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/instantiation`.

311.2.9 Upgrading From LiveDocx Free or LiveDocx Premium to LiveDocx Fully Licensed

LiveDocx Free and Livedocx Premium are provided by *Text Control GmbH* as a service. They are addressed over the Internet. However, for certain applications, for example, ones that process very sensitive data (banking, health or financial), you may not want to send your data across the Internet to a third party service, regardless of the SSL encryption that both LiveDocx Free and Livedocx Premium offer as standard. For such scenarios, you can license LiveDocx and install an entire LiveDocx server in your own network. As such, you completely control the flow of data between your application and the backend LiveDocx server. For an overview of the benefits of LiveDocx Fully Licensed, please take a look at LiveDocx pricing.

This section of the manual offers a technical overview of how to upgrade from LiveDocx Free or LiveDocx Premium to LiveDocx Fully Licensed.

All you have to do, is make a very small change to the code that runs with LiveDocx Free or LiveDocx Premium. Your instantiation and initialization of LiveDocx Free or LiveDocx Premium probably looks as follows:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge()

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);
       // or
       // ->setService (MailMerge::SERVICE_PREMIUM);

// rest of your application here

unset($mailMerge);
```

To use LiveDocx Fully Licensed, you simply need to set the WSDL of the backend LiveDocx server in your own network. You can do this as follows:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge()

$mailMerge->setUsername('myFullyLicensedUsername')
          ->setPassword('myFullyLicensedPassword')
          ->setWsdl    ('http://api.example.com/2.1/mailmerge.asmx?wsdl');

// rest of your application here
```

```
10
11 unset($mailMerge);
```

And that is all there is to it. You are now using LiveDocx Fully Licensed.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/instantiation`.

CHAPTER

TWO

ZENDSERVICENIRVANIX

312.1 Introduction

Nirvanix provides an Internet Media File System (IMFS), an Internet storage service that allows applications to upload, store and organize files and subsequently access them using a standard Web Services interface. An IMFS is distributed clustered file system, accessed over the Internet, and optimized for dealing with media files (audio, video, etc). The goal of an IMFS is to provide massive scalability to deal with the challenges of media storage growth, with guaranteed access and availability regardless of time and location. Finally, an IMFS gives applications the ability to access data securely, without the large fixed costs associated with acquiring and maintaining physical storage assets.

312.2 Registering with Nirvanix

Before you can get started with `ZendService\Nirvanix`, you must first register for an account. Please see the Getting Started page on the Nirvanix website for more information.

After registering, you will receive a Username, Password, and Application Key. All three are required to use `ZendService\Nirvanix`.

312.3 API Documentation

Access to the Nirvanix IMFS is available through both *SOAP* and a faster REST service. `ZendService\Nirvanix` provides a relatively thin *PHP* 5 wrapper around the REST service.

`ZendService\Nirvanix` aims to make using the Nirvanix REST service easier but understanding the service itself is still essential to be successful with Nirvanix.

The Nirvanix API Documentation provides an overview as well as detailed information using the service. Please familiarize yourself with this document and refer back to it as you use `ZendService\Nirvanix`.

312.4 Features

Nirvanix's REST service can be used effectively with *PHP* using the SimpleXML extension and `Zend\Http\Client` alone. However, using it this way is somewhat inconvenient due to repetitive operations like passing the session token on every request and repeatedly checking the response body for error codes.

`ZendService\Nirvanix` provides the following functionality:

- A single point for configuring your Nirvanix authentication credentials that can be used across the Nirvanix namespaces.
- A proxy object that is more convenient to use than an *HTTP* client alone, mostly removing the need to manually construct *HTTP* POST requests to access the REST service.
- A response wrapper that parses each response body and throws an exception if an error occurred, alleviating the need to repeatedly check the success of many commands.
- Additional convenience methods for some of the more common operations.

312.5 Getting Started

Once you have registered with Nirvanix, you're ready to store your first file on the IMFS. The most common operations that you will need to do on the IMFS are creating a new file, downloading an existing file, and deleting a file. `ZendService\Nirvanix` provides convenience methods for these three operations.

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new ZendService\Nirvanix\Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$imfs->putContents('/foo.txt', 'contents to store');

echo $imfs->getContents('/foo.txt');

$imfs->unlink('/foo.txt');
```

The first step to using `ZendService\Nirvanix` is always to authenticate against the service. This is done by passing your credentials to the `ZendService\Nirvanix` constructor above. The associative array is passed directly to Nirvanix as POST parameters.

Nirvanix divides its web services into namespaces. Each namespace encapsulates a group of related operations. After getting an instance of `ZendService\Nirvanix`, call the `getService()` method to create a proxy for the namespace you want to use. Above, a proxy for the `IMFS` namespace is created.

After you have a proxy for the namespace you want to use, call methods on it. The proxy will allow you to use any command available on the REST *API*. The proxy may also make convenience methods available, which wrap web service commands. The example above shows using the IMFS convenience methods to create a new file, retrieve and display that file, and finally delete the file.

312.6 Understanding the Proxy

In the previous example, we used the `getService()` method to return a proxy object to the `IMFS` namespace. The proxy object allows you to use the Nirvanix REST service in a way that's closer to making a normal *PHP* method call, as opposed to constructing your own *HTTP* request objects.

A proxy object may provide convenience methods. These are methods that the `ZendService\Nirvanix` provides to simplify the use of the Nirvanix web services. In the previous example, the methods `putContents()`, `getContents()`, and `unlink()` do not have direct equivalents in the REST *API*. They are convenience methods provided by `ZendService\Nirvanix` that abstract more complicated operations on the REST *API*.

For all other method calls to the proxy object, the proxy will dynamically convert the method call to the equivalent *HTTP* POST request to the REST *API*. It does this by using the method name as the *API* command, and an associative array in the first argument as the POST parameters.

Let's say you want to call the REST *API* method RenameFile, which does not have a convenience method in `ZendService\Nirvanix`:

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new ZendService\Nirvanix\Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$result = $imfs->renameFile(array('filePath' => '/path/to/foo.txt',
                                  'newFileName' => 'bar.txt'));
```

Above, a proxy for the `IMFS` namespace is created. A method, `renameFile()`, is then called on the proxy. This method does not exist as a convenience method in the *PHP* code, so it is trapped by `__call()` and converted into a POST request to the REST *API* where the associative array is used as the POST parameters.

Notice in the Nirvanix *API* documentation that *sessionToken* is required for this method but we did not give it to the proxy object. It is added automatically for your convenience.

The result of this operation will either be a `ZendService\Nirvanix\Response` object wrapping the *XML* returned by Nirvanix, or a `ZendService\Nirvanix\Exception` if an error occurred.

312.7 Examining Results

The Nirvanix REST *API* always returns its results in *XML*. `ZendService\Nirvanix` parses this *XML* with the *SimpleXML* extension and then decorates the resulting *SimpleXMLElement* with a `ZendService\Nirvanix\Response` object.

The simplest way to examine a result from the service is to use the built-in *PHP* functions like `print_r()`:

```
<?php
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new ZendService\Nirvanix\Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$result = $imfs->putContents('/foo.txt', 'fourteen bytes');
print_r($result);
?>

ZendService\Nirvanix\Response Object
(
    [_sxml:protected] => SimpleXMLElement Object
        (
            [ResponseCode] => 0
            [FilesUploaded] => 1
            [BytesUploaded] => 14
        )
)
```

You can access any property or method of the decorated *SimpleXMLElement*. In the above example, *$result->BytesUploaded* could be used to see the number of bytes received. Should you want to access the *SimpleXMLElement* directly, just use *$result->getSxml()*.

The most common response from Nirvanix is success (*ResponseCode* of zero). It is not normally necessary to check *ResponseCode* because any non-zero result will throw a `ZendService\Nirvanix\Exception`. See the next section on handling errors.

312.8 Handling Errors

When using Nirvanix, it's important to anticipate errors that can be returned by the service and handle them appropriately.

All operations against the REST service result in an *XML* return payload that contains a *ResponseCode* element, such as the following example:

```
<Response>
   <ResponseCode>0</ResponseCode>
</Response>
```

When the *ResponseCode* is zero such as in the example above, the operation was successful. When the operation is not successful, the *ResponseCode* is non-zero and an *ErrorMessage* element should be present.

To alleviate the need to repeatedly check if the *ResponseCode* is non-zero, `ZendService\Nirvanix` automatically checks each response returned by Nirvanix. If the *ResponseCode* indicates an error, a `ZendService\Nirvanix\Exception` will be thrown.

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');
$nirvanix = new ZendService\Nirvanix\Nirvanix($auth);

try {

  $imfs = $nirvanix->getService('IMFS');
  $imfs->unlink('/a-nonexistant-path');

} catch (ZendService\Nirvanix\Exception\DomainException $e) {
  echo $e->getMessage() . "\n";
  echo $e->getCode();
}
```

In the example above, `unlink()` is a convenience method that wraps the *DeleteFiles* command on the REST *API*. The *filePath* parameter required by the DeleteFiles command contains a path that does not exist. This will result in a `ZendService\Nirvanix` exception being thrown with the message "Invalid path" and code 70005.

The Nirvanix API Documentation describes the errors associated with each command. Depending on your needs, you may wrap each command in a *try* block or wrap many commands in the same *try* block for convenience.

CHAPTER

THREE

ZEND\SERVICE\RACKSPACE

313.1 Introduction

The `ZendService\Rackspace` is a class that provides a simple *API* to manage the Rackspace services Cloud Files and Cloud Servers.

Note: Load balancers service

The load balancers service of Rackspace is not implemented yet. We are planning to release it in the next future.

313.2 Registering with Rackspace

Before you can get started with `ZendService\Rackspace`, you must first register for an account. Please see the Cloud services page on the Rackspace website for more information.

After registering, you can get the Username and the API Key from the Rackspace management console under the menu "Your Account" > "API Access". These informations are required to use the `ZendService\Rackspace` classes.

313.3 Cloud Files

The Cloud Files is a service to store any files in a cloud environment. A user can store an unlimited quantity of files and each file can be as large as 5 gigabytes. The files can be private or public. The private files can be accessed using the API of Rackspace. The public files are accessed using a *CDN* (Content Delivery Network). Rackspace exposes a *REST* API to manage the Cloud Files.

`ZendService\Rackspace\Files` provides the following functionality:

- Upload files programmatically for tight integration with your application
- Enable Cloud Files CDN integration on any container for public distribution
- Create Containers programmatically
- Retrieve lists of containers and files

313.4 Cloud Servers

Rackspace Cloud Servers is a compute service that provides server capacity in the cloud. Cloud Servers come in different flavors of memory, disk space, and CPU.

`ZendService\Rackspace\Servers` provides the following functionality:

- Create/delete new servers
- List and get information on each server
- Manage the public/private IP addresses of a server
- Resize the server capacity
- Reboot a server
- Create new images for a server
- Manage the backup of a server
- Create a group of server to share the IP addresses for High Availability architecture

313.5 Available Methods

Eeach service class (Files, Servers) of Rackspace extends the `ZendService\Rackspace` abstract class. This class contains a set of public methods shared with all the service. This public methods are reported as follow:

authenticate `authenticate()`

Authenticate the Rackspace API using the user and the key specified in the concrete class that extend `ZendService\Rackspace`. Return **true** in case of success and **false** in case of error.

getAuthUrl `getAuthUrl()`

Get the authentication URL of Rackspace. Returns a string.

getCdnUrl `getCdnUrl()`

Get the URL for the CDN. Returns a string.

getErrorCode `getErrorCode()`

Get the last HTTP error code. Returns a string.

getErrorMsg `getErrorMsg()`

Get the last error message. Returns a string.

getHttpClient `getHttpClient()`

Get the HTTP client used to call the API of the Rackspace. Returns a `Zend\Http\Client` instance.

getKey `getKey()`

Get the authentication key. Returns a string.

getManagementUrl `getManagementUrl()`

Get the URL for the management services. Returns a string.

getStorageUrl `getStorageUrl()`

Get the URL for the storage (files) service. Returns a string.

getToken `getToken()`

Get the token returned after a successful authentication. Returns a string.

getUser `getUser()`

Get the user authenticated with the Rackspace service. Returns a string.

isSuccessful `isSuccessful()`

Return **true** if the last service call was successful, false otherwise.

setAuthUrl `setAuthUrl(string $url)`

Set the authentication URL to be used.

$url is the URL for the authentication

setKey `setKey(string $key)`

Set the key for the API authentication.

$key is the key string for the authentication

setUser `setUser(string $user)`

Set the user for the API authentication.

$user is the user string for the authentication

CHAPTER

FOUR

ZENDSERVICERECAPTCHA

314.1 Introduction

`ZendService\ReCaptcha` provides a client for the reCAPTCHA Web Service. Per the reCAPTCHA site, "reCAPTCHA is a free CAPTCHA service that helps to digitize books." Each reCAPTCHA requires the user to input two words, the first of which is the actual CAPTCHA, and the second of which is a word from some scanned text that Optical Character Recognition (OCR) software has been unable to identify. The assumption is that if a user correctly provides the first word, the second is likely correctly entered as well, and can be used to improve OCR software for digitizing books.

In order to use the reCAPTCHA service, you will need to sign up for an account and register one or more domains with the service in order to generate public and private keys.

314.2 Simplest use

Instantiate a `ZendService\ReCaptcha\ReCaptcha` object, passing it your public and private keys:

Creating an instance of the reCAPTCHA service

```
$recaptcha = new ZendService\ReCaptcha\ReCaptcha($pubKey, $privKey);
```

To render the reCAPTCHA, simply call the `getHTML()` method:

Displaying the reCAPTCHA

```
echo $recaptcha->getHTML();
```

When the form is submitted, you should receive two fields, 'recaptcha_challenge_field' and 'recaptcha_response_field'. Pass these to the reCAPTCHA object's `verify()` method:

Verifying the form fields

```
$result = $recaptcha->verify(
    $_POST['recaptcha_challenge_field'],
    $_POST['recaptcha_response_field']
);
```

Once you have the result, test against it to see if it is valid. The result is a `ZendService\ReCaptcha\Response` object, which provides an `isValid()` method.

Validating the reCAPTCHA

```
if (!$result->isValid()) {
    // Failed validation
}
```

It is even simpler to use *the reCAPTCHA* `Zend\Captcha` adapter, or to use that adapter as a backend for the *CAPTCHA form element*. In each case, the details of rendering and validating the reCAPTCHA are automated for you.

314.3 Hiding email addresses

`ZendService\ReCaptcha\MailHide` can be used to hide email addresses. It will replace a part of an email address with a link that opens a popup window with a reCAPTCHA challenge. Solving the challenge will reveal the complete email address.

In order to use this component you will need an account to generate public and private keys for the mailhide *API*.

Using the mail hide component

```
// The mail address we want to hide
$mail = 'mail@example.com';

// Create an instance of the mailhide component, passing it your public
// and private keys, as well as the mail address you want to hide
$mailHide = new ZendService\ReCaptcha\Mailhide();
$mailHide->setPublicKey($pubKey);
$mailHide->setPrivateKey($privKey);
$mailHide->setEmail($mail);

// Display it
print($mailHide);
```

The example above will display "m...@example.com" where "..." has a link that opens up a popup window with a reCAPTCHA challenge.

The public key, private key, and the email address can also be specified in the constructor of the class. A fourth argument also exists that enables you to set some options for the component. The available options are listed in the following table:

Table 314.1: ZendServiceReCaptchaMailHide options

Option	Description	Expected Values	Default Value
linkTitle	The title attribute of the link	string	'Reveal this e=mail address'
linkHiddenText	The text that includes the popup link	string	'...'
popupWidth	The width of the popup window	int	500
popupHeight	The height of the popup window	int	300

The configuration options can be set by sending them as the fourth argument to the constructor or by calling `setOptions($options)`, which takes an associative array or an instance of *ZendConfigConfig*.

Generating many hidden email addresses

```
// Create an instance of the mailhide component, passing it your public
// and private keys, as well as some configuration options
$mailHide = new ZendService\ReCaptcha\Mailhide();
$mailHide->setPublicKey($pubKey);
$mailHide->setPrivateKey($privKey);
$mailHide->setOptions(array(
    'linkTitle' => 'Click me',
    'linkHiddenText' => '+++++',
));

// The mail addresses we want to hide
$mailAddresses = array(
    'mail@example.com',
    'johndoe@example.com',
    'janedoe@example.com',
);

foreach ($mailAddresses as $mail) {
    $mailHide->setEmail($mail);
    print($mailHide);
}
```

CHAPTER

FIVE

ZENDSERVICESLIDESHARE

The `ZendService\SlideShare` component is used to interact with the slideshare.net web services for hosting slide shows online. With this component, you can embed slide shows which are hosted on this web site within a web site and even upload new slide shows to your account.

315.1 Getting Started with ZendServiceSlideShare

In order to use the `ZendService\SlideShare` component you must first create an account on the slideshare.net servers (more information can be found here) in order to receive an *API* key, username, password and shared secret value – all of which are needed in order to use the `ZendService\SlideShare` component.

Once you have setup an account, you can begin using the `ZendService\SlideShare` component by creating a new instance of the `ZendService\SlideShare` object and providing these values as shown below:

```
// Create a new instance of the component
$ss = new ZendService\SlideShare\SlideShare('APIKEY',
                                            'SHAREDSECRET',
                                            'USERNAME',
                                            'PASSWORD');
```

315.2 The SlideShow object

All slide shows in the `ZendService\SlideShare` component are represented using the `ZendService\SlideShare\SlideShow` object (both when retrieving and uploading new slide shows). For your reference a pseudo-code version of this class is provided below.

```
class ZendService\SlideShare\SlideShow {

    /**
     * Retrieves the location of the slide show
     */
    public function getLocation() {
        return $this->_location;
    }

    /**
     * Gets the transcript for this slide show
     */
    public function getTranscript() {
        return $this->_transcript;
```

```
    }

    /**
     * Adds a tag to the slide show
     */
    public function addTag($tag) {
        $this->_tags[] = (string)$tag;
        return $this;
    }

    /**
     * Sets the tags for the slide show
     */
    public function setTags(Array $tags) {
        $this->_tags = $tags;
        return $this;
    }

    /**
     * Gets all of the tags associated with the slide show
     */
    public function getTags() {
        return $this->_tags;
    }

    /**
     * Sets the filename on the local filesystem of the slide show
     * (for uploading a new slide show)
     */
    public function setFilename($file) {
        $this->_slideShowFilename = (string)$file;
        return $this;
    }

    /**
     * Retrieves the filename on the local filesystem of the slide show
     * which will be uploaded
     */
    public function getFilename() {
        return $this->_slideShowFilename;
    }

    /**
     * Gets the ID for the slide show
     */
    public function getId() {
        return $this->_slideShowId;
    }

    /**
     * Retrieves the HTML embed code for the slide show
     */
    public function getEmbedCode() {
        return $this->_embedCode;
    }

    /**
     * Retrieves the Thumbnail URi for the slide show
```

```
     */
    public function getThumbnailUrl() {
        return $this->_thumbnailUrl;
    }

    /**
     * Sets the title for the Slide show
     */
    public function setTitle($title) {
        $this->_title = (string)$title;
        return $this;
    }

    /**
     * Retrieves the Slide show title
     */
    public function getTitle() {
        return $this->_title;
    }

    /**
     * Sets the description for the Slide show
     */
    public function setDescription($desc) {
        $this->_description = (string)$desc;
        return $this;
    }

    /**
     * Gets the description of the slide show
     */
    public function getDescription() {
        return $this->_description;
    }

    /**
     * Gets the numeric status of the slide show on the server
     */
    public function getStatus() {
        return $this->_status;
    }

    /**
     * Gets the textual description of the status of the slide show on
     * the server
     */
    public function getStatusDescription() {
        return $this->_statusDescription;
    }

    /**
     * Gets the permanent link of the slide show
     */
    public function getPermaLink() {
        return $this->_permalink;
    }

    /**
```

```
     * Gets the number of views the slide show has received
     */
    public function getNumViews() {
        return $this->_numViews;
    }
}
```

Note: The above pseudo-class only shows those methods which should be used by end-user developers. Other available methods are internal to the component.

When using the `ZendService\SlideShare` component, this data class will be used frequently to browse or add new slide shows to or from the web service.

315.3 Retrieving a single slide show

The simplest usage of the `ZendService\SlideShare` component is the retrieval of a single slide show by slide show ID provided by the slideshare.net application and is done by calling the `getSlideShow()` method of a `ZendService\SlideShare` object and using the resulting `ZendService\SlideShare\SlideShow` object as shown.

```
// Create a new instance of the component
$ss = new ZendService\SlideShare\SlideShare('APIKEY',
                                            'SHAREDSECRET',
                                            'USERNAME',
                                            'PASSWORD');

$slideshow = $ss->getSlideShow(123456);

print "Slide Show Title: {$slideshow->getTitle()}<br/>\n";
print "Number of views: {$slideshow->getNumViews()}<br/>\n";
```

315.4 Retrieving Groups of Slide Shows

If you do not know the specific ID of a slide show you are interested in retrieving, you can retrieving groups of slide shows by using one of three methods:

- **Slide shows from a specific account**

  You can retrieve slide shows from a specific account by using the `getSlideShowsByUsername()` method and providing the username from which the slide shows should be retrieved

- **Slide shows which contain specific tags**

  You can retrieve slide shows which contain one or more specific tags by using the `getSlideShowsByTag()` method and providing one or more tags which the slide show must have assigned to it in order to be retrieved

- **Slide shows by group**

  You can retrieve slide shows which are a member of a specific group using the `getSlideShowsByGroup()` method and providing the name of the group which the slide show must belong to in order to be retrieved

Each of the above methods of retrieving multiple slide shows a similar approach is used. An example of using each method is shown below:

```
// Create a new instance of the component
$ss = new ZendService\SlideShare\SlideShare('APIKEY',
                                 'SHAREDSECRET',
                                 'USERNAME',
                                 'PASSWORD');

$starting_offset = 0;
$limit = 10;

// Retrieve the first 10 of each type
$ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
$ss_tags = $ss->getSlideShowsByTag('zend', $starting_offset, $limit);
$ss_group = $ss->getSlideShowsByGroup('mygroup', $starting_offset, $limit);

// Iterate over the slide shows
foreach ($ss_user as $slideshow) {
   print "Slide Show Title: {$slideshow->getTitle}<br/>\n";
}
```

315.5 ZendServiceSlideShare Caching policies

By default, `ZendService\SlideShare\SlideShare` will cache any request against the web service automatically to the filesystem (default path `/tmp`) for 12 hours. If you desire to change this behavior, you must provide your own *ZendCacheCache* object using the `setCacheObject()` method as shown:

```
$frontendOptions = array(
                       'lifetime' => 7200,
                       'automatic_serialization' => true);
$backendOptions  = array(
                       'cache_dir' => '/webtmp/');

$cache = Zend\Cache\Cache::factory('Core',
                            'File',
                            $frontendOptions,
                            $backendOptions);

$ss = new ZendService\SlideShare\SlideShare('APIKEY',
                                 'SHAREDSECRET',
                                 'USERNAME',
                                 'PASSWORD');
$ss->setCacheObject($cache);

$ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
```

315.6 Changing the behavior of the HTTP Client

If for whatever reason you would like to change the behavior of the *HTTP* client when making the web service request, you can do so by creating your own instance of the `Zend\Http\Client` object (see *ZendHttp*). This is useful for instance when it is desirable to set the timeout for the connection to something other then default as shown:

```
$client = new Zend\Http\Client();
$client->setConfig(array('timeout' => 5));

```

```
4 $ss = new ZendService\SlideShare\SlideShare('APIKEY',
5                                   'SHAREDSECRET',
6                                   'USERNAME',
7                                   'PASSWORD');
8 $ss->setHttpClient($client);
9 $ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
```

CHAPTER

SIX

ZENDSERVICESTRIKEIRON

`ZendService\StrikeIron` provides a *PHP* 5 client to StrikeIron web services. See the following sections:

- *ZendServiceStrikeIron*
- *Bundled Services*
- *Advanced Use*

316.1 Overview

StrikeIron offers hundreds of commercial data services ("Data as a Service") such as Online Sales Tax, Currency Rates, Stock Quotes, Geocodes, Global Address Verification, Yellow/White Pages, MapQuest Driving Directions, Dun & Bradstreet Business Credit Checks, and much, much more.

Each StrikeIron web service shares a standard *SOAP* (and REST) *API*, making it easy to integrate and manage multiple services. StrikeIron also manages customer billing for all services in a single account, making it perfect for solution providers. Get started with free web services at http://www.strikeiron.com/sdp.

StrikeIron's services may be used through the PHP 5 SOAP extension alone. However, using StrikeIron this way does not give an ideal *PHP*-like interface. The `ZendService\StrikeIron` component provides a lightweight layer on top of the *SOAP* extension for working with StrikeIron services in a more convenient, *PHP*-like manner.

Note: The *PHP* 5 *SOAP* extension must be installed and enabled to use `ZendService\StrikeIron`.

The `ZendService\StrikeIron` component provides:

- A single point for configuring your StrikeIron authentication credentials that can be used across many StrikeIron services.
- A standard way of retrieving your StrikeIron subscription information such as license status and the number of hits remaining to a service.
- The ability to use any StrikeIron service from its WSDL without creating a *PHP* wrapper class, and the option of creating a wrapper for a more convenient interface.
- Wrappers for three popular StrikeIron services.

316.2 Registering with StrikeIron

Before you can get started with `ZendService\StrikeIron`, you must first register for a StrikeIron developer account.

After registering, you will receive a StrikeIron username and password. These will be used when connecting to StrikeIron using `ZendService\StrikeIron`.

You will also need to sign up for StrikeIron's Super Data Pack Web Service.

Both registration steps are free and can be done relatively quickly through the StrikeIron website.

316.3 Getting Started

Once you have registered for a StrikeIron account and signed up for the Super Data Pack, you're ready to start using `ZendService\StrikeIron`.

StrikeIron consists of hundreds of different web services. `ZendService\StrikeIron` can be used with many of these services but provides supported wrappers for three of them:

- *ZIP Code Information*
- *US Address Verification*
- *Sales & Use Tax Basic*

The class `ZendService\StrikeIron` provides a simple way of specifying your StrikeIron account information and other options in its constructor. It also has a factory method that will return clients for StrikeIron services:

```
$strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username',
                                                 'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
```

The `getService()` method will return a client for any StrikeIron service by the name of its *PHP* wrapper class. In this case, the name 'SalesUseTaxBasic' refers to the wrapper class `ZendService\StrikeIron\SalesUseTaxBasic`. Wrappers are included for three services and described in *Bundled Services*.

The `getService()` method can also return a client for a StrikeIron service that does not yet have a *PHP* wrapper. This is explained in *Using Services by WSDL*.

316.4 Making Your First Query

Once you have used the `getService()` method to get a client for a particular StrikeIron service, you can utilize that client by calling methods on it just like any other *PHP* object.

```
$strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username',
                                                 'password' => 'your-password'));

// Get a client for the Sales & Use Tax Basic service
$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

// Query tax rate for Ontario, Canada
$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
echo $rateInfo->province;
echo $rateInfo->abbreviation;
echo $rateInfo->GST;
```

In the example above, the `getService()` method is used to return a client to the *Sales & Use Tax Basic* service. The client object is stored in `$taxBasic`.

The `getTaxRateCanada()` method is then called on the service. An associative array is used to supply keyword parameters to the method. This is the way that all StrikeIron methods are called.

The result from `getTaxRateCanada()` is stored in `$rateInfo` and has properties like `province` and `GST`.

Many of the StrikeIron services are as simple to use as the example above. See *Bundled Services* for detailed information on three StrikeIron services.

316.5 Examining Results

When learning or debugging the StrikeIron services, it's often useful to dump the result returned from a method call. The result will always be an object that is an instance of `ZendService\StrikeIron\Decorator`. This is a small decorator object that wraps the results from the method call.

The simplest way to examine a result from the service is to use the built-in *PHP* functions like print_r():

```
<?php
$strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username',
                                                          'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
print_r($rateInfo);
?>

ZendService\StrikeIron\Decorator Object
(
    [_name:protected] => GetTaxRateCanadaResult
    [_object:protected] => stdClass Object
        (
            [abbreviation] => ON
            [province] => ONTARIO
            [GST] => 0.06
            [PST] => 0.08
            [total] => 0.14
            [HST] => Y
        )
)
```

In the output above, we see that the decorator (`$rateInfo`) wraps an object named `GetTaxRateCanadaResult`, the result of the call to `getTaxRateCanada()`.

This means that `$rateInfo` has public properties like `abbreviation`, `province`>, and `GST`. These are accessed like `$rateInfo->province`.

Tip: StrikeIron result properties sometimes start with an uppercase letter such as `Foo` or `Bar` where most *PHP* object properties normally start with a lowercase letter as in `foo` or `bar`. The decorator will automatically do this inflection so you may read a property `Foo` as `foo`.

If you ever need to get the original object or its name out of the decorator, use the respective methods `getDecoratedObject()` and `getDecoratedObjectName()`.

316.6 Handling Errors

The previous examples are naive, i.e. no error handling was shown. It's possible that StrikeIron will return a fault during a method call. Events like bad account credentials or an expired subscription can cause StrikeIron to raise a fault.

An exception will be thrown when such a fault occurs. You should anticipate and catch these exceptions when making method calls to the service:

```
$strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username',
                                                 'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

try {

  $taxBasic->getTaxRateCanada(array('province' => 'ontario'));

} catch (ZendService\StrikeIron\Exception\RuntimeException $e) {

  // error handling for events like connection
  // problems or subscription errors

}
```

The exceptions thrown will always be `ZendService\StrikeIron\Exception`.

It's important to understand the difference between exceptions and normal failed method calls. Exceptions occur for **exceptional** conditions, such as the network going down or your subscription expiring. Failed method calls that are a common occurrence, such as `getTaxRateCanada()` not finding the `province` you supplied, will not result an in exception.

Note: Every time you make a method call to a StrikeIron service, you should check the response object for validity and also be prepared to catch an exception.

316.7 Checking Your Subscription

StrikeIron provides many different services. Some of these are free, some are available on a trial basis, and some are pay subscription only. When using StrikeIron, it's important to be aware of your subscription status for the services you are using and check it regularly.

Each StrikeIron client returned by the `getService()` method has the ability to check the subscription status for that service using the `getSubscriptionInfo()` method of the client:

```
// Get a client for the Sales & Use Tax Basic service
$strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username',
                                                 'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class => 'SalesUseTaxBasic'));

// Check remaining hits for the Sales & Use Tax Basic service
$subscription = $taxBasic->getSubscriptionInfo();
echo $subscription->remainingHits;
```

The `getSubscriptionInfo()` method will return an object that typically has a `remainingHits` property. It's important to check the status on each service that you are using. If a method call is made to StrikeIron after the remaining hits have been used up, an exception will occur.

Checking your subscription to a service does not use any remaining hits to the service. Each time any method call to the service is made, the number of hits remaining will be cached and this cached value will be returned by `getSubscriptionInfo()` without connecting to the service again. To force `getSubscriptionInfo()` to override its cache and query the subscription information again, use `getSubscriptionInfo(true)`.

CHAPTER

SEVEN

ZENDSERVICETECHNORATI

317.1 Introduction

`ZendService\Technorati` provides an easy, intuitive and object-oriented interface for using the Technorati *API*. It provides access to all available Technorati API queries and returns the original *XML* response as a friendly *PHP* object.

Technorati is one of the most popular blog search engines. The *API* interface enables developers to retrieve information about a specific blog, search blogs matching a single tag or phrase and get information about a specific author (blogger). For a full list of available queries please see the Technorati API documentation or the *Available Technorati queries* section of this document.

317.2 Getting Started

Technorati requires a valid *API* key for usage. To get your own *API* Key you first need to create a new Technorati account, then visit the API Key section.

Note: API Key limits

You can make up to 500 Technorati *API* calls per day, at no charge. Other usage limitations may apply, depending on the current Technorati *API* license.

Once you have a valid *API* key, you're ready to start using `ZendService\Technorati`.

317.3 Making Your First Query

In order to run a query, first you need a `ZendService\Technorati` instance with a valid *API* key. Then choose one of the available query methods, and call it providing required arguments.

Sending your first query

```
// create a new ZendService\Technorati
// with a valid API_KEY
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');

```

```
// search Technorati for PHP keyword
$resultSet = $technorati->search('PHP');
```

Each query method accepts an array of optional parameters that can be used to refine your query.

Refining your query

```
// create a new ZendService\Technorati
// with a valid API_KEY
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');

// filter your query including only results
// with some authority (Results from blogs with a handful of links)
$options = array('authority' => 'a4');

// search Technorati for PHP keyword
$resultSet = $technorati->search('PHP', $options);
```

A `ZendService\Technorati` instance is not a single-use object. That is, you don't need to create a new instance for each query call; simply use your current `ZendService\Technorati` object as long as you need it.

Sending multiple queries with the same ZendServiceTechnorati instance

```
// create a new ZendService\Technorati
// with a valid API_KEY
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
$search = $technorati->search('PHP');

// get top tags indexed by Technorati
$topTags = $technorati->topTags();
```

317.4 Consuming Results

You can get one of two types of result object in response to a query.

The first group is represented by `ZendService\Technorati\*ResultSet` objects. A result set object is basically a collection of result objects. It extends the basic `ZendService\Technorati\ResultSet` class and implements the `SeekableIterator` *PHP* interface. The best way to consume a result set object is to loop over it with the *PHP* `foreach()` statement.

Consuming a result set object

```
// create a new ZendService\Technorati
// with a valid API_KEY
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
// $resultSet is an instance of ZendService\Technorati\SearchResultSet
$resultSet = $technorati->search('PHP');
```

```

// loop over all result objects
foreach ($resultSet as $result) {
    // $result is an instance of ZendService\Technorati\SearchResult
}
```

Because `ZendService\Technorati\ResultSet` implements the `SeekableIterator` interface, you can seek a specific result object using its position in the result collection.

Seeking a specific result set object

```
// create a new ZendService\Technorati\Technorati
// with a valid API_KEY
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
// $resultSet is an instance of ZendService\Technorati\SearchResultSet
$resultSet = $technorati->search('PHP');

// $result is an instance of ZendService\Technorati\SearchResult
$resultSet->seek(1);
$result = $resultSet->current();
```

Note: `SeekableIterator` works as an array and counts positions starting from index 0. Fetching position number 1 means getting the second result in the collection.

The second group is represented by special standalone result objects. `ZendService\Technorati\GetInfoResult`, `ZendService\Technorati\BlogInfoResult` and `ZendService\Technorati\KeyInfoResult` act as wrappers for additional objects, such as `ZendService\Technorati\Author` and `ZendService\Technorati\Weblog`.

Consuming a standalone result object

```
// create a new ZendService\Technorati\Technorati
// with a valid API_KEY
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');

// get info about weppos author
$result = $technorati->getInfo('weppos');

$author = $result->getAuthor();
echo '<h2>Blogs authored by ' . $author->getFirstName() . " " .
          $author->getLastName() . '</h2>';
echo '<ol>';
foreach ($result->getWeblogs() as $weblog) {
    echo '<li>' . $weblog->getName() . '</li>';
}
echo "</ol>";
```

Please read the *ZendServiceTechnorati Classes* section for further details about response classes.

317.5 Handling Errors

Each `ZendService\Technorati` query method throws a `ZendService\Technorati\Exception` exception on failure with a meaningful error message.

There are several reasons that may cause a `ZendService\Technorati` query to fail. `ZendService\Technorati` validates all parameters for any query request. If a parameter is invalid or it contains an invalid value, a new `ZendService\Technorati\Exception` exception is thrown. Additionally, the Technorati *API* interface could be temporally unavailable, or it could return a response that is not well formed.

You should always wrap a Technorati query with a `try ... catch` block.

Handling a Query Exception

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
try {
    $resultSet = $technorati->search('PHP');
} catch(ZendService\Technorati\Exception $e) {
    echo "An error occurred: " $e->getMessage();
}
```

317.6 Checking Your API Key Daily Usage

From time to time you probably will want to check your *API* key daily usage. By default Technorati limits your *API* usage to 500 calls per day, and an exception is returned by `ZendService\Technorati` if you try to use it beyond this limit. You can get information about your *API* key usage using the `ZendService\Technorati::keyInfo()` method.

`ZendService\Technorati::keyInfo()` returns a `ZendService\Technorati\KeyInfoResult` object. For full details please see the API reference guide.

Getting API key daily usage information

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$key = $technorati->keyInfo();

echo "API Key: " . $key->getApiKey() . "<br />";
echo "Daily Usage: " . $key->getApiQueries() . "/" .
     $key->getMaxQueries() . "<br />";
```

317.7 Available Technorati Queries

`ZendService\Technorati` provides support for the following queries:

- *Cosmos*
- *Search*
- *Tag*
- *DailyCounts*

- *TopTags*
- *BlogInfo*
- *BlogPostTags*
- *GetInfo*

317.7.1 Technorati Cosmos

Cosmos query lets you see what blogs are linking to a given *URL*. It returns a *ZendServiceTechnoratiCosmosResultSet* object. For full details please see `ZendService\Technorati::cosmos()` in the API reference guide.

Cosmos Query

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$resultSet = $technorati->cosmos('http://devzone.zend.com/');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

317.7.2 Technorati Search

The Search query lets you see what blogs contain a given search string. It returns a *ZendServiceTechnoratiSearchResultSet* object. For full details please see `ZendService\Technorati\Technorati::search()` in the API reference guide.

Search Query

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$resultSet = $technorati->search('zend framework');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

317.7.3 Technorati Tag

The Tag query lets you see what posts are associated with a given tag. It returns a *ZendServiceTechnoratiTagResultSet* object. For full details please see `ZendService\Technorati\Technorati::tag()` in the API reference guide.

Tag Query

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$resultSet = $technorati->tag('php');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

317.7.4 Technorati DailyCounts

The DailyCounts query provides daily counts of posts containing the queried keyword. It returns a *ZendServiceTechnoratiDailyCountsResultSet* object. For full details please see `ZendService\Technorati::dailyCounts()` in the API reference guide.

DailyCounts Query

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$resultSet = $technorati->dailyCounts('php');

foreach ($resultSet as $result) {
    echo "<li>" . $result->getDate() .
         "(" . $result->getCount() . ")</li>";
}
echo "</ol>";
```

317.7.5 Technorati TopTags

The TopTags query provides information on top tags indexed by Technorati. It returns a *ZendServiceTechnoratiTagsResultSet* object. For full details please see `ZendService\Technorati\Technorati::topTags()` in the API reference guide.

TopTags Query

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$resultSet = $technorati->topTags();

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getTag() . "</li>";
}
echo "</ol>";
```

317.7.6 Technorati BlogInfo

The BlogInfo query provides information on what blog, if any, is associated with a given *URL*. It returns a *ZendServiceTechnoratiBlogInfoResult* object. For full details please see `ZendService\Technorati\Technorati::blogInfo()` in the API reference guide.

BlogInfo Query

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$result = $technorati->blogInfo('http://devzone.zend.com/');

echo '<h2><a href="' . (string) $result->getWeblog()->getUrl() . '">' .
     $result->getWeblog()->getName() . '</a></h2>';
```

317.7.7 Technorati BlogPostTags

The BlogPostTags query provides information on the top tags used by a specific blog. It returns a *ZendServiceTechnoratiTagsResultSet* object. For full details please see `ZendService\Technorati\Technorati::blogPostTags()` in the API reference guide.

BlogPostTags Query

```
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$resultSet = $technorati->blogPostTags('http://devzone.zend.com/');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getTag() . "</li>";
}
echo "</ol>";
```

317.7.8 Technorati GetInfo

The GetInfo query tells you things that Technorati knows about a member. It returns a *ZendServiceTechnoratiGetInfoResult* object. For full details please see `ZendService\Technorati\Technorati::getInfo()` in the API reference guide.

GetInfo Query

```
$technorati = new ZendService\Technorati('VALID_API_KEY');
$result = $technorati->getInfo('weppos');

$author = $result->getAuthor();
echo "<h2>Blogs authored by " . $author->getFirstName() . " " .
     $author->getLastName() . "</h2>";
echo "<ol>";
foreach ($result->getWeblogs() as $weblog) {
```

```
    echo "<li>" . $weblog->getName() . "</li>";
}
echo "</ol>";
```

317.7.9 Technorati KeyInfo

The KeyInfo query provides information on daily usage of an *API* key. It returns a *ZendServiceTechnoratiKeyInfoResult* object. For full details please see `ZendService\Technorati\Technorati::keyInfo()` in the API reference guide.

317.8 ZendServiceTechnorati Classes

The following classes are returned by the various Technorati queries. Each `ZendService\Technorati\*ResultSet` class holds a type-specific result set which can be easily iterated, with each result being contained in a type result object. All result set classes extend `ZendService\Technorati\ResultSet` class and implement the `SeekableIterator` interface, allowing for easy iteration and seeking to a specific result.

- *ZendServiceTechnoratiResultSet*
- *ZendServiceTechnoratiCosmosResultSet*
- *ZendServiceTechnoratiSearchResultSet*
- *ZendServiceTechnoratiTagResultSet*
- *ZendServiceTechnoratiDailyCountsResultSet*
- *ZendServiceTechnoratiTagsResultSet*
- *ZendServiceTechnoratiResult*
- *ZendServiceTechnoratiCosmosResult*
- *ZendServiceTechnoratiSearchResult*
- *ZendServiceTechnoratiTagResult*
- *ZendServiceTechnoratiDailyCountsResult*
- *ZendServiceTechnoratiTagsResult*
- *ZendServiceTechnoratiGetInfoResult*
- *ZendServiceTechnoratiBlogInfoResult*
- *ZendServiceTechnoratiKeyInfoResult*

Note: `ZendService\Technorati\GetInfoResult`, `ZendService\Technorati\BlogInfoResult` and `ZendService\Technorati\KeyInfoResult` represent exceptions to the above because they don't belong to a result set and they don't implement any interface. They represent a single response object and they act as a wrapper for additional `ZendService\Technorati` objects, such as `ZendService\Technorati\Author` and `ZendService\Technorati\Weblog`.

The `ZendService\Technorati` library includes additional convenient classes representing specific response objects. `ZendService\Technorati\Author` represents a single Technorati account, also known as a blog author

or blogger. `ZendService\Technorati\Weblog` represents a single weblog object, along with all specific weblog properties such as feed *URL*s or blog name. For full details please see `ZendService\Technorati` in the API reference guide.

317.8.1 ZendServiceTechnoratiResultSet

`ZendService\Technorati\ResultSet` is the most essential result set. The scope of this class is to be extended by a query-specific child result set class, and it should never be used to initialize a standalone object. Each of the specific result sets represents a collection of query-specific *ZendServiceTechnoratiResult* objects.

`ZendService\Technorati\ResultSet` implements the *PHP* `SeekableIterator` interface, and you can iterate all result objects via the *PHP* `foreach()` statement.

Iterating result objects from a resultset collection

```
// run a simple query
$technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
$resultSet = $technorati->search('php');

// $resultSet is now an instance of
// ZendService\Technorati\SearchResultSet
// it extends ZendService\Technorati\ResultSet
foreach ($resultSet as $result) {
    // do something with your
    // ZendService\Technorati\SearchResult object
}
```

317.8.2 ZendServiceTechnoratiCosmosResultSet

`ZendService\Technorati\CosmosResultSet` represents a Technorati Cosmos query result set.

Note: `ZendService\Technorati\CosmosResultSet` extends *ZendServiceTechnoratiResultSet*.

317.8.3 ZendServiceTechnoratiSearchResultSet

`ZendService\Technorati\SearchResultSet` represents a Technorati Search query result set.

Note: `ZendService\Technorati\SearchResultSet` extends *ZendServiceTechnoratiResultSet*.

317.8.4 ZendServiceTechnoratiTagResultSet

`ZendService\Technorati\TagResultSet` represents a Technorati Tag query result set.

Note: `ZendService\Technorati\TagResultSet` extends *ZendServiceTechnoratiResultSet*.

317.8.5 ZendServiceTechnoratiDailyCountsResultSet

`ZendService\Technorati\DailyCountsResultSet` represents a Technorati DailyCounts query result set.

Note: `ZendService\Technorati\DailyCountsResultSet` extends *ZendServiceTechnoratiResultSet*.

317.8.6 ZendServiceTechnoratiTagsResultSet

`ZendService\Technorati\TagsResultSet` represents a Technorati TopTags or BlogPostTags queries result set.

Note: `ZendService\Technorati\TagsResultSet` extends *ZendServiceTechnoratiResultSet*.

317.8.7 ZendServiceTechnoratiResult

`ZendService\Technorati\Result` is the most essential result object. The scope of this class is to be extended by a query specific child result class, and it should never be used to initialize a standalone object.

317.8.8 ZendServiceTechnoratiCosmosResult

`ZendService\Technorati\CosmosResult` represents a single Technorati Cosmos query result object. It is never returned as a standalone object, but it always belongs to a valid *ZendServiceTechnoratiCosmosResultSet* object.

Note: `ZendService\Technorati\CosmosResult` extends *ZendServiceTechnoratiResult*.

317.8.9 ZendServiceTechnoratiSearchResult

`ZendService\Technorati\SearchResult` represents a single Technorati Search query result object. It is never returned as a standalone object, but it always belongs to a valid *ZendServiceTechnoratiSearchResultSet* object.

Note: `ZendService\Technorati\SearchResult` extends *ZendServiceTechnoratiResult*.

317.8.10 ZendServiceTechnoratiTagResult

`ZendService\Technorati\TagResult` represents a single Technorati Tag query result object. It is never returned as a standalone object, but it always belongs to a valid *ZendServiceTechnoratiTagResultSet* object.

Note: `ZendService\Technorati\TagResult` extends *ZendServiceTechnoratiResult*.

317.8.11 ZendServiceTechnoratiDailyCountsResult

`ZendService\Technorati\DailyCountsResult` represents a single Technorati DailyCounts query result object. It is never returned as a standalone object, but it always belongs to a valid *ZendServiceTechnoratiDailyCountsResultSet* object.

Note: `ZendService\Technorati\DailyCountsResult` extends *ZendServiceTechnoratiResult*.

317.8.12 ZendServiceTechnoratiTagsResult

`ZendService\Technorati\TagsResult` represents a single Technorati TopTags or BlogPostTags query result object. It is never returned as a standalone object, but it always belongs to a valid *ZendServiceTechnoratiTagsResultSet* object.

Note: `ZendService\Technorati\TagsResult` extends *ZendServiceTechnoratiResult*.

317.8.13 ZendServiceTechnoratiGetInfoResult

`ZendService\Technorati\GetInfoResult` represents a single Technorati GetInfo query result object.

317.8.14 ZendServiceTechnoratiBlogInfoResult

`ZendService\Technorati\BlogInfoResult` represents a single Technorati BlogInfo query result object.

317.8.15 ZendServiceTechnoratiKeyInfoResult

`ZendService\Technorati\KeyInfoResult` represents a single Technorati KeyInfo query result object. It provides information about your *Technorati API Key daily usage*.

CHAPTER

EIGHT

ZENDSERVICETWITTER

318.1 Introduction

`ZendService\Twitter` provides a client for the Twitter API. `ZendService\Twitter` allows you to query the public timeline. If you provide a username and OAuth details for Twitter, or your access token and secret, it will allow you to get and update your status, reply to friends, direct message friends, mark tweets as favorites, and much more.

`ZendService\Twitter` wraps all web service operations, including OAuth, and all methods return an instance of `ZendService\Twitter\Response`.

`ZendService\Twitter` is broken up into subsections so you can easily identify which type of call is being requested.

- *account* allows you to check that your account credentials are valid
- *application* allows you to check your API rate limits.
- *blocks* blocks and unblocks users from following you.
- *directMessages* retrieves the authenticated user's received direct messages, deletes direct messages, and sends new direct messages.
- *favorites* lists, creates, and removes favorite tweets.
- *friendships* creates and removes friendships for the authenticated user.
- *search* allows you to search statuses for specific criteria.
- *statuses* retrieves the public and user timelines and shows, updates, destroys, and retrieves replies for the authenticated user.
- *users* retrieves friends and followers for the authenticated user and returns extended information about a passed user.

318.2 Authentication

With the exception of fetching the public timeline, `ZendService\Twitter` requires authentication as a valid user. This is achieved using the OAuth authentication protocol. OAuth is the only supported authentication mode for Twitter as of August 2010. The OAuth implementation used by `ZendService\Twitter` is `ZendOAuth`.

Creating the Twitter Class

`ZendService\Twitter` must authorize itself, on behalf of a user, before use with the Twitter API (except for public timeline). This must be accomplished using OAuth since Twitter has disabled it's basic HTTP authentication as of August 2010.

There are two options to establishing authorization. The first is to implement the workflow of `ZendOAuth` via `ZendService\Twitter` which proxies to an internal `ZendOAuth\Consumer` object. Please refer to the `ZendOAuth` documentation for a full example of this workflow - you can call all documented `ZendOAuth\Consumer` methods on `ZendService\Twitter` including constructor options. You may also use `ZendOAuth` directly and only pass the resulting access token into `ZendService\Twitter`. This is the normal workflow once you have established a reusable access token for a particular Twitter user. The resulting OAuth access token should be stored to a database for future use (otherwise you will need to authorize for every new instance of `ZendService\Twitter`). Bear in mind that authorization via OAuth results in your user being redirected to Twitter to give their consent to the requested authorization (this is not repeated for stored access tokens). This will require additional work (i.e. redirecting users and hosting a callback URL) over the previous HTTP authentication mechanism where a user just needed to allow applications to store their username and password.

The following example demonstrates setting up `ZendService\Twitter` which is given an already established OAuth access token. Please refer to the `ZendOAuth` documentation to understand the workflow involved. The access token is a serializable object, so you may store the serialized object to a database, and unserialize it at retrieval time before passing the objects into `ZendService\Twitter`. The `ZendOAuth` documentation demonstrates the workflow and objects involved.

```
/**
 * We assume $serializedToken is the serialized token retrieved from a database
 * or even $_SESSION (if following the simple ZendOAuth documented example)
 */
$token = unserialize($serializedToken);

$twitter = new ZendService\Twitter\Twitter(array(
    'accessToken' => $token,
    'oauth_options' => array(
        'username' => 'johndoe',
    ),
));

// verify user's credentials with Twitter
$response = $twitter->account->verifyCredentials();
```

Note: In order to authenticate with Twitter, ALL applications MUST be registered with Twitter in order to receive a Consumer Key and Consumer Secret to be used when authenticating with OAuth. This can not be reused across multiple applications - you must register each new application separately. Twitter access tokens have no expiry date, so storing them to a database is advised (they can, of course, be refreshed simply be repeating the OAuth authorization process). This can only be done while interacting with the user associated with that access token.

The previous pre-OAuth version of `ZendService\Twitter` allowed passing in a username as the first parameter rather than within an array. This is no longer supported.

If you have registered an application with Twitter, you can also use the access token and access token secret they provide you in order to setup the OAuth consumer. This can be done as follows:

```
$twitter = new Zend_Service_Twitter(array(
    'access_token' => array( // or use "accessToken" as the key; both work
        'token' => 'your-access-token',
        'secret' => 'your-access-token-secret',
```

```
    ),
    'oauth_options' => array( // or use "oauthOptions" as the key; both work
        'consumerKey' => 'your-consumer-key',
        'consumerSecret' => 'your-consumer-secret',
    ),
));
```

If desired, you can also specify a specific HTTP client instance to use, or provide configuration for the HTTP client. To provide the HTTP client, use the `http_client` or `httpClient` key, and provide an instance. To provide HTTP client configuration for setting up an instance, use the key `http_client_options` or `httpClientOptions`. As a full example:

```
$twitter = new Zend_Service_Twitter(array(
    'access_token' => array( // or use "accessToken" as the key; both work
        'token' => 'your-access-token',
        'secret' => 'your-access-token-secret',
    ),
    'oauth_options' => array( // or use "oauthOptions" as the key; both work
        'consumerKey' => 'your-consumer-key',
        'consumerSecret' => 'your-consumer-secret',
    ),
    'http_client_options' => array(
        'adapter' => 'Zend_Http_Client_Adapter_Curl',
    ),
));
```

318.3 Account Methods

- `verifyCredentials()` tests if supplied user credentials are valid with minimal overhead.

Verifying credentials

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->account->verifyCredentials();
```

318.4 Application Methods

- `rateLimitStatus()` returns the remaining number of *API* requests available to the authenticating user before the *API* limit is reached for the current hour.

Rating limit status

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->application->rateLimitStatus();
$userTimelineLimit = $response->resources->statuses->{'/statuses/user_timeline'}->remaining;
```

318.5 Blocking Methods

- `create()` blocks the user specified in the *id* parameter as the authenticating user and destroys a friendship to the blocked user if one exists. Returns the blocked user in the requested format when successful.

Blocking a user

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->blocks->create('usertoblock');
```

- `destroy()` un-blocks the user specified in the *id* parameter for the authenticating user. Returns the un-blocked user in the requested format when successful.

Removing a block

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->blocks->destroy('blockeduser');
```

- `ids()` returns an array of user identifiers that the authenticating user is blocking.

Who are you blocking (identifiers only)

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->blocks->ids();
```

- `list()` returns an array of user objects that the authenticating user is blocking.

Who are you blocking

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->blocks->list();
```

318.6 Direct Message Methods

- `messages()` returns a list of the 20 most recent direct messages sent to the authenticating user.

Retrieving recent direct messages received

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->directMessages->messages();
```

The `message()` method accepts an array of optional parameters to modify the query.

  - *since_id* narrows the returned results to just those statuses after the specified identifier (up to 24 hours old).
  - *max_id* narrows the returned results to just those statuses earlier than the specified identifier.
  - *count* specifies the number of statuses to return, up to 200.

  - *skip_status*, when set to boolean true, "t", or 1 will skip including a user's most recent status in the results.
  - *include_entities* controls whether or not entities, which includes URLs, mentioned users, and hashtags, will be returned.

- `sent()` returns a list of the 20 most recent direct messages sent by the authenticating user.

Retrieving recent direct messages sent

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->directMessages->sent();
```

The `sent()` method accepts an array of optional parameters to modify the query.

  - *count* specifies the number of statuses to return, up to 20.
  - *page* specifies the page of results to return, based on the *count* provided.
  - *since_id* narrows the returned results to just those statuses after the specified identifier (up to 24 hours old).
  - *max_id* narrows the returned results to just those statuses earlier than the specified identifier.
  - *include_entities* controls whether or not entities, which includes URLs, mentioned users, and hashtags, will be returned.

- `new()` sends a new direct message to the specified user from the authenticating user. Requires both the user and text parameters below.

Sending a direct message

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->directMessages->new('myfriend', 'mymessage');
```

- `destroy()` destroys the direct message specified in the required *id* parameter. The authenticating user must be the recipient of the specified direct message.

Deleting a direct message

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->directMessages->destroy(123548);
```

318.7 Favorites Methods

- `list()` returns the 20 most recent favorite statuses for the authenticating user or user specified by the *id* parameter.

Retrieving favorites

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->favorites->list();
```

The `list()` method accepts an array of optional parameters to modify the query.

  - *user_id* specifies the ID of the user for whom to return the timeline.
  - *screen_name* specifies the screen name of the user for whom to return the timeline.
  - *since_id* narrows the returned results to just those statuses after the specified identifier (up to 24 hours old).
  - *max_id* narrows the returned results to just those statuses earlier than the specified identifier.
  - *count* specifies the number of statuses to return, up to 200.
  - *include_entities* controls whether or not entities, which includes URLs, mentioned users, and hashtags, will be returned.

- `create()` favorites the status specified in the *id* parameter as the authenticating user.

Creating favorites

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->favorites->create(12351);
```

- `destroy()` un-favorites the status specified in the *id* parameter as the authenticating user.

Deleting a favorite

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->favorites->destroy(12351);
```

318.8 Friendship Methods

- `create()` befriends the user specified in the *id* parameter with the authenticating user.

Creating a friend

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->friendships->create('mynewfriend');
```

- `destroy()` discontinues friendship with the user specified in the *id* parameter and the authenticating user.

Deleting a friend

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->friendships->destroy('myoldfriend');
```

318.9 Search Methods

- `tweets()` returns a list of tweets matching the criteria specified in *$query*. By default, 15 will be returned, but this value may be changed using the *count* option.

Searching for tweets

```
$twitter  = new Zend_Service_Twitter($options);
$response = $twitter->search->tweets('#zendframework');
```

The `tweets()` method accepts an optional second argument, array of optional parameters to modify the query.

- *since_id* narrows the returned results to just those statuses after the specified identifier (up to 24 hours old).
- *max_id* narrows the returned results to just those statuses earlier than the specified identifier.
- *count* specifies the number of statuses to return, up to 200.
- *include_entities* controls whether or not entities, which includes URLs, mentioned users, and hashtags, will be returned.
- *lang* indicates which two-letter language code to restrict results to.
- *locale* indicates which two-letter language code is being used in the query.
- *geocode* can be used to indicate the geographical radius in which tweets should originate; the string should be in the form "latitude,longitude,radius", with "radius" being a unit followed by one of "mi" or "km".
- *result_type* indicates what type of results to retrieve, and should be one of "mixed," "recent," or "popular."
- *until* can be used to specify a the latest date for which to return tweets.

318.10 Status Methods

- `sample()` returns the 20 most recent statuses from non-protected users with a custom user icon. The public timeline is cached by Twitter for 60 seconds.

Retrieving the public timeline

```
$twitter  = new ZendService\Twitter($options);
$response = $twitter->statuses->sample();
```

- `homeTimeline()` returns the 20 most recent statuses posted by the authenticating user and that user's friends.

Retrieving the home timeline

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->statuses->homeTimeline();
```

The `homeTimeline()` method accepts an array of optional parameters to modify the query.

- *since_id* narrows the returned results to just those statuses after the specified identifier (up to 24 hours old).
- *max_id* narrows the returned results to just those statuses earlier than the specified identifier.
- *count* specifies the number of statuses to return, up to 200.
- *trim_user*, when set to boolean true, "t", or 1, will list the author identifier only in embedded user objects in the statuses returned.
- *contributor_details*, when set to boolean true, will return the screen name of any contributors to a status (instead of only the contributor identifier).

  - *include_entities* controls whether or not entities, which includes URLs, mentioned users, and hashtags, will be returned.
  - *exclude_replies* controls whether or not status updates that are in reply to other statuses will be returned.
- `userTimeline()` returns the 20 most recent statuses posted from the authenticating user.

Retrieving the user timeline

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->statuses->userTimeline();
```

The `userTimeline()` method accepts an array of optional parameters to modify the query.

  - *user_id* specifies the ID of the user for whom to return the timeline.
  - *screen_name* specifies the screen name of the user for whom to return the timeline.
  - *since_id* narrows the returned results to just those statuses after the specified identifier (up to 24 hours old).
  - *max_id* narrows the returned results to just those statuses earlier than the specified identifier.
  - *count* specifies the number of statuses to return, up to 200.
  - *trim_user*, when set to boolean true, "t", or 1, will list the author identifier only in embedded user objects in the statuses returned.
  - *contributor_details*, when set to boolean true, will return the screen name of any contributors to a status (instead of only the contributor identifier).
  - *include_rts* controls whether or not to include native retweets in the returned list.
  - *exclude_replies* controls whether or not status updates that are in reply to other statuses will be returned.
- `show()` returns a single status, specified by the *id* parameter below. The status' author will be returned inline.

Showing user status

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->statuses->show(1234);
```

- `update()` updates the authenticating user's status. This method requires that you pass in the status update that you want to post to Twitter.

Updating user status

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->statuses->update('My Great Tweet');
```

The `update()` method accepts a second additional parameter.

  - *inReplyTo_StatusId* specifies the ID of an existing status that the status to be posted is in reply to.
- `mentionsTimeline()` returns the 20 most recent @replies (status updates prefixed with @username) for the authenticating user.

Showing user replies

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->statuses->mentionsTimeline();
```

The `mentionsTimeline()` method accepts an array of optional parameters to modify the query.

- *since_id* narrows the returned results to just those statuses after the specified identifier (up to 24 hours old).
- *max_id* narrows the returned results to just those statuses earlier than the specified identifier.
- *count* specifies the number of statuses to return, up to 200.
- *trim_user*, when set to boolean true, "t", or 1, will list the author identifier only in embedded user objects in the statuses returned.
- *contributor_details*, when set to boolean true, will return the screen name of any contributors to a status (instead of only the contributor identifier).
- *include_entities* controls whether or not entities, which includes URLs, mentioned users, and hashtags, will be returned.

- `destroy()` destroys the status specified by the required *id* parameter.

Deleting user status

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->statuses->destroy(12345);
```

318.11 User Methods

- `show()` returns extended information of a given user, specified by ID or screen name as per the required *id* parameter below.

Showing user information

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->users->show('myfriend');
```

- `search()` will search for users matching the query provided.

Searching for users

```
$twitter  = new ZendService\Twitter\Twitter($options);
$response = $twitter->users->search('Zend');
```

The `search()` method accepts an array of optional parameters to modify the query.

- *count* specifies the number of statuses to return, up to 20.
- *page* specifies the page of results to return, based on the *count* provided.
- *include_entities* controls whether or not entities, which includes URLs, mentioned users, and hashtags, will be returned.

CHAPTER

NINE

ZENDSERVICEWINDOWSAZURE

319.1 Introduction

Windows Azure is the name for Microsoft's Software + Services platform, an operating system in the cloud providing services for hosting, management, scalable storage with support for simple blobs, tables, and queues, as well as a management infrastructure for provisioning and geo-distribution of cloud-based services, and a development platform for the Azure Services layer.

319.2 Installing the Windows Azure SDK

There are two development scenario's when working with Windows Azure.

- You can develop your application using `ZendService\WindowsAzure` and the Windows Azure *SDK*, which provides a local development environment of the services provided by Windows Azure's cloud infrastructure.
- You can develop your application using `ZendService\WindowsAzure`, working directly with the Windows Azure cloud infrastructure.

The first case requires you to install the Windows Azure SDK on your development machine. It is currently only available for Windows environments; progress is being made on a Java-based version of the *SDK* which can run on any platform.

The latter case requires you to have an account at Azure.com.

319.3 API Documentation

The `ZendService\WindowsAzure` class provides the *PHP* wrapper to the Windows Azure *REST* interface. Please consult the REST documentation for detailed description of the service. You will need to be familiar with basic concepts in order to use this service.

319.4 Features

`ZendService\WindowsAzure` provides the following functionality:

- *PHP* classes for Windows Azure Blobs, Tables and Queues (for *CRUD* operations)
- Helper Classes for *HTTP* transport, AuthN, AuthZ, *REST* and Error Management

- Manageability, Instrumentation and Logging support

319.5 Architecture

`ZendService\WindowsAzure` provides access to Windows Azure's storage, computation and management interfaces by abstracting the *REST-XML* interface Windows Azure provides into a simple *PHP API*.

An application built using `ZendService\WindowsAzure` can access Windows Azure's features, no matter if it is hosted on the Windows Azure platform or on an in-premise web server.

CHAPTER

ZENDSERVICEWINDOWSAZURESTORAGEBL

Blob Storage stores sets of binary data. Blob storage offers the following three resources: the storage account, containers, and blobs. Within your storage account, containers provide a way to organize sets of blobs within your storage account.

Blob Storage is offered by Windows Azure as a *REST API* which is wrapped by the `ZendService\WindowsAzure\Storage\Blob` class in order to provide a native *PHP* interface to the storage account.

320.1 API Examples

This topic lists some examples of using the `ZendService\WindowsAzure\Storage\Blob` class. Other features are available in the download package, as well as a detailed *API* documentation of those features.

320.1.1 Creating a storage container

Using the following code, a blob storage container can be created on development storage.

Creating a storage container

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob();
$result = $storageClient->createContainer('testcontainer');

echo 'Container name is: ' . $result->Name;
```

320.1.2 Deleting a storage container

Using the following code, a blob storage container can be removed from development storage.

Deleting a storage container

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob();
$storageClient->deleteContainer('testcontainer');
```

320.1.3 Storing a blob

Using the following code, a blob can be uploaded to a blob storage container on development storage. Note that the container has already been created before.

Storing a blob

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob();

// upload /home/maarten/example.txt to Azure
$result = $storageClient->putBlob(
    'testcontainer', 'example.txt', '/home/maarten/example.txt'
);

echo 'Blob name is: ' . $result->Name;
```

320.1.4 Copying a blob

Using the following code, a blob can be copied from inside the storage account. The advantage of using this method is that the copy operation occurs in the Azure cloud and does not involve downloading the blob. Note that the container has already been created before.

Copying a blob

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob();

// copy example.txt to example2.txt
$result = $storageClient->copyBlob(
    'testcontainer', 'example.txt', 'testcontainer', 'example2.txt'
);

echo 'Copied blob name is: ' . $result->Name;
```

320.1.5 Downloading a blob

Using the following code, a blob can be downloaded from a blob storage container on development storage. Note that the container has already been created before and a blob has been uploaded.

Downloading a blob

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob();

// download file to /home/maarten/example.txt
$storageClient->getBlob(
    'testcontainer', 'example.txt', '/home/maarten/example.txt'
);
```

320.1.6 Making a blob publicly available

By default, blob storage containers on Windows Azure are protected from public viewing. If any user on the Internet should have access to a blob container, its ACL can be set to public. Note that this applies to a complete container and not to a single blob!

Using the following code, blob storage container ACL can be set on development storage. Note that the container has already been created before.

Making a blob publicly available

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob();

// make container publicly available
$storageClient->setContainerAcl(
    'testcontainer',
    ZendService\WindowsAzure\Storage\Blob::ACL_PUBLIC
);
```

320.2 Root container

Windows Azure Blob Storage provides support to work with a "root container". This means that a blob can be stored in the root of your storage account, i.e. `http://myaccount.blob.core.windows.net/somefile.txt`.

In order to work with the root container, it should first be created using the `createContainer()` method, naming the container `$root`. All other operations on the root container should be issued with the container name set to `$root`.

320.3 Blob storage stream wrapper

The Windows Azure *SDK* for *PHP* provides support for registering a blob storage client as a *PHP* file stream wrapper. The blob storage stream wrapper provides support for using regular file operations on Windows Azure Blob Storage. For example, one can open a file from Windows Azure Blob Storage with the `fopen()` function:

Example usage of blob storage stream wrapper

```
$fileHandle = fopen('azure://mycontainer/myfile.txt', 'r');

// ...

fclose($fileHandle);
```

In order to do this, the Windows Azure *SDK* for *PHP* blob storage client must be registered as a stream wrapper. This can be done by calling the `registerStreamWrapper()` method:

Registering the blob storage stream wrapper

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob();
// registers azure:// on this storage client
$storageClient->registerStreamWrapper();


// or:

// registers blob:// on this storage client
$storageClient->registerStreamWrapper('blob://');
```

To unregister the stream wrapper, the `unregisterStreamWrapper()` method can be used.

320.4 Shared Access Signature

Windows Azure Bob Storage provides a feature called "Shared Access Signatures". By default, there is only one level of authorization possible in Windows Azure Blob Storage: either a container is private or it is public. Shared Access Signatures provide a more granular method of authorization: read, write, delete and list permissions can be assigned on a container or a blob and given to a specific client using an URL-based model.

An example would be the following signature:

```
http://phpstorage.blob.core.windows.net/phpazuretestshared1?st=2009-08-17T09%3A06%3A17Z&se=2009-08-1
```

The above signature gives write access to the "phpazuretestshared1" container of the "phpstorage" account.

320.4.1 Generating a Shared Access Signature

When you are the owner of a Windows Azure Bob Storage account, you can create and distribute a shared access key for any type of resource in your account. To do this, the `generateSharedAccessUrl()` method of the `ZendService\WindowsAzure\Storage\Blob` storage client can be used.

The following example code will generate a Shared Access Signature for write access in a container named "container1", within a timeframe of 3000 seconds.

Generating a Shared Access Signature for a container

```
$storageClient   = new ZendService\WindowsAzure\Storage\Blob();
$sharedAccessUrl = $storageClient->generateSharedAccessUrl(
    'container1',
    '',
    'c',
    'w',
    $storageClient ->isoDate(time() - 500),
    $storageClient ->isoDate(time() + 3000)
);
```

The following example code will generate a Shared Access Signature for read access in a blob named `test.txt` in a container named "container1" within a time frame of 3000 seconds.

Generating a Shared Access Signature for a blob

```
$storageClient   = new ZendService\WindowsAzure\Storage\Blob();
$sharedAccessUrl = $storageClient->generateSharedAccessUrl(
    'container1',
    'test.txt',
    'b',
    'r',
    $storageClient ->isoDate(time() - 500),
    $storageClient ->isoDate(time() + 3000)
);
```

320.4.2 Working with Shared Access Signatures from others

When you receive a Shared Access Signature from someone else, you can use the Windows Azure *SDK* for *PHP* to work with the addressed resource. For example, the following signature can be retrieved from the owner of a storage account:

```
http://phpstorage.blob.core.windows.net/phpazuretestshared1?st=2009-08-17T09%3A06%3A17Z&se=2009-08-1
```

The above signature gives write access to the "phpazuretestshared1" "container" of the phpstorage account. Since the shared key for the account is not known, the Shared Access Signature can be used to work with the authorized resource.

Consuming a Shared Access Signature for a container

```
$storageClient = new ZendService\WindowsAzure\Storage\Blob(
    'blob.core.windows.net', 'phpstorage', ''
);
$storageClient->setCredentials(
    new ZendService\WindowsAzure\Credentials\SharedAccessSignature()
);
$storageClient->getCredentials()->setPermissionSet(array(
    'http://phpstorage.blob.core.windows.net/phpazuretestshared1?st=2009-08-17T09%3A06%3A17Z&se=2009-
));
$storageClient->putBlob(
    'phpazuretestshared1', 'NewBlob.txt', 'C:\Files\dataforazure.txt'
);
```

Note that there was no explicit permission to write to a specific blob. Instead, the Windows Azure *SDK* for *PHP* determined that a permission was required to either write to that specific blob, or to write to its container. Since only a signature was available for the latter, the Windows Azure *SDK* for *PHP* chose those credentials to perform the request on Windows Azure blob storage.

CHAPTER

ONE

ZENDSERVICEWINDOWSAZURESTORAGETAI

The Table service offers structured storage in the form of tables.

Table Storage is offered by Windows Azure as a REST *API* which is wrapped by the `ZendService\WindowsAzure\Storage\Table` class in order to provide a native *PHP* interface to the storage account.

This topic lists some examples of using the `ZendService\WindowsAzure\Storage\Table` class. Other features are available in the download package, as well as a detailed *API* documentation of those features.

Note that development table storage (in the Windows Azure *SDK*) does not support all features provided by the *API*. Therefore, the examples listed on this page are to be used on Windows Azure production table storage.

321.1 Operations on tables

This topic lists some samples of operations that can be executed on tables.

321.1.1 Creating a table

Using the following code, a table can be created on Windows Azure production table storage.

Creating a table

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$result = $storageClient->createTable('testtable');

echo 'New table name is: ' . $result->Name;
```

321.1.2 Listing all tables

Using the following code, a list of all tables in Windows Azure production table storage can be queried.

Listing all tables

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$result = $storageClient->listTables();
foreach ($result as $table) {
    echo 'Table name is: ' . $table->Name . "\r\n";
}
```

321.2 Operations on entities

Tables store data as collections of entities. Entities are similar to rows. An entity has a primary key and a set of properties. A property is a named, typed-value pair, similar to a column.

The Table service does not enforce any schema for tables, so two entities in the same table may have different sets of properties. Developers may choose to enforce a schema on the client side. A table may contain any number of entities.

`ZendService\WindowsAzure\Storage\Table` provides 2 ways of working with entities:

- Enforced schema
- No enforced schema

All examples will make use of the following enforced schema class.

Enforced schema used in samples

```
class SampleEntity extends ZendService\WindowsAzure\Storage\TableEntity
{
    /**
    * @azure Name
    */
    public $Name;

    /**
    * @azure Age Edm.Int64
    */
    public $Age;

    /**
    * @azure Visible Edm.Boolean
    */
    public $Visible = false;
}
```

Note that if no schema class is passed into table storage methods, `ZendService\WindowsAzure\Storage\Table` automatically works with `ZendService\WindowsAzure\Storage\DynamicTableEntity`.

321.2.1 Enforced schema entities

To enforce a schema on the client side using the `ZendService\WindowsAzure\Storage\Table` class, you can create a class which inherits `ZendService\WindowsAzure\Storage\TableEntity`. This class pro-

vides some basic functionality for the `ZendService\WindowsAzure\Storage\Table` class to work with a client-side schema.

Base properties provided by `ZendService\WindowsAzure\Storage\TableEntity` are:

- PartitionKey (exposed through `getPartitionKey()` and `setPartitionKey()`)
- RowKey (exposed through `getRowKey()` and `setRowKey()`)
- Timestamp (exposed through `getTimestamp()` and `setTimestamp()`)
- Etag value (exposed through `getEtag()` and `setEtag()`)

Here's a sample class inheriting `ZendService\WindowsAzure\Storage\TableEntity`:

Sample enforced schema class

```
class SampleEntity extends ZendService\WindowsAzure\Storage\TableEntity
{
    /**
     * @azure Name
     */
    public $Name;

    /**
     * @azure Age Edm.Int64
     */
    public $Age;

    /**
     * @azure Visible Edm.Boolean
     */
    public $Visible = false;
}
```

The `ZendService\WindowsAzure\Storage\Table` class will map any class inherited from `ZendService\WindowsAzure\Storage\TableEntity` to Windows Azure table storage entities with the correct data type and property name. All there is to storing a property in Windows Azure is adding a docblock comment to a public property or public getter/setter, in the following format:

Enforced property

```
/**
 * @azure <property name in Windows Azure> <optional property type>
 */
public $<property name in PHP>;
```

Let's see how to define a property "Age" as an integer on Windows Azure table storage:

Sample enforced property

```
/**
 * @azure Age Edm.Int64
 */
public $Age;
```

Note that a property does not necessarily have to be named the same on Windows Azure table storage. The Windows Azure table storage property name can be defined as well as the type.

The following data types are supported:

- `Edm.Binary`- An array of bytes up to 64 KB in size.
- `Edm.Boolean`- A boolean value.
- `Edm.DateTime`- A 64-bit value expressed as Coordinated Universal Time (UTC). The supported DateTime range begins from 12:00 midnight, January 1, 1601 A.D. (C.E.), Coordinated Universal Time (UTC). The range ends at December 31st, 9999.
- `Edm.Double`- A 64-bit floating point value.
- `Edm.Guid`- A 128-bit globally unique identifier.
- `Edm.Int32`- A 32-bit integer.
- `Edm.Int64`- A 64-bit integer.
- `Edm.String`- A UTF-16-encoded value. String values may be up to 64 KB in size.

321.2.2 No enforced schema entities (a.k.a. DynamicEntity)

To use the `ZendService\WindowsAzure\Storage\Table` class without defining a schema, you can make use of the `ZendService\WindowsAzure\Storage\DynamicTableEntity` class. This class inherits `ZendService\WindowsAzure\Storage\TableEntity` like an enforced schema class does, but contains additional logic to make it dynamic and not bound to a schema.

Base properties provided by `ZendService\WindowsAzure\Storage\DynamicTableEntity` are:

- PartitionKey (exposed through `getPartitionKey()` and `setPartitionKey()`)
- RowKey (exposed through `getRowKey()` and `setRowKey()`)
- Timestamp (exposed through `getTimestamp()` and `setTimestamp()`)
- Etag value (exposed through `getEtag()` and `setEtag()`)

Other properties can be added on the fly. Their Windows Azure table storage type will be determined on-the-fly:

Dynamically adding properties ZendServiceWindowsAzureStorageDynamicTableEntity

```
$target = new ZendService\WindowsAzure\Storage\DynamicTableEntity(
    'partition1', '000001'
);
$target->Name = 'Name'; // Will add property "Name" of type "Edm.String"
$target->Age  = 25;     // Will add property "Age" of type "Edm.Int32"
```

Optionally, a property type can be enforced:

Forcing property types on ZendServiceWindowsAzureStorageDynamicTableEntity

```
$target = new ZendService\WindowsAzure\Storage\DynamicTableEntity(
    'partition1', '000001'
);
$target->Name = 'Name'; // Will add property "Name" of type "Edm.String"
$target->Age  = 25;     // Will add property "Age" of type "Edm.Int32"
```

```

// Change type of property "Age" to "Edm.Int32":
$target->setAzurePropertyType('Age', 'Edm.Int64');
```

The `ZendService\WindowsAzure\Storage\Table` class automatically works with `ZendService\WindowsAzure\Storage\TableEntity` if no specific class is passed into Table Storage methods.

321.2.3 Entities API examples

321.2.4 Inserting an entity

Using the following code, an entity can be inserted into a table named "testtable". Note that the table has already been created before.

Inserting an entity

```
$entity = new SampleEntity ('partition1', 'row1');
$entity->FullName = "Maarten";
$entity->Age = 25;
$entity->Visible = true;

$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$result = $storageClient->insertEntity('testtable', $entity);

// Check the timestamp and etag of the newly inserted entity
echo 'Timestamp: ' . $result->getTimestamp() . "\n";
echo 'Etag: ' . $result->getEtag() . "\n";
```

321.2.5 Retrieving an entity by partition key and row key

Using the following code, an entity can be retrieved by partition key and row key. Note that the table and entity have already been created before.

Retrieving an entity by partition key and row key

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity= $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);
```

321.2.6 Updating an entity

Using the following code, an entity can be updated. Note that the table and entity have already been created before.

Updating an entity

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity = $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);

$entity->Name = 'New name';
$result = $storageClient->updateEntity('testtable', $entity);
```

If you want to make sure the entity has not been updated before, you can make sure the *Etag* of the entity is checked. If the entity already has had an update, the update will fail to make sure you do not overwrite any newer data.

Updating an entity (with Etag check)

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity = $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);

$entity->Name = 'New name';

// last parameter instructs the Etag check:
$result = $storageClient->updateEntity('testtable', $entity, true);
```

321.2.7 Deleting an entity

Using the following code, an entity can be deleted. Note that the table and entity have already been created before.

Deleting an entity

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity = $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);
$result = $storageClient->deleteEntity('testtable', $entity);
```

321.2.8 Performing queries

Queries in `ZendService\WindowsAzure\Storage\Table` table storage can be performed in two ways:

- By manually creating a filter condition (involving learning a new query language)
- By using the fluent interface provided by the `ZendService\WindowsAzure\Storage\Table`

Using the following code, a table can be queried using a filter condition. Note that the table and entities have already been created before.

Performing queries using a filter condition

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entities = $storageClient->storageClient->retrieveEntities(
    'testtable',
    'Name eq \'Maarten\' and PartitionKey eq \'partition1\'',
    'SampleEntity'
);

foreach ($entities as $entity) {
    echo 'Name: ' . $entity->Name . "\n";
}
```

Using the following code, a table can be queried using a fluent interface. Note that the table and entities have already been created before.

Performing queries using a fluent interface

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entities = $storageClient->storageClient->retrieveEntities(
    'testtable',
    $storageClient->select()
                  ->from($tableName)
                  ->where('Name eq ?', 'Maarten')
                  ->andWhere('PartitionKey eq ?', 'partition1'),
    'SampleEntity'
);

foreach ($entities as $entity) {
    echo 'Name: ' . $entity->Name . "\n";
}
```

321.2.9 Batch operations

This topic demonstrates how to use the table entity group transaction features provided by Windows Azure table storage. Windows Azure table storage supports batch transactions on entities that are in the same table and belong to the same partition group. A transaction can include at most 100 entities.

The following example uses a batch operation (transaction) to insert a set of entities into the "testtable" table. Note that the table has already been created before.

Executing a batch operation

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);

// Start batch
$batch = $storageClient->startBatch();
```

```

// Insert entities in batch
$entities = generateEntities();
foreach ($entities as $entity) {
    $storageClient->insertEntity($tableName, $entity);
}

// Commit
$batch->commit();
```

321.3 Table storage session handler

When running a *PHP* application on the Windows Azure platform in a load-balanced mode (running 2 Web Role instances or more), it is important that *PHP* session data can be shared between multiple Web Role instances. The Windows Azure *SDK* for *PHP* provides the `ZendService\WindowsAzure\SessionHandler` class, which uses Windows Azure Table Storage as a session handler for *PHP* applications.

To use the `ZendService\WindowsAzure\SessionHandler` session handler, it should be registered as the default session handler for your *PHP* application:

Registering table storage session handler

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);

$sessionHandler = new ZendService\WindowsAzure\SessionHandler(
    $storageClient , 'sessionstable'
);
$sessionHandler->register();
```

The above classname registers the `ZendService\WindowsAzure\SessionHandler` session handler and will store sessions in a table called "sessionstable".

After registration of the `ZendService\WindowsAzure\SessionHandler` session handler, sessions can be started and used in the same way as a normal *PHP* session:

Using table storage session handler

```
$storageClient = new ZendService\WindowsAzure\Storage\Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);

$sessionHandler = new ZendService\WindowsAzure\SessionHandler(
    $storageClient , 'sessionstable'
);
$sessionHandler->register();

session_start();

if (!isset($_SESSION['firstVisit'])) {
    $_SESSION['firstVisit'] = time();
}
```

```
15
16 // ...
```

Warning: The `ZendService\WindowsAzure\SessionHandler` session handler should be registered before a call to `session_start()` is made!

CHAPTER

TWO

ZENDSERVICEWINDOWSAZURESTORAGEQU

The Queue service stores messages that may be read by any client who has access to the storage account.

A queue can contain an unlimited number of messages, each of which can be up to 8 KB in size. Messages are generally added to the end of the queue and retrieved from the front of the queue, although first in/first out (*FIFO*) behavior is not guaranteed. If you need to store messages larger than 8 KB, you can store message data as a queue or in a table and then store a reference to the data as a message in a queue.

Queue Storage is offered by Windows Azure as a *REST API* which is wrapped by the `ZendService\WindowsAzure\Storage\Queue` class in order to provide a native *PHP* interface to the storage account.

322.1 API Examples

This topic lists some examples of using the `ZendService\WindowsAzure\Storage\Queue` class. Other features are available in the download package, as well as a detailed *API* documentation of those features.

322.1.1 Creating a queue

Using the following code, a queue can be created on development storage.

Creating a queue

```
$storageClient = new ZendService\WindowsAzure\Storage\Queue();
$result = $storageClient->createQueue('testqueue');

echo 'Queue name is: ' . $result->Name;
```

322.1.2 Deleting a queue

Using the following code, a queue can be removed from development storage.

Deleting a queue

```
$storageClient = new ZendService\WindowsAzure\Storage\Queue();
$storageClient->deleteQueue('testqueue');
```

322.1.3 Adding a message to a queue

Using the following code, a message can be added to a queue on development storage. Note that the queue has already been created before.

Adding a message to a queue

```
$storageClient = new ZendService\WindowsAzure\Storage\Queue();

// 3600 = time-to-live of the message, if omitted defaults to 7 days
$storageClient->putMessage('testqueue', 'This is a test message', 3600);
```

322.1.4 Reading a message from a queue

Using the following code, a message can be read from a queue on development storage. Note that the queue and message have already been created before.

Reading a message from a queue

```
$storageClient = new ZendService\WindowsAzure\Storage\Queue();

// retrieve 10 messages at once
$messages = $storageClient->getMessages('testqueue', 10);

foreach ($messages as $message) {
    echo $message->MessageText . "\r\n";
}
```

The messages that are read using `getMessages()` will be invisible in the queue for 30 seconds, after which the messages will re-appear in the queue. To mark a message as processed and remove it from the queue, use the `deleteMessage()` method.

Marking a message as processed

```
$storageClient = new ZendService\WindowsAzure\Storage\Queue();

// retrieve 10 messages at once
$messages = $storageClient->getMessages('testqueue', 10);

foreach ($messages as $message) {
    echo $message . "\r\n";

    // Mark the message as processed
    $storageClient->deleteMessage('testqueue', $message);
}
```

322.1.5 Check if there are messages in a queue

Using the following code, a queue can be checked for new messages. Note that the queue and message have already been created before.

Check if there are messages in a queue

```
$storageClient = new ZendService\WindowsAzure\Storage\Queue();

// retrieve 10 messages at once
$messages = $storageClient->peekMessages('testqueue', 10);

foreach ($messages as $message) {
    echo $message->MessageText . "\r\n";
}
```

Note that messages that are read using `peekMessages()` will not become invisible in the queue, nor can they be marked as processed using the `deleteMessage()` method. To do this, use `getMessages()` instead.

CHAPTER

THREE

COPYRIGHT INFORMATION

The following copyrights are applicable to portions of Zend Framework.

Copyright © 2005-Zend Technologies Inc. (http://www.zend.com)

CHAPTER

FOUR

INTRODUCTION TO ZEND FRAMEWORK 2

- *Overview*
- *Installation*

CHAPTER

FIVE

USER GUIDE

The user guide is provided to take you through a non-trivial example, showing you various techniques and features of the framework in order to build an application.

- *Getting Started with Zend Framework 2*
- *Getting started: A skeleton application*
- *Modules*
- *Routing and controllers*
- *Database and models*
- *Styling and Translations*
- *Forms and actions*
- *Conclusion*

CHAPTER

SIX

ZEND FRAMEWORK TOOL (ZFTOOL)

- *Zend Framework Tool (ZFTool)*

CHAPTER

SEVEN

LEARNING ZEND FRAMEWORK 2

- *Learning Dependency Injection*
- *Unit Testing a Zend Framework 2 application*
- *Using the EventManager*
- *Using Zend\Navigation in your Album Module*
- *Using Zend\Paginator in your Album Module*
- *Setting up a database adapter*

CHAPTER

EIGHT

|MIGRATION|

- *Migration from Zend Framework 1*
- *Namespacing Old Classes*
- *Running Zend Framework 2 and Zend Framework 1 in parallel*

CHAPTER

NINE

ZEND FRAMEWORK 2 REFERENCE

329.1 Zend\Authentication

- *Introduction to Zend\Authentication*
- *Database Table Authentication*
- *Digest Authentication*
- *HTTP Authentication Adapter*
- *LDAP Authentication*
- *Authentication Validator*

329.2 Zend\Barcode

- *Introduction to Zend\Barcode*
- *Barcode creation using Zend\Barcode\Barcode class*
- *Zend\Barcode\Barcode Objects*
- *Zend\Barcode Renderers*

329.3 Zend\Cache

- *Zend\Cache\Storage\Adapter*
- *Zend\Cache\Storage\Capabilities*
- *Zend\Cache\Storage\Plugin*
- *Zend\Cache\Pattern*
- *Zend\Cache\Pattern\CallbackCache*
- *Zend\Cache\Pattern\ClassCache*
- *Zend\Cache\Pattern\ObjectCache*
- *Zend\Cache\Pattern\OutputCache*
- *Zend\Cache\Pattern\CaptureCache*

329.4 Zend\Captcha

- *Introduction to Zend\Captcha*
- *Captcha Operation*
- *CAPTCHA Adapters*

329.5 Zend\Config

- *Introduction to Zend\Config*
- *Theory of Operation*
- *Zend\Config\Reader*
- *Zend\Config\Writer*
- *Zend\Config\Processor*
- *The Factory*

329.6 Zend\Console

- *Introduction to Zend\Console*
- *Console routes and routing*
- *Console-aware modules*
- *Console-aware action controllers*
- *Console adapters*
- *Console prompts*

329.7 Zend\Crypt

- *Introduction to Zend\Crypt*
- *Encrypt/decrypt using block ciphers*
- *Key derivation function*
- *Password*
- *Public key cryptography*

329.8 Zend\Db

- *Zend\Db\Adapter*
- *Zend\Db\ResultSet*
- *Zend\Db\Sql*

- *Zend\Db\TableGateway*
- *Zend\Db\RowGateway*
- *Zend\Db\Metadata*

329.9 Zend\Di

- *Introduction to Zend\Di*
- *Zend\Di Quickstart*
- *Zend\Di Definition*
- *Zend\Di InstanceManager*
- *Zend\Di Configuration*
- *Zend\Di Debugging & Complex Use Cases*

329.10 Zend\Dom

- *Introduction to Zend\Dom*
- *Zend\Dom\Query*

329.11 Zend\Escaper

- *Introduction to Zend\Escaper*
- *Theory of Operation*
- *Configuring Zend\Escaper*
- *Escaping HTML*
- *Escaping HTML Attributes*
- *Escaping Javascript*
- *Escaping Cascading Style Sheets*
- *Escaping URLs*

329.12 Zend\EventManager

- *The EventManager*

329.13 Zend\Feed

- *Introduction to Zend\Feed*
- *Importing Feeds*

- *Retrieving Feeds from Web Pages*
- *Consuming an RSS Feed*
- *Consuming an Atom Feed*
- *Consuming a Single Atom Entry*
- *Zend\Feed and Security*
- *Zend\Feed\Reader\Reader*
- *Zend\Feed\Writer\Writer*
- *Zend\Feed\PubSubHubbub*

329.14 Zend\File

- *Zend\File\ClassFileLocator*

329.15 Zend\Filter

- *Introduction to Zend\Filter*
- *Standard Filter Classes*
- *Word Filters*
- *File Filter Classes*
- *Filter Chains*
- *Zend\Filter\Inflector*
- *Writing Filters*

329.16 Zend\Form

- *Introduction to Zend\Form*
- *Form Quick Start*
- *Form Collections*
- *File Uploading*
- *Advanced use of forms*
- *Form Elements*
- *Form View Helpers*

329.17 Zend\Http

- *Overview of Zend\Http*
- *The Request Class*

- *The Response Class*
- *The Headers Class*
- *HTTP Client - Overview*
- *HTTP Client - Connection Adapters*
- *HTTP Client - Advanced Usage*
- *HTTP Client - Static Usage*

329.18 Zend\I18n

- *Translating*
- *I18n View Helpers*
- *I18n Filters*
- *I18n Validators*

329.19 Zend\InputFilter

- *Introduction to Zend\InputFilter*
- *File Upload Input*

329.20 Zend\Json

- *Introduction to Zend\Json*
- *Basic Usage*
- *Advanced Usage of Zend\Json*
- *XML to JSON conversion*
- *Zend\Json\Server - JSON-RPC server*

329.21 Zend\Ldap

- *Introduction to Zend\Ldap*
- *API overview*
- *Usage Scenarios*
- *Tools*
- *Object-oriented access to the LDAP tree using Zend\Ldap\Node*
- *Getting information from the LDAP server*
- *Serializing LDAP data to and from LDIF*

329.22 Zend\Loader

- *The AutoloaderFactory*
- *The StandardAutoloader*
- *The ClassMapAutoloader*
- *The ModuleAutoloader*
- *The SplAutoloader Interface*
- *The PluginClassLoader*
- *The ShortNameLocator Interface*
- *The PluginClassLocator interface*
- *The Class Map Generator utility: bin/classmap_generator.php*

329.23 Zend\Log

- *Overview of Zend\Log*
- *Writers*
- *Filters*
- *Formatters*

329.24 Zend\Mail

- *Introduction to Zend\Mail*
- *Zend\Mail\Message*
- *Zend\Mail\Transport*
- *Zend\Mail\Transport\SmtpOptions*
- *Zend\Mail\Transport\FileOptions*

329.25 Zend\Math

- *Introduction to Zend\Math*

329.26 Zend\Mime

- *Zend\Mime*
- *Zend\Mime\Message*
- *Zend\Mime\Part*

329.27 Zend\ModuleManager

- *Introduction to the Module System*
- *The Module Manager*
- *The Module Class*
- *The Module Autoloader*
- *Best Practices when Creating Modules*

329.28 Zend\Mvc

- *Introduction to the MVC Layer*
- *Quick Start*
- *Default Services*
- *Routing*
- *The MvcEvent*
- *The SendResponseEvent*
- *Available Controllers*
- *Controller Plugins*
- *Examples*

329.29 Zend\Navigation

- *Introduction to Zend\Navigation*
- *Quick Start*
- *Pages*
- *Containers*
- *View Helpers*
- *View Helper - Breadcrumbs*
- *View Helper - Links*
- *View Helper - Menu*
- *View Helper - Sitemap*
- *View Helper - Navigation Proxy*

329.30 Zend\Paginator

- *Introduction to Zend\Paginator*
- *Usage*

- *Configuration*
- *Advanced usage*

329.31 Zend\Permissions\Acl

- *Introduction to Zend\Permissions\Acl*
- *Refining Access Controls*
- *Advanced Usage*

329.32 Zend\Permissions\Rbac

- *Introduction to Zend\Permissions\Rbac*
- *Methods*
- *Examples*

329.33 Zend\ProgressBar

- *Progress Bars*
- *File Upload Handlers*

329.34 Zend\Serializer

- *Introduction to Zend\Serializer*
- *Zend\Serializer\Adapter*

329.35 Zend\Server

- *Introduction to Zend\Server*
- *Zend\Server\Reflection*

329.36 Zend\ServiceManager

- *Zend\ServiceManager*
- *Zend\ServiceManager Quick Start*

329.37 Zend\Session

- *Session Config*
- *Session Container*
- *Session Manager*
- *Session Save Handlers*
- *Session Storage*
- *Session Validators*

329.38 Zend\Soap

- *Zend\Soap\Server*
- *Zend\Soap\Client*
- *WSDL Accessor*
- *AutoDiscovery*

329.39 Zend\Stdlib

- *Zend\Stdlib\Hydrator*
- *Zend\Stdlib\Hydrator\Filter*
- *Zend\Stdlib\Hydrator\Strategy*

329.40 Zend\Tag

- *Introduction to Zend\Tag*
- *Creating tag clouds with Zend\Tag\Cloud*

329.41 Zend\Test

- *Introduction to Zend\Test*
- *Unit testing with PHPUnit*

329.42 Zend\Text

- *Zend\Text\Figlet*
- *Zend\Text\Table*

329.43 Zend\Uri

- *Zend\Uri*

329.44 Zend\Validator

- *Introduction to Zend\Validator*
- *Standard Validation Classes*
- *File Validation Classes*
- *Validator Chains*
- *Writing Validators*
- *Validation Messages*

329.45 Zend\Version

- *Getting the Zend Framework Version*

329.46 Zend\View

- *Zend\View Quick Start*
- *The PhpRenderer*
- *PhpRenderer View Scripts*
- *The ViewEvent*
- *View Helpers*
- *Advanced usage of helpers*

329.47 Zend\XmlRpc

- *Introduction to Zend\XmlRpc*
- *Zend\XmlRpc\Client*
- *Zend\XmlRpc\Server*

CHAPTER

SERVICES FOR ZEND FRAMEWORK 2 REFERENCE

330.1 ZendService\Akismet

- *ZendServiceAkismet*

330.2 ZendService\Amazon

- *ZendServiceAmazon*

330.3 ZendService\AppleApns

- *ZendService\Apple\Apns*

330.4 ZendService\Audioscrobbler

- *ZendServiceAudioscrobbler*

330.5 ZendService\Del.icio.us

- *ZendServiceDelicious*

330.6 ZendService\Developer Garden

- *Zend_Service_DeveloperGarden*

330.7 ZendService\Flickr

- *ZendServiceFlickr*

330.8 ZendService\Google\Gcm

- `modules/zendservice.google.gcm`

330.9 ZendService\LiveDocx

- *ZendService\LiveDocx*

330.10 ZendService\Nirvanix

- *ZendServiceNirvanix*

330.11 ZendService\Rackspace

- *Zend\Service\Rackspace*

330.12 ZendService\ReCaptcha

- *ZendServiceReCaptcha*

330.13 ZendService\SlideShare

- *ZendServiceSlideShare*

330.14 ZendService\StrikeIron

- *ZendServiceStrikeIron*

330.15 ZendService\Technorati

- *ZendServiceTechnorati*

330.16 ZendService\Twitter

- *ZendServiceTwitter*

330.17 ZendService\Windows Azure

- *ZendServiceWindowsAzure*

CHAPTER

ONE

COPYRIGHT

- *Copyright Information*

CHAPTER

TWO

INDICES AND TABLES

- *Programmer's Reference Guide of Zend Framework 2*
- *search*