
Zend Framework 2 Documentation

Release 2.4.13dev

Zend Technologies Ltd.

Aug 02, 2017

Contents

1	Overview	1
2	Installation	3
2.1	Using Composer	3
2.2	Using Git submodules	3
2.3	Web Server Setup	4
3	Getting Started with Zend Framework 2	7
3.1	Some assumptions	7
3.2	The tutorial application	7
4	Getting started: A skeleton application	9
4.1	Using the Apache Web Server	10
4.2	Using the Built-in PHP CLI Server	13
4.3	Error reporting	13
5	Modules	15
5.1	Setting up the Album module	15
5.2	Configuration	17
5.3	Informing the application about our new module	17
6	Routing and controllers	19
7	Create the controller	21
7.1	Initialise the view scripts	22
8	Database and models	23
8.1	The database	23
8.2	The model files	24
8.3	Using ServiceManager to configure the table gateway and inject into the AlbumTable	25
8.4	Back to the controller	27
8.5	Listing albums	27
9	Styling and Translations	31
10	Forms and actions	33
10.1	Adding new albums	33
10.2	Editing an album	38

10.3	Deleting an album	41
10.4	Ensuring that the home page displays the list of albums	42
11	Conclusion	45
12	Introducing our first “Blog” Module	47
12.1	Writing a new Module	47
12.2	Configuring the Module	48
13	Introducing Services and the ServiceManager	53
13.1	What is a Service?	53
13.2	Writing the PostService	53
13.3	Writing the required Model Files	55
13.4	Bringing Life into our PostService	57
13.5	Bringing the Service into the Controller	59
13.6	Writing a Factory Class	61
13.7	Registering Services	62
13.8	Using the Service at our Controller	62
13.9	Accessing View Variables	63
13.10	Summary	64
14	Preparing for different Database-Backends	65
14.1	What is database abstraction?	65
14.2	Creating the PostMapperInterface	65
14.3	Refactoring the PostService	66
14.4	The PostService has a dependency	68
14.5	Conclusion	69
15	Introducing Zend\Db\Sql and Zend\Stdlib\Hydrator	71
15.1	Preparing the Database	71
15.2	Quick Facts Zend\Db\Sql	72
15.3	Writing the mapper implementation	73
15.4	Refactoring hidden dependencies	80
15.5	Finishing the mapper	82
15.6	Conclusion	83
16	Understanding the Router	85
16.1	Different route types	85
16.2	Different routing concepts	87
16.3	A practical example for our Blog Module	90
17	Making use of Forms and Fieldsets	95
17.1	Core components	95
17.2	Creating your first Fieldset	96
17.3	Creating the PostForm	96
17.4	Adding a new Post	97
17.5	Displaying the form	101
17.6	Controller Logic for basically all Forms	103
17.7	Zend\Form and Zend\Stdlib\Hydrator working together	109
17.8	Conclusion	113
18	Editing and Deleting Data	115
18.1	Binding Objects to Forms	115
18.2	Adding the edit-route	117
18.3	Creating the edit-template	118

18.4	Implementing the delete functionality	119
18.5	Summary	127
19	Reviewing the Blog-application	129
19.1	Do we always need all the layers and interfaces?	129
19.2	Having many objects, won't there be much code-duplication?	129
19.3	Why are there so many controllers?	130
19.4	Do you have more questions? PR them!	130
20	Getting Started with Zend Framework 2	131
20.1	Installation	131
20.2	Getting Started	132
21	A quick tour of the skeleton application	137
21.1	The dispatch cycle	138
22	The MyTaskList application	141
22.1	The Checklist module	141
22.2	The Module class	143
23	The application's pages	145
23.1	Routing	145
23.2	The TaskController	147
23.3	The model	147
23.4	Using Service Manager to configure the database credentials and inject into the controller	150
24	Listing tasks	153
24.1	Redirect the home page	155
25	Styling	157
26	Adding new tasks	159
27	Editing a task	165
28	Deleting a task	169
29	Application Diagnostics	171
30	Step-by-step debugging	173
31	Conclusion	175
32	Zend Framework Tool (ZFTool)	177
32.1	Installation using Composer	177
32.2	Manual installation	177
32.3	Without installation, using the PHAR file	178
32.4	Usage	178
33	Learning Dependency Injection	181
33.1	Very brief introduction to Di.	181
33.2	Simplest usage case (2 classes, one consumes the other)	181
33.3	Simplest Usage Case Without Type-hints	183
33.4	Simplest usage case with Compiled Definition	185
33.5	Creating a precompiled definition for others to use	186
33.6	Using Multiple Definitions From Multiple Sources	186

33.7	Generating Service Locators	187
34	Unit Testing a Zend Framework 2 application	191
34.1	Setting up phpunit to use composer's autoload.php	191
34.2	Setting up the tests directory	191
34.3	Bootstrapping your tests	192
34.4	Your first controller test	194
34.5	A failing test case	195
34.6	Configuring the service manager for the tests	196
34.7	Testing actions with POST	197
34.8	Testing model entities	198
34.9	Testing model tables	201
34.10	Conclusion	204
35	Using the EventManager	205
35.1	Terminology	205
35.2	Getting started	205
35.3	Shared managers	207
36	Wildcards	209
37	Listener aggregates	211
37.1	Introspecting results	212
37.2	Short-circuiting listener execution	212
37.3	Keeping it in order	213
37.4	Custom event objects	214
37.5	Putting it together: Implementing a simple caching system	215
37.6	Conclusion	217
38	Advanced Configuration Tricks	219
38.1	System configuration	219
38.2	Module Configuration	223
38.3	Configuration mapping table	224
38.4	Configuration Priority	224
38.5	Manipulating merged configuration	224
38.6	Configuration merging workflow	225
39	Using Zend\Navigation in your Album Module	227
39.1	Preparation	227
39.2	Setting Up Zend\Navigation	228
39.3	Configuring our Site Map	228
39.4	Adding the Menu View Helper	229
39.5	Adding Breadcrumbs	229
40	Using Zend\Paginator in your Album Module	231
40.1	Preparation	231
40.2	Modifying the AlbumTable	234
40.3	Modifying the AlbumController	235
40.4	Updating the View Script	236
40.5	Creating the Pagination Control Partial	236
41	Using the PaginationControl View Helper	239
42	Setting up a database adapter	241
42.1	Introduction	241

42.2	Basic setup	241
42.3	Setting a static adapter	242
43	Migration from Zend Framework 1	243
44	Namespacing Old Classes	245
44.1	Namespacing a ZF1 Application	246
44.2	HOWTO Namespace Your Models	247
45	Running Zend Framework 2 and Zend Framework 1 in parallel	249
45.1	Use ZF2 in a ZF1 project	249
45.2	Use ZF1 in a ZF2 project	250
45.3	Run ZF1 and ZF2 together	250
46	Introduction to Zend\Authentication	253
46.1	Adapters	253
46.2	Results	254
46.3	Identity Persistence	255
46.4	Usage	258
47	Database Table Authentication	261
47.1	Introduction	261
47.2	Basic Usage	262
47.3	Advanced Usage: Persisting a DbTable Result Object	264
48	Digest Authentication	267
48.1	Introduction	267
48.2	Specifics	267
48.3	Identity	268
49	HTTP Authentication Adapter	269
49.1	Introduction	269
49.2	Design Overview	269
49.3	Configuration Options	270
49.4	Resolvers	270
49.5	Basic Usage	271
50	LDAP Authentication	273
50.1	Introduction	273
50.2	Usage	273
50.3	The API	275
50.4	Server Options	276
50.5	Collecting Debugging Messages	278
50.6	Common Options for Specific Servers	278
51	Authentication Validator	281
51.1	Introduction	281
51.2	Basic Usage	281
52	Introduction to Zend\Barcode	283
52.1	Overview	283
53	Barcode creation using Zend\Barcode\Barcode class	285
53.1	Using Zend\Barcode\Barcode::factory	285
53.2	Drawing a barcode	286
53.3	Rendering a barcode	287

54	Zend\Barcode Objects	289
54.1	Common Options	290
54.2	Common Additional Getters	291
54.3	Description of shipped barcodes	291
55	Zend\Barcode Renderers	299
55.1	Common Options	299
55.2	Zend\Barcode\Renderer\Image	300
55.3	Zend\Barcode\Renderer\Pdf	300
56	Zend\Cache\Storage\Adapter	301
56.1	Overview	301
56.2	Quick Start	301
56.3	Basic Configuration Options	302
56.4	The StorageInterface	302
56.5	The AvailableSpaceCapableInterface	304
56.6	The TotalSpaceCapableInterface	304
56.7	The ClearByNamespaceInterface	305
56.8	The ClearByPrefixInterface	305
56.9	The ClearExpiredInterface	305
56.10	The FlushableInterface	305
56.11	The IterableInterface	305
56.12	The OptimizableInterface	306
56.13	The TaggableInterface	306
56.14	The Apc Adapter	306
56.15	The Dba Adapter	307
56.16	The Filesystem Adapter	308
56.17	The Memcached Adapter	309
56.18	The Redis Adapter	310
56.19	The Memory Adapter	312
56.20	The MongoDB Adapter	313
56.21	The WinCache Adapter	314
56.22	The XCache Adapter	315
56.23	The ZendServerDisk Adapter	316
56.24	The ZendServerShm Adapter	317
56.25	Examples	317
57	Zend\Cache\Storage\Capabilities	321
57.1	Overview	321
57.2	Available Methods	321
57.3	Examples	323
58	Zend\Cache\Storage\Plugin	325
58.1	Overview	325
58.2	Quick Start	325
58.3	The ClearExpiredByFactor Plugin	326
58.4	The ExceptionHandler Plugin	326
58.5	The IgnoreUserAbort Plugin	326
58.6	The OptimizeByFactor Plugin	327
58.7	The Serializer Plugin	327
58.8	Available Methods	327
58.9	Examples	328
59	Zend\Cache\Pattern	331
59.1	Overview	331

59.2	Quick Start	331
59.3	Available Methods	332
60	Zend\Cache\Pattern\CallbackCache	333
60.1	Overview	333
60.2	Quick Start	333
60.3	Configuration Options	334
60.4	Available Methods	334
60.5	Examples	334
61	Zend\Cache\Pattern\ClassCache	335
61.1	Overview	335
61.2	Quick Start	335
61.3	Configuration Options	336
61.4	Available Methods	336
61.5	Examples	337
62	Zend\Cache\Pattern\ObjectCache	339
62.1	Overview	339
62.2	Quick Start	339
62.3	Configuration Options	340
62.4	Available Methods	340
62.5	Examples	341
63	Zend\Cache\Pattern\OutputCache	343
63.1	Overview	343
63.2	Quick Start	343
63.3	Configuration Options	343
63.4	Available Methods	343
63.5	Examples	344
64	Zend\Cache\Pattern\CaptureCache	345
64.1	Overview	345
64.2	Quick Start	345
64.3	Configuration Options	346
64.4	Available Methods	346
64.5	Examples	347
65	Introduction to Zend\Captcha	349
65.1	Overview	349
66	Captcha Operation	351
66.1	The AdapterInterface	351
66.2	Basic Usage	351
67	CAPTCHA Adapters	353
67.1	Zend\Captcha\AbstractWord	353
67.2	Zend\Captcha\Dumb	354
67.3	Zend\Captcha\Figlet	354
67.4	Zend\Captcha\Image	354
67.5	Zend\Captcha\ReCaptcha	355
68	Introduction	357
68.1	Theory of Operation	357

69	Zend\Code\Generator Reference	361
69.1	Abstract Classes and Interfaces	361
69.2	Concrete CodeGenerator Classes	362
70	Zend\Code\Generator Examples	369
70.1	Generating PHP classes	369
70.2	Generating PHP files	373
70.3	Add code to existing PHP files and classes	374
71	Introduction to Zend\Config	377
71.1	Using Zend\Config\Config with a Reader Class	377
71.2	Using Zend\Config\Config with a PHP Configuration File	378
72	Theory of Operation	379
73	Zend\Config\Reader	381
73.1	Zend\Config\Reader\Ini	381
73.2	Zend\Config\Reader\Xml	383
73.3	Zend\Config\Reader\Json	384
73.4	Zend\Config\Reader\Yaml	385
73.5	Zend\Config\Reader\JavaProperties	386
74	Zend\Config\Writer	387
74.1	Zend\Config\Writer\Ini	387
74.2	Zend\Config\Writer\Xml	388
74.3	Zend\Config\Writer\PhpArray	389
74.4	Zend\Config\Writer\Json	390
74.5	Zend\Config\Writer\Yaml	390
75	Zend\Config\Processor	393
75.1	Zend\Config\Processor\Constant	393
75.2	Zend\Config\Processor\Filter	394
75.3	Zend\Config\Processor\Queue	394
75.4	Zend\Config\Processor\Token	394
75.5	Zend\Config\Processor\Translator	395
76	The Factory	397
76.1	Loading configuration file	397
76.2	Storing configuration file	397
77	Introduction to Zend\Console	399
77.1	Writing console routes	399
77.2	Handling console requests	401
77.3	Adding console usage info	402
78	Console routes and routing	405
78.1	Router configuration	405
78.2	Basic route	406
78.3	Catchall route	411
78.4	Console routes cheat-sheet	412
79	Console-aware modules	413
79.1	Application banner	413
79.2	Basic usage	415
79.3	Best practices	422

80 Console-aware action controllers	423
80.1 Handling console requests	423
80.2 Sending output to console	425
80.3 Are we in a console?	426
80.4 Reading values from console parameters	427
81 Console adapters	431
81.1 Retrieving console adapter	431
81.2 Using console adapter	432
82 Console prompts	435
82.1 Confirm	436
82.2 Line	436
82.3 Char	437
82.4 Select	438
82.5 Password	439
83 Zend\Console\Getopt	441
83.1 Introduction	441
84 Declaring Getopt Rules	443
84.1 Declaring Options with the Short Syntax	443
84.2 Declaring Options with the Long Syntax	444
85 Fetching Options and Arguments	445
85.1 Handling Getopt Exceptions	445
85.2 Fetching Options by Name	446
85.3 Reporting Options	446
85.4 Fetching Non-option Arguments	447
86 Configuring Zend\Console\Getopt	449
86.1 Adding Option Rules	449
86.2 Adding Help Messages	449
86.3 Adding Option Aliases	450
86.4 Adding Argument Lists	450
86.5 Adding Configuration	451
87 Introduction to Zend\Crypt	453
88 Encrypt/decrypt using block ciphers	455
89 Encrypt/decrypt a file	457
90 Key derivation function	459
90.1 Pbkdf2 adapter	459
90.2 SaltedS2k adapter	460
90.3 Scrypt adapter	460
91 Password	463
91.1 Bcrypt	463
91.2 Apache	465
92 Public key cryptography	467
92.1 Diffie-Hellman	467
92.2 RSA	470

93 Zend\Db\Adapter	473
93.1 Creating an Adapter - Quickstart	473
93.2 Creating an Adapter Using Dependency Injection	474
93.3 Query Preparation Through Zend\Db\Adapter\Adapter::query()	475
93.4 Query Execution Through Zend\Db\Adapter\Adapter::query()	475
93.5 Creating Statements	475
93.6 Using the Driver Object	475
93.7 Using The Platform Object	477
93.8 Using The Parameter Container	478
93.9 Examples	479
94 Zend\Db\ResultSet	481
94.1 Quickstart	481
94.2 Zend\Db\ResultSet\ResultSet and Zend\Db\ResultSet\AbstractResultSet	482
94.3 Zend\Db\ResultSet\HydratingResultSet	482
95 Zend\Db\Sql	485
95.1 Zend\Db\Sql\Sql (Quickstart)	485
95.2 Zend\Db\Sql's Select, Insert, Update and Delete	486
95.3 Zend\Db\Sql\Select	486
95.4 Zend\Db\Sql\Insert	489
95.5 Zend\Db\Sql\Update	490
95.6 Zend\Db\Sql\Delete	490
95.7 Zend\Db\Sql\Where & Zend\Db\Sql\Having	491
96 Zend\Db\Sql\Ddl	497
96.1 Creating Tables	497
96.2 Altering Tables	498
96.3 Dropping Tables	498
96.4 Executing DDL Statements	498
96.5 Currently Supported Data Types	499
96.6 Currently Supported Constraint Types	499
97 Zend\Db\TableGateway	501
97.1 Basic Usage	501
97.2 TableGateway Features	503
98 Zend\Db\RowGateway	505
98.1 Quickstart	505
98.2 ActiveRecord Style Objects	506
99 Zend\Db\Metadata	507
99.1 Basic Usage	508
100 Dumping Variables	513
100.1 Example of dump() method	513
101 Introduction to Zend\Di	515
101.1 Dependency Injection	515
101.2 Dependency Injection Containers	516
102 Zend\Di Quickstart	517
103 Zend\Di Definition	521
103.1 DefinitionList	521
103.2 RuntimeDefinition	521

103.3 CompilerDefinition	522
103.4 ClassDefinition	523
104Zend\Di InstanceManager	525
104.1 Parameters	525
104.2 Preferences	526
104.3 Aliases	527
105Zend\Di Configuration	529
106Zend\Di Debugging & Complex Use Cases	531
106.1 Debugging a DiC	531
106.2 Complex Use Cases	531
107Introduction to Zend\Dom	535
108Zend\Dom\Query	537
108.1 Theory of Operation	537
108.2 Methods Available	538
109Introduction to Zend\Escaper	541
109.1 Overview	541
109.2 What Zend\Escaper is not	542
110Theory of Operation	543
110.1 The Problem with Inconsistent Functionality	544
110.2 Why Contextual Escaping?	544
111Configuring Zend\Escaper	547
112Escaping HTML	549
112.1 Examples of Bad HTML Escaping	549
112.2 Examples of Good HTML Escaping	550
113Escaping HTML Attributes	551
113.1 Examples of Bad HTML Attribute Escaping	551
113.2 Examples of Good HTML Attribute Escaping	552
114Escaping Javascript	555
114.1 Examples of Bad Javascript Escaping	555
114.2 Examples of Good Javascript Escaping	556
115Escaping Cascading Style Sheets	557
115.1 Examples of Bad CSS Escaping	557
115.2 Examples of Good CSS Escaping	558
116Escaping URLs	559
116.1 Examples of Bad URL Escaping	559
116.2 Examples of Good URL Escaping	559
117The EventManager	561
117.1 Overview	561
117.2 Quick Start	561
117.3 Configuration Options	565
117.4 Available Methods	565
117.5 Examples	566

118	Introduction to Zend\Feed	571
118.1	Reading RSS Feed Data	571
119	Importing Feeds	573
119.1	Dumping the contents of a feed	573
120	Retrieving Feeds from Web Pages	575
120.1	Find Feed Links	575
121	Consuming an RSS Feed	577
121.1	Reading a feed	577
121.2	Get properties	577
122	Consuming an Atom Feed	579
122.1	Basic Use of an Atom Feed	579
123	Consuming a Single Atom Entry	581
123.1	Reading a Single-Entry Atom Feed	581
124	Zend\Feed and Security	583
124.1	Introduction	583
124.2	Filtering data using HTMLPurifier	583
124.3	Escaping data using Zend\Escaper	585
125	Zend\Feed\Reader\Reader	587
125.1	Introduction	587
125.2	Importing Feeds	587
125.3	Retrieving Underlying Feed and Entry Sources	588
125.4	Cache Support and Intelligent Requests	589
125.5	Locating Feed URIs from Websites	590
125.6	Attribute Collections	591
125.7	Retrieving Feed Information	592
125.8	Retrieving Entry/Item Information	594
125.9	Extending Feed and Entry APIs	596
126	Zend\Feed\Writer\Writer	601
126.1	Introduction	601
126.2	Architecture	601
126.3	Getting Started	602
126.4	Setting Feed Data Points	604
126.5	Setting Entry Data Points	606
127	Zend\Feed\PubSubHubbub	609
127.1	What is PubSubHubbub?	609
127.2	Architecture	610
127.3	Zend\Feed\PubSubHubbub\Publisher	610
127.4	Zend\Feed\PubSubHubbub\Subscriber	611
128	Zend\File\ClassFileLocator	619
128.1	Overview	619
128.2	Available Methods	619
128.3	Examples	619
129	Introduction to Zend\Filter	621
129.1	What is a filter?	621
129.2	Basic usage of filters	621

129.3 Using the StaticFilter	622
129.4 Double filtering	622
130Standard Filter Classes	625
130.1 Alnum	625
130.2 Alpha	626
130.3 BaseName	627
130.4 Blacklist	627
130.5 Boolean	628
130.6 Callback	630
130.7 Compress and Decompress	631
130.8 Digits	637
130.9 Dir	637
130.10Encrypt and Decrypt	638
130.11HtmlEntities	644
130.12ToInt	645
130.13ToNull	646
130.14NumberFormat	648
130.15NumberParse	648
130.16PregReplace	649
130.17RealPath	650
130.18StringToLower	651
130.19StringToUpper	652
130.20StringTrim	652
130.21StripNewlines	653
130.22StripTags	653
130.23UriNormalize	655
130.24Whitelist	655
131Word Filters	657
131.1 CamelCaseToDash	657
131.2 CamelCaseToSeparator	657
131.3 CamelCaseToUnderscore	658
131.4 DashToCamelCase	658
131.5 DashToSeparator	659
131.6 DashToUnderscore	659
131.7 SeparatorToCamelCase	660
131.8 SeparatorToDash	660
131.9 SeparatorToSeparator	661
131.10UnderscoreToCamelCase	662
131.11UnderscoreToSeparator	662
131.12UnderscoreToDash	663
132File Filter Classes	665
132.1 Decrypt	665
132.2 Encrypt	665
132.3 Lowercase	665
132.4 Rename	665
132.5 RenameUpload	667
132.6 Uppercase	669
133Filter Chains	671
133.1 Setting Filter Chain Order	671
133.2 Using the Plugin Manager	672

134	Zend\Filter\Inflector	673
134.1	Transforming MixedCase and camelCaseText to another format	673
134.2	Static Rules	675
134.3	Filter Inflector Rules	676
134.4	Setting Many Rules At Once	677
134.5	Example	678
135	Using the StaticFilter	679
136	Writing Filters	681
136.1	Example	681
137	Introduction	683
138	Quick Start	685
138.1	Programmatic Form Creation	685
138.2	Creation via Factory	687
138.3	Factory-backed Form Extension	690
138.4	Validating Forms	691
138.5	Hinting to the Input Filter	692
138.6	Binding an object	694
138.7	Rendering	695
138.8	Validation Groups	698
138.9	Using Annotations	699
139	Form Collections	703
139.1	Creating Fieldsets	706
139.2	The Form Element	710
139.3	The Controller	711
139.4	The View	712
139.5	Adding New Elements Dynamically	713
139.6	Validation groups for fieldsets and collection	716
140	File Uploading	719
140.1	Standard Example	719
140.2	File Post-Redirect-Get Plugin	722
140.3	HTML5 Multi-File Uploads	724
140.4	Upload Progress	725
140.5	Additional Info	730
141	Advanced use of forms	731
141.1	Short names	731
141.2	Creating custom elements	731
141.3	Handling dependencies	735
141.4	The specific case of initializers	737
142	Form Elements	739
142.1	Introduction	739
142.2	Element Base Class	739
142.3	Standard Elements	741
142.4	HTML5 Elements	758
143	Form View Helpers	773
143.1	Introduction	773
143.2	Standard Helpers	773

143.3 HTML5 Helpers	788
143.4 File Upload Progress Helpers	794
144Zend\Http	797
144.1 Overview	797
144.2 Zend\Http Request, Response and Headers	797
145The Request Class	799
145.1 Overview	799
145.2 Quick Start	799
145.3 Configuration Options	800
145.4 Available Methods	800
145.5 Examples	803
146The Response Class	805
146.1 Overview	805
146.2 Quick Start	805
146.3 Configuration Options	806
146.4 Available Methods	806
146.5 Examples	809
147The Headers Class	811
147.1 Overview	811
147.2 Quick Start	811
147.3 Configuration Options	812
147.4 Available Methods	812
147.5 Zend\Http\Header\HeaderInterface Methods	814
147.6 Zend\Http\Header\AbstractAccept Methods	814
147.7 Zend\Http\Header\AbstractDate Methods	815
147.8 Zend\Http\Header\AbstractLocation Methods	816
147.9 List of HTTP Header Types	816
147.10Examples	821
148HTTP Client	825
148.1 Overview	825
148.2 Quick Start	825
148.3 Configuration	826
148.4 Examples	827
149HTTP Client - Connection Adapters	831
149.1 Overview	831
149.2 The Socket Adapter	831
149.3 The Proxy Adapter	834
149.4 The cURL Adapter	835
149.5 The Test Adapter	836
149.6 Creating your own connection adapters	838
150HTTP Client - Advanced Usage	841
150.1 HTTP Redirections	841
150.2 Adding Cookies and Using Cookie Persistence	841
150.3 Setting Custom Request Headers	843
150.4 File Uploads	844
150.5 Sending Raw POST Data	844
150.6 HTTP Authentication	845
150.7 Sending Multiple Requests With the Same Client	845

150.8 Data Streaming	846
151HTTP Client - Static Usage	849
151.1 Overview	849
151.2 Quick Start	849
151.3 Available Methods	850
152Translating	851
152.1 Adding translations	851
152.2 Supported formats	852
152.3 Setting a locale	852
152.4 Translating messages	852
152.5 Caching	853
153I18n View Helpers	855
153.1 Introduction	855
153.2 CurrencyFormat Helper	855
153.3 DateFormat Helper	857
153.4 NumberFormat Helper	858
153.5 Plural Helper	860
153.6 Translate Helper	861
153.7 TranslatePlural Helper	862
153.8 Abstract Translator Helper	863
154I18n Filters	865
154.1 Alnum	865
154.2 Alpha	866
154.3 NumberFormat	867
154.4 NumberParse	867
155I18n Validators	869
156Alnum Validator	871
156.1 Supported Options	871
156.2 Basic usage	871
156.3 Using whitespaces	871
156.4 Using different languages	872
157Alpha Validator	873
158IsFloat	875
158.1 Migration from 2.0-2.3 to 2.4+	876
158.2 Supported Options	876
158.3 Simple integer validation	876
158.4 Localized integer validation	876
158.5 Migration from 2.0-2.3 to 2.4+	877
159Introduction	879
160Input filter specifications	885
160.1 Setup	885
160.2 Example	885
161File Upload Input	887
161.1 Basic Usage	887

162	Introduction	889
163	Basic Usage	891
163.1	Pretty-printing JSON	891
164	Advanced Usage	893
164.1	JSON Objects	893
164.2	Encoding PHP objects	893
164.3	Internal Encoder/Decoder	894
164.4	JSON Expressions	894
165	XML to JSON conversion	895
165.1	Example	895
166	Zend\Json\Server - JSON-RPC server	897
166.1	Introduction	897
166.2	Basic Usage	897
166.3	Advanced Details	900
167	Introduction to Zend\Ldap	905
167.1	Theory of operation	905
168	API overview	909
168.1	Configuration / options	909
168.2	API Reference	910
169	Zend\Ldap\Ldap	913
169.1	Zend\Ldap\Collection	914
170	Zend\Ldap\Attribute	915
171	Zend\Ldap\Converter\Converter	917
172	Zend\Ldap\Dn	919
173	Zend\Ldap\Filter	923
174	Zend\Ldap\Node	927
175	Zend\Ldap\Node\RootDse	929
175.1	OpenLDAP	931
175.2	ActiveDirectory	931
175.3	eDirectory	932
176	Zend\Ldap\Node\Schema	935
176.1	OpenLDAP	937
176.2	ActiveDirectory	938
177	Zend\Ldap\Ldif\Encoder	939
178	Usage Scenarios	941
178.1	Authentication scenarios	941
178.2	Basic CRUD operations	941
178.3	Extended operations	943
179	Tools	945
179.1	Creation and modification of DN strings	945

179.2 Using the filter API to create search filters	945
179.3 Modify LDAP entries using the Attribute API	946
180Object-oriented access to the LDAP tree using Zend\Ldap\Node	947
180.1 Basic CRUD operations	947
180.2 Extended operations	947
180.3 Tree traversal	947
181Getting information from the LDAP server	949
181.1 RootDSE	949
181.2 Schema Browsing	949
182Serializing LDAP data to and from LDIF	951
182.1 Serialize a LDAP entry to LDIF	951
182.2 Deserialize a LDIF string into a LDAP entry	952
183The AutoloaderFactory	955
183.1 Overview	955
183.2 Quick Start	955
183.3 Configuration Options	956
183.4 Available Methods	956
183.5 Examples	957
184The StandardAutoloader	959
184.1 Overview	959
184.2 Quick Start	960
184.3 Configuration Options	961
184.4 Available Methods	961
184.5 Examples	962
185The ClassMapAutoloader	963
185.1 Overview	963
185.2 Quick Start	963
185.3 Configuration Options	964
185.4 Available Methods	964
185.5 Examples	965
186The ModuleAutoloader	967
186.1 Overview	967
186.2 Quickstart	967
186.3 Configuration Options	967
186.4 Available Methods	968
186.5 Examples	968
187The SplAutoloader Interface	969
187.1 Overview	969
187.2 Quick Start	969
187.3 Configuration Options	970
187.4 Available Methods	970
187.5 Examples	971
188The PluginClassLoader	973
188.1 Overview	973
188.2 Quick Start	973
188.3 Configuration Options	974

188.4 Available Methods	974
188.5 Examples	975
189The ShortNameLocator Interface	979
189.1 Overview	979
189.2 Quick Start	979
189.3 Configuration Options	980
189.4 Available Methods	980
189.5 Examples	980
190The PluginClassLocator interface	981
190.1 Overview	981
190.2 Quick Start	981
190.3 Configuration Options	981
190.4 Available Methods	981
190.5 Examples	982
191The Class Map Generator utility: bin/classmap_generator.php	983
191.1 Overview	983
191.2 Quick Start	983
191.3 Configuration Options	983
192Zend\Log	985
192.1 Overview	985
192.2 Creating a Log	985
192.3 Logging Messages	986
192.4 Destroying a Log	986
192.5 Using Built-in Priorities	986
192.6 Understanding Log Events	987
192.7 Log PHP Errors	987
193Writers	989
193.1 Writing to Streams	989
193.2 Writing to Databases	990
193.3 Writing to FirePHP	991
193.4 Writing to ChromePHP	991
193.5 Writing to Mail	991
193.6 Writing to MongoDB	991
193.7 Writing to Syslog	991
193.8 Writing to Zend Monitor	991
193.9 Stubbing Out the Writer	991
193.10Testing with the Mock	992
193.11Compositing Writers	992
194Filters	995
194.1 Overview	995
194.2 Available filters	995
195Formatters	997
195.1 Simple Formatting	997
195.2 Formatting to XML	998
195.3 Formatting to FirePhp	998
196Introduction to Zend\Mail	999
196.1 Getting started	999

196.2 Configuring the default sendmail transport	1000
197Zend\Mail\Message	1001
197.1 Overview	1001
197.2 Quick Start	1001
197.3 Configuration Options	1003
197.4 Available Methods	1003
197.5 Examples	1006
198Zend\Mail\Transport	1007
198.1 Overview	1007
198.2 Quick Start	1007
198.3 Configuration Options	1009
198.4 Available Methods	1009
198.5 Examples	1009
199Zend\Mail\Transport\SmtpOptions	1011
199.1 Overview	1011
199.2 Quick Start	1011
199.3 Configuration Options	1013
199.4 Available Methods	1013
199.5 Examples	1014
200Zend\Mail\Transport\FileOptions	1015
200.1 Overview	1015
200.2 Quick Start	1015
200.3 Configuration Options	1015
200.4 Available Methods	1016
200.5 Examples	1016
201Introduction to Zend\Math	1017
201.1 Random number generator	1017
201.2 Big integers	1018
202Overview	1021
202.1 Introduction	1021
202.2 Theory of Operation	1022
203Memory Manager	1025
203.1 Creating a Memory Manager	1025
203.2 Managing Memory Objects	1025
203.3 Memory Manager Settings	1027
204Memory Objects	1029
204.1 Movable	1029
204.2 Locked	1029
204.3 Memory container 'value' property	1029
204.4 Memory container interface	1030
205Zend\Mime	1033
205.1 Introduction	1033
205.2 Static Methods and Constants	1033
205.3 Instantiating Zend\Mime	1034
206Zend\Mime\Message	1035
206.1 Introduction	1035

206.2	Instantiation	1035
206.3	Adding MIME Parts	1035
206.4	Boundary handling	1035
206.5	Parsing a string to create a Zend\Mime\Message object	1036
206.6	Available methods	1036
207	Zend\Mime\Part	1037
207.1	Introduction	1037
207.2	Instantiation	1037
207.3	Methods for rendering the message part to a string	1037
207.4	Available methods	1038
208	Introduction to the Module System	1039
208.1	The autoload_*.php Files	1040
209	The Module Manager	1041
209.1	Module Manager Events	1041
209.2	Module Manager Listeners	1042
210	The Module Class	1045
210.1	A Minimal Module	1045
210.2	A Typical Module Class	1046
210.3	The “loadModules.post” Event	1046
210.4	The MVC “bootstrap” Event	1047
211	The Module Autoloader	1049
211.1	Module Autoloader Usage	1049
211.2	Non-Standard / Explicit Module Paths	1050
211.3	Packaging Modules with Phar	1051
212	Best Practices when Creating Modules	1053
212.1	Keep the init() and onBootstrap() methods lightweight	1053
212.2	Do not perform writes within a module	1053
212.3	Utilize a vendor prefix for module names	1054
212.4	Utilize a module prefix for service names	1054
213	Introduction to the MVC Layer	1055
213.1	Basic Application Structure	1056
213.2	Basic Module Structure	1056
213.3	Bootstrapping an Application	1058
213.4	Bootstrapping a Modular Application	1060
213.5	Conclusion	1061
214	Quick Start	1063
214.1	Install the Zend Skeleton Application	1063
214.2	Create a New Module	1064
214.3	Update the Module Class	1064
214.4	Create a Controller	1065
214.5	Create a View Script	1066
214.6	View scripts for module names with subnamespaces	1067
214.7	Create a Route	1067
214.8	Tell the Application About our Module	1069
214.9	Test it Out!	1069
215	Default Services	1071

215.1 Theory of Operation	1071
215.2 ServiceManager	1071
215.3 Abstract Factories	1076
215.4 Plugin Managers	1078
215.5 ViewManager	1079
215.6 Application Configuration Options	1080
215.7 Default Configuration Options	1082
216Routing	1085
216.1 Router Types	1087
216.2 HTTP Route Types	1087
216.3 HTTP Routing Examples	1093
216.4 Console Route Types	1097
217The MvcEvent	1099
217.1 Order of events	1100
217.2 MvcEvent::EVENT_BOOTSTRAP	1100
217.3 MvcEvent::EVENT_ROUTE	1101
217.4 MvcEvent::EVENT_DISPATCH	1102
217.5 MvcEvent::EVENT_DISPATCH_ERROR	1103
217.6 MvcEvent::EVENT_RENDER	1105
217.7 MvcEvent::EVENT_RENDER_ERROR	1106
217.8 MvcEvent::EVENT_FINISH	1107
218The SendResponseEvent	1109
218.1 Listeners	1109
218.2 Triggerers	1110
219Available Controllers	1111
219.1 Common Interfaces Used With Controllers	1111
219.2 The AbstractActionController	1113
219.3 The AbstractRestfulController	1114
220Controller Plugins	1117
220.1 AcceptableViewModelSelector Plugin	1118
220.2 FlashMessenger Plugin	1118
220.3 Forward Plugin	1121
220.4 Identity Plugin	1121
220.5 Layout Plugin	1122
220.6 Params Plugin	1122
220.7 Post/Redirect/Get Plugin	1123
220.8 File Post/Redirect/Get Plugin	1124
220.9 Redirect Plugin	1125
220.10Url Plugin	1126
221Examples	1127
221.1 Controllers	1127
221.2 Bootstrapping	1128
222Introduction to Zend\Navigation	1131
222.1 Pages and Containers	1131
222.2 View Helpers	1132
223Quick Start	1133
223.1 Using multiple navigations	1134

224	Pages	1137
224.1	Common page features	1137
224.2	Zend\Navigation\Page\Mvc	1139
224.3	Zend\Navigation\Page\Uri	1142
224.4	Creating custom page types	1143
224.5	Creating pages using the page factory	1144
225	Containers	1147
225.1	Creating containers	1147
225.2	Adding pages	1154
225.3	Removing pages	1154
225.4	Finding pages	1155
225.5	Iterating containers	1157
225.6	Other operations	1158
226	View Helpers	1161
226.1	Introduction	1161
226.2	Translation of labels and titles	1162
226.3	Integration with ACL	1163
226.4	Navigation setup used in examples	1163
227	View Helper - Breadcrumbs	1169
227.1	Introduction	1169
227.2	Basic usage	1170
227.3	Specifying indentation	1170
227.4	Customize output	1170
227.5	Rendering using a partial view script	1171
228	View Helper - Links	1173
228.1	Introduction	1173
228.2	Basic usage	1175
229	View Helper - Menu	1179
229.1	Introduction	1179
229.2	Basic usage	1180
229.3	Calling renderMenu() directly	1181
229.4	Rendering the deepest active menu	1182
229.5	Rendering with maximum depth	1182
229.6	Rendering with minimum depth	1183
229.7	Rendering only the active branch	1184
229.8	Rendering only the active branch with minimum depth	1185
229.9	Rendering only the active branch with maximum depth	1186
229.10	Rendering only the active branch with maximum depth and no parents	1186
229.11	Rendering a custom menu using a partial view script	1186
230	View Helper - Sitemap	1189
230.1	Introduction	1189
230.2	Basic usage	1190
230.3	Rendering using no ACL role	1191
230.4	Rendering using a maximum depth	1192
231	View Helper - Navigation Proxy	1195
231.1	Introduction	1195
231.2	Methods	1195

232	Introduction to Zend\Paginator	1197
233	Usage	1199
233.1	Paginating data collections	1199
233.2	The DbSelect adapter	1200
233.3	Rendering pages with view scripts	1201
234	Configuration	1207
235	Advanced usage	1209
235.1	Custom data source adapters	1209
235.2	Custom scrolling styles	1209
235.3	Caching features	1210
236	Introduction to Zend\Permissions\Acl	1213
236.1	Resources	1213
236.2	Roles	1214
236.3	Creating the Access Control List	1215
236.4	Registering Roles	1215
236.5	Defining Access Controls	1216
236.6	Querying an ACL	1217
237	Refining Access Controls	1219
237.1	Precise Access Controls	1219
237.2	Removing Access Controls	1221
238	Advanced Usage	1223
238.1	Storing ACL Data for Persistence	1223
238.2	Writing Conditional ACL Rules with Assertions	1223
239	Introduction to Zend\Permissions\Rbac	1225
239.1	Roles	1225
239.2	Permissions	1225
239.3	Dynamic Assertions	1226
240	Methods	1227
241	Examples	1229
241.1	Roles	1229
241.2	Permissions	1230
241.3	Dynamic Assertions	1230
242	Progress Bars	1233
242.1	Introduction	1233
242.2	Basic Usage	1233
242.3	Persistent Progress	1234
242.4	Standard Adapters	1234
243	File Upload Handlers	1237
243.1	Introduction	1237
243.2	Methods of Reporting Progress	1237
243.3	Standard Handlers	1238
244	Introduction to Zend\Serializer	1241
244.1	Quick Start	1241
244.2	Basic configuration Options	1242

244.3 Available Methods	1242
245Zend\Serializer\Adapter	1245
245.1 The PhpSerialize Adapter	1245
245.2 The IgBinary Adapter	1245
245.3 The Wddx Adapter	1245
245.4 The Json Adapter	1246
245.5 The PythonPickle Adapter	1246
245.6 The PhpCode Adapter	1247
246Introduction to Zend\Server	1249
247Zend\Server\Reflection	1251
247.1 Introduction	1251
247.2 Usage	1251
248Zend\ServiceManager	1253
248.1 Introduction	1253
248.2 Additional API	1253
249Zend\ServiceManager Quick Start	1257
249.1 Using Configuration	1257
249.2 Modules as Service Providers	1258
249.3 Examples	1258
250Delegator service factories	1263
250.1 Delegator factory signature	1263
250.2 A Delegator factory use case	1263
251Lazy Services	1267
251.1 Use cases	1267
251.2 Setup	1267
251.3 Practical example	1267
251.4 Configuration	1269
252Session Config	1271
252.1 Standard Config	1271
252.2 Session Config	1272
252.3 Custom Configuration	1274
253Session Container	1275
253.1 Basic Usage	1275
253.2 Setting the Default Session Manager	1275
254Session Manager	1277
254.1 Initializing the Session Manager	1277
254.2 Session Compatibility	1279
255Session Save Handlers	1281
255.1 Cache	1281
255.2 DbTableGateway	1282
255.3 MongoDB	1282
255.4 Custom Save Handlers	1283
256Session Storage	1285
256.1 Array Storage	1285

256.2 Session Storage	1286
256.3 Session Array Storage	1286
256.4 Custom Storage	1286
257Session Validators	1287
257.1 Http User Agent	1287
257.2 Remote Addr	1287
257.3 Custom Validators	1288
258Zend\Soap\Server	1289
258.1 Zend\Soap\Server constructor	1289
258.2 Methods to define Web Service API	1290
258.3 Request and response objects handling	1291
258.4 Document/Literal WSDL Handling	1293
259Zend\Soap\Client	1295
259.1 Zend\Soap\Client Constructor	1295
259.2 Performing SOAP Requests	1296
260WSDL Accessor	1299
260.1 Zend\Soap\Wsd constructor	1299
260.2 addMessage() method	1299
260.3 addPortType() method	1300
260.4 addPortOperation() method	1300
260.5 addBinding() method	1300
260.6 addBindingOperation() method	1301
260.7 addSoapBinding() method	1301
260.8 addSoapOperation() method	1301
260.9 addService() method	1301
260.10Type mapping	1302
260.11addDocumentation() method	1303
260.12Get finalized WSDL document	1304
261AutoDiscovery	1305
261.1 AutoDiscovery Introduction	1305
261.2 Class autodiscovering	1306
261.3 Functions autodiscovering	1307
261.4 Autodiscovering Datatypes	1307
261.5 WSDL Binding Styles	1308
262Zend\Stdlib\Hydrator	1309
262.1 HydratorInterface	1309
262.2 Usage	1310
262.3 Available Implementations	1310
263Zend\Stdlib\Hydrator\Filter	1311
263.1 Filter implementations	1311
263.2 Remove filters	1312
263.3 Add filters	1312
263.4 Use the composite for complex filters	1313
263.5 Using the provider interface	1314
264Zend\Stdlib\Hydrator\Strategy	1317
264.1 Adding strategies to the hydrators	1318
264.2 Available implementations	1318

264.3 Writing custom strategies	1319
265AggregateHydrator	1321
265.1 Installation requirements	1321
265.2 Basic usage	1321
265.3 Advanced use cases	1322
266CompositeNamingStrategy	1325
266.1 Basic Usage	1325
267IdentityNamingStrategy	1327
267.1 Basic Usage	1327
268MapNamingStrategy	1329
268.1 Basic Usage	1329
269UnderscoreNamingStrategy	1331
269.1 Basic Usage	1331
270Introduction to Zend\Tag	1333
271Creating tag clouds with Zend\Tag\Cloud	1335
271.1 Decorators	1337
272Introduction to Zend\Test	1341
273Unit testing with PHPUnit	1343
273.1 Setup your TestCase	1344
273.2 Testing your Controllers and MVC Applications	1344
274Zend\Text\Figlet	1349
274.1 Introduction	1349
274.2 Basic Usage	1350
275Zend\Text\Table	1351
275.1 Introduction	1351
275.2 Basic Usage	1352
276Zend\Uri	1353
276.1 Overview	1353
276.2 Creating a New URI	1353
276.3 Manipulating an Existing URI	1354
276.4 Common Instance Methods	1354
277Introduction to Zend\Validator	1359
277.1 What is a validator?	1359
277.2 Basic usage of validators	1359
277.3 Customizing messages	1360
277.4 Translating messages	1361
278Standard Validation Classes	1363
278.1 Included Validators	1363
278.2 Deprecated Validators	1364
279Alnum Validator	1365
279.1 Supported Options	1365

279.2 Basic usage	1365
279.3 Using whitespaces	1365
279.4 Using different languages	1366
280Alpha Validator	1367
280.1 Supported options for Zend\Validator\Alpha	1367
280.2 Basic usage	1367
280.3 Using whitespaces	1367
280.4 Using different languages	1368
281Barcode Validator	1369
281.1 Supported options for Zend\Validator\Barcode	1371
281.2 Basic usage	1372
281.3 Optional checksum	1372
281.4 Writing custom adapters	1372
282Between Validator	1375
282.1 Supported options for Zend\Validator\Between	1375
282.2 Default behaviour for Zend\Validator\Between	1375
282.3 Validation exclusive the border values	1376
283Callback Validator	1377
283.1 Supported options for Zend\Validator\Callback	1377
283.2 Basic usage	1377
283.3 Usage with closures	1378
283.4 Usage with class-based callbacks	1378
283.5 Adding options	1379
284Date Validator	1381
284.1 Supported options for Zend\Validator\Date	1381
284.2 Default date validation	1381
284.3 Self defined date validation	1381
285Db\RecordExists and Db\NoRecordExists Validators	1383
285.1 Supported options for Zend\Validator\Db*	1383
285.2 Basic usage	1383
285.3 Excluding records	1384
285.4 Database Schemas	1385
285.5 Using a Select object	1386
286Digits Validator	1387
286.1 Supported options for Zend\Validator\Digits	1387
286.2 Validating digits	1387
287File Validation Classes	1389
287.1 Crc32	1389
287.2 ExcludeExtension	1390
287.3 ExcludeMimeType	1390
287.4 Exists	1390
287.5 Extension	1391
287.6 Hash	1392
287.7 ImageSize	1393
287.8 IsCompressed	1394
287.9 IsImage	1394
287.10Md5	1394

287.11MimeType	1395
287.12NotExists	1396
287.13Sha1	1397
287.14Size	1398
287.15UploadFile	1399
287.16WordCount	1399
288Hex Validator	1401
288.1 Supported options for Zend\Validator\Hex	1401
289Hostname Validator	1403
289.1 Supported options for Zend\Validator\Hostname	1403
289.2 Basic usage	1403
289.3 Validating different types of hostnames	1404
289.4 Validating International Domains Names	1404
289.5 Validating Top Level Domains	1405
290Iban Validator	1407
290.1 Supported options for Zend\Validator\Iban	1407
290.2 IBAN validation	1407
291Identical Validator	1409
291.1 Supported options for Zend\Validator\Identical	1409
291.2 Basic usage	1409
291.3 Identical objects	1410
291.4 Form elements	1410
291.5 Strict validation	1412
291.6 Configuration	1412
292Ip Validator	1415
292.1 Supported options for Zend\Validator\Ip	1415
292.2 Basic usage	1415
292.3 Validate IPv4 or IPV6 alone	1416
293Isbn Validator	1417
293.1 Supported options for Zend\Validator\Isbn	1417
293.2 Basic usage	1417
293.3 Setting an explicit ISBN validation type	1417
293.4 Specifying a separator restriction	1418
294IsInstanceOf Validator	1419
294.1 Supported options	1419
294.2 Basic usage	1419
295IsFloat	1421
295.1 Supported options for Zend\Validator\IsFloat	1421
295.2 Simple float validation	1421
295.3 Localized float validation	1421
296IsInt	1423
296.1 Supported Options	1423
296.2 Simple integer validation	1423
296.3 Localized integer validation	1423
296.4 Migration from 2.0-2.3 to 2.4+	1424

297	Regex Validator	1425
297.1	Supported options for Zend\Validator\Regex	1425
297.2	Validation with Zend\Validator\Regex	1425
297.3	Pattern handling	1425
298	Sitemap Validators	1427
298.1	Sitemap\Changefreq	1427
298.2	Sitemap\Lastmod	1427
298.3	Sitemap\Loc	1428
298.4	Sitemap\Priority	1428
298.5	Supported options for Zend\Validator\Sitemap_*	1428
299	Step Validator	1429
299.1	Supported options for Zend\Validator\Step	1429
299.2	Basic usage	1429
299.3	Using floating-point values	1429
300	Timezone Validator	1431
300.1	Supported validation types	1431
300.2	Basic Usage	1431
301	Uri Validator	1433
301.1	Supported options	1433
301.2	Basic usage	1433
302	Validator Chains	1435
302.1	Overview	1435
302.2	Setting Validator Chain Order	1436
303	Writing Validators	1437
303.1	Overview	1437
303.2	Creating a Simple Validation Class	1437
303.3	Writing a Validation Class having Dependent Conditions	1438
303.4	Validation with Independent Conditions, Multiple Reasons for Failure	1439
304	Validation Messages	1441
304.1	Using pre-translated validation messages	1442
304.2	Limit the size of a validation message	1442
305	Getting the Zend Framework Version	1443
305.1	Overview	1443
305.2	Example of the compareVersion() Method	1443
305.3	Example of the getLatest() Method	1443
306	Zend\View Quick Start	1445
306.1	Overview	1445
306.2	Usage	1446
307	The PhpRenderer	1459
307.1	Usage	1459
307.2	Options and Configuration	1463
307.3	Additional Methods	1464
308	PhpRenderer View Scripts	1465
308.1	Escaping Output	1466

309The ViewEvent	1467
309.1 Order of events	1468
309.2 ViewEvent::EVENT_RENDERER	1468
309.3 ViewEvent::EVENT_RENDERER_POST	1469
309.4 ViewEvent::EVENT_RESPONSE	1469
310View Helpers	1471
310.1 Introduction	1471
310.2 Included Helpers	1472
311View Helper - BasePath	1475
311.1 Introduction	1475
311.2 Basic Usage	1475
312View Helper - Cycle	1477
312.1 Introduction	1477
312.2 Basic Usage	1477
312.3 Working with two or more cycles	1478
313View Helper - Doctype	1479
313.1 Introduction	1479
313.2 Basic Usage	1480
313.3 Retrieving the Doctype	1480
313.4 Choosing a Doctype to Use with the Open Graph Protocol	1480
313.5 Zend MVC View Manager	1481
314FlashMessenger Helper	1483
314.1 Introduction	1483
314.2 Basic Usage	1483
314.3 CSS Layout	1483
314.4 HTML Layout	1484
314.5 Sample Modification for Twitter Bootstrap 3	1484
314.6 Alternative Configuration of the ViewHelper Layout	1485
315Gravatar Helper	1487
315.1 Introduction	1487
315.2 Basic Usage	1487
315.3 Custom Settings	1487
316View Helper - HeadLink	1491
316.1 Introduction	1491
316.2 Basic Usage	1492
317View Helper - HeadMeta	1493
317.1 Introduction	1493
317.2 Basic Usage	1494
317.3 Usage with XHTML1_RDFa doctype	1495
318View Helper - HeadScript	1497
318.1 Introduction	1497
318.2 Basic Usage	1499
318.3 Capturing Scripts	1499
319View Helper - HeadStyle	1501
319.1 Introduction	1501
319.2 Basic Usage	1504

319.3 Capturing Style Declarations	1505
320View Helper - HeadTitle	1507
320.1 Introduction	1507
320.2 Basic Usage	1507
321View Helper - HtmlList	1509
321.1 Introduction	1509
321.2 Basic Usage	1509
322View Helper - HTML Object	1513
322.1 Introduction	1513
322.2 Flash helper	1513
322.3 Customizing the object by passing additional arguments	1514
323View Helper - Identity	1515
323.1 Introduction	1515
323.2 Basic Usage	1515
323.3 Using with ServiceManager	1515
324View Helper - InlineScript	1517
324.1 Introduction	1517
324.2 Basic Usage	1517
324.3 Capturing Scripts	1518
325View Helper - JSON	1519
325.1 Introduction	1519
325.2 Basic Usage	1519
326View Helper - Partial	1521
326.1 Introduction	1521
326.2 Basic Usage	1521
326.3 Using PartialLoop to Render Iterable Models	1522
327View Helper - Placeholder	1525
327.1 Introduction	1525
327.2 Basic Usage	1525
327.3 Aggregate Content	1525
327.4 Capture Content	1526
327.5 Concrete Implementations	1527
328View Helper - URL	1529
328.1 Basic Usage	1529
328.2 Query String Arguments	1530
328.3 Fragments	1531
328.4 Fully Qualified Domain Name	1532
328.5 Reusing Matched Parameters	1532
329Advanced usage of helpers	1535
329.1 Registering Helpers	1535
329.2 Writing Custom Helpers	1536
329.3 Registering Concrete Helpers	1538
330Introduction to Zend\XmlRpc	1539
330.1 Quick Start	1539

331	Zend\XmlRpc\Client	1541
331.1	Introduction	1541
331.2	Method Calls	1541
331.3	Types and Conversions	1542
331.4	Server Proxy Object	1544
331.5	Error Handling	1544
331.6	Server Introspection	1545
331.7	From Request to Response	1546
331.8	HTTP Client and Testing	1546
332	Zend\XmlRpc\Server	1549
332.1	Introduction	1549
332.2	Basic Usage	1549
332.3	Server Structure	1549
332.4	Anatomy of a webservice	1550
332.5	Conventions	1550
332.6	Utilizing Namespaces	1551
332.7	Custom Request Objects	1552
332.8	Custom Responses	1552
332.9	Handling Exceptions via Faults	1552
332.10	Caching Server Definitions Between Requests	1552
332.11	Usage Examples	1553
332.12	Performance optimization	1557
333	ZendService\Akismet	1559
333.1	Introduction	1559
333.2	Verify an API key	1559
333.3	Check for spam	1560
333.4	Submitting known spam	1561
333.5	Submitting false positives (ham)	1561
333.6	Zend-specific Methods	1562
334	ZendService\Amazon	1563
334.1	Introduction	1563
334.2	Country Codes	1564
334.3	Looking up a Specific Amazon Item by ASIN	1564
334.4	Performing Amazon Item Searches	1565
334.5	Using the Alternative Query API	1565
334.6	ZendService\Amazon Classes	1566
335	ZendService\Amazon\S3	1573
335.1	Introduction	1573
335.2	Registering with Amazon S3	1573
335.3	API Documentation	1573
335.4	Features	1573
335.5	Getting Started	1574
335.6	Bucket operations	1574
335.7	Object operations	1575
335.8	Data Streaming	1576
335.9	Stream wrapper	1577
336	ZendService\Amazon\Sqs	1579
336.1	Introduction	1579
336.2	Registering with Amazon SQS	1579
336.3	API Documentation	1579

336.4 Features	1580
336.5 Getting Started	1580
336.6 Queue operations	1580
336.7 Message operations	1581
337ZendService\Amazon\Ec2	1583
337.1 Introduction	1583
337.2 What is Amazon Ec2?	1583
337.3 Static Methods	1583
338ZendService\Amazon\Ec2: CloudWatch Monitoring	1585
338.1 CloudWatch Usage	1585
339ZendService\Amazon\Ec2: Elastic Block Storage (EBS)	1587
339.1 Create EBS Volumes and Snapshots	1587
339.2 Describing EBS Volumes and Snapshots	1588
339.3 Attach and Detaching Volumes from Instances	1589
339.4 Deleting EBS Volumes and Snapshots	1589
340ZendService\Amazon\Ec2: Elastic IP Addresses	1591
340.1 Allocating a new Elastic IP	1591
341ZendService\Amazon\Ec2: Instances	1593
341.1 Instance Types	1593
341.2 Running Amazon EC2 Instances	1594
341.3 Amazon Instance Utilities	1596
342ZendService\Amazon\Ec2: Regions and Availability Zones	1599
342.1 Amazon EC2 Regions	1599
342.2 Amazon EC2 Availability Zones	1600
343ZendService\Amazon\Ec2: Reserved Instances	1601
343.1 How Reserved Instances are Applied	1601
343.2 Reserved Instances Usage	1602
344ZendService\Amazon\Ec2: Security Groups	1603
344.1 Security Group Maintenance	1603
344.2 Authorizing Access	1604
344.3 Revoking Access	1605
345ZendService\Amazon\Ec2: Windows Instances	1607
345.1 Windows Instances Usage	1608
346ZendService\Apple\Apns	1609
346.1 Introduction	1609
346.2 Quick Start	1609
346.3 Feedback Service	1611
347ZendService\Audioscrobbler	1613
347.1 Introduction	1613
347.2 Users	1613
347.3 Artists	1615
347.4 Tracks	1616
347.5 Tags	1616
347.6 Groups	1616
347.7 Forums	1617

348	ZendService\Delicious	1619
348.1	Introduction	1619
348.2	Retrieving posts	1619
348.3	ZendService\Delicious\PostList	1620
348.4	Editing posts	1621
348.5	Deleting posts	1622
348.6	Adding new posts	1622
348.7	Tags	1622
348.8	Bundles	1623
348.9	Public data	1623
348.10	HTTP client	1624
349	ZendService\DeveloperGarden	1625
349.1	Introduction	1625
349.2	BaseUserService	1626
349.3	IP Location	1628
349.4	Local Search	1628
349.5	Send SMS	1628
349.6	SMS Validation	1629
349.7	Voice Call	1629
349.8	ConferenceCall	1630
349.9	Performance and Caching	1632
350	ZendService\Flickr	1633
350.1	Introduction	1633
350.2	Finding Flickr Users' Photos and Information	1633
350.3	Finding photos From a Group Pool	1634
350.4	Retrieving Flickr Image Details	1634
350.5	ZendService\Flickr Result Classes	1635
351	ZendService\Google\Gcm	1637
351.1	Introduction	1637
351.2	Quick Start	1637
352	ZendService\LiveDocx	1639
352.1	Introduction	1639
352.2	ZendService\LiveDocx\MailMerge	1641
353	ZendService\Rackspace	1661
353.1	Introduction	1661
353.2	Registering with Rackspace	1661
353.3	Cloud Files	1661
353.4	Cloud Servers	1662
353.5	Available Methods	1662
354	ZendService\Rackspace\Servers	1665
354.1	Overview	1665
354.2	Terminology	1665
354.3	Quick Start	1666
354.4	Available Methods	1667
354.5	Examples	1671
355	ZendService\Rackspace\Files	1673
355.1	Overview	1673
355.2	Quick Start	1673

355.3 Available Methods	1675
355.4 Examples	1678
356ZendService\ReCaptcha	1681
356.1 Introduction	1681
356.2 Simplest use	1681
356.3 Hiding email addresses	1682
357ZendService\SlideShare	1685
357.1 Getting Started with ZendService\SlideShare	1685
357.2 The SlideShow object	1685
357.3 Retrieving a single slide show	1688
357.4 Retrieving Groups of Slide Shows	1688
357.5 ZendService\SlideShare Caching policies	1689
357.6 Changing the behavior of the HTTP Client	1690
358ZendService\StrikeIron	1691
358.1 Overview	1691
358.2 Registering with StrikeIron	1692
358.3 Getting Started	1692
358.4 Making Your First Query	1693
358.5 Examining Results	1693
358.6 Handling Errors	1694
358.7 Checking Your Subscription	1695
359ZendService\StrikeIron: Bundled Services	1697
359.1 ZIP Code Information	1697
359.2 U.S. Address Verification	1698
359.3 Sales & Use Tax Basic	1699
360ZendService\StrikeIron: Advanced Uses	1701
360.1 Using Services by WSDL	1701
360.2 Viewing SOAP Transactions	1702
361ZendService\Technorati	1703
361.1 Introduction	1703
361.2 Getting Started	1703
361.3 Making Your First Query	1703
361.4 Consuming Results	1704
361.5 Handling Errors	1706
361.6 Checking Your API Key Daily Usage	1706
361.7 Available Technorati Queries	1707
361.8 ZendService\Technorati Classes	1710
362ZendService\Twitter	1715
362.1 Introduction	1715
362.2 Quick Start	1716
362.3 Authentication	1717
362.4 Account Methods	1718
362.5 Application Methods	1719
362.6 Blocking Methods	1719
362.7 Direct Message Methods	1720
362.8 Favorites Methods	1721
362.9 Friendship Methods	1722
362.10Search Methods	1722

362.11	Status Methods	1722
362.12	User Methods	1725
363	ZendService\WindowsAzure	1727
363.1	Introduction	1727
363.2	Installing the Windows Azure SDK	1727
363.3	API Documentation	1727
363.4	Features	1728
363.5	Architecture	1728
364	ZendService\WindowsAzure\Storage\Blob	1729
364.1	API Examples	1729
364.2	Root container	1731
364.3	Blob storage stream wrapper	1731
364.4	Shared Access Signature	1732
365	ZendService\WindowsAzure\Storage\Table	1735
365.1	Operations on tables	1735
365.2	Operations on entities	1736
365.3	Table storage session handler	1742
366	ZendService\WindowsAzure\Storage\Queue	1745
366.1	API Examples	1745
367	Copyright Information	1749
368	Introduction to Zend Framework 2	1751
369	User Guide	1753
370	In-depth tutorial for beginners	1755
371	Getting Started With Zend Studio 10 & Zend Server 6	1757
372	Zend Framework Tool (ZFTool)	1759
373	Learning Zend Framework 2	1761
374	Migration	1763
375	Zend Framework 2 Reference	1765
375.1	Zend\Authentication	1765
375.2	Zend\Barcode	1765
375.3	Zend\Cache	1765
375.4	Zend\Captcha	1766
375.5	Zend\Code\Generator	1766
375.6	Zend\Config	1766
375.7	Zend\Console	1766
375.8	Zend\Console\Getopt	1767
375.9	Zend\Crypt	1767
375.10	Zend\Db	1767
375.11	Zend\Debug	1767
375.12	Zend\Di	1767
375.13	Zend\Dom	1768
375.14	Zend\Escaper	1768
375.15	Zend\EventManager	1768

375.16	Zend\Feed	1768
375.17	Zend\File	1769
375.18	Zend\Filter	1769
375.19	Zend\Form	1769
375.20	Zend\Http	1769
375.21	Zend\I18n	1770
375.22	Zend\InputFilter	1770
375.23	Zend\Json	1770
375.24	Zend\Ldap	1770
375.25	Zend\Loader	1770
375.26	Zend\Log	1771
375.27	Zend\Mail	1771
375.28	Zend\Math	1771
375.29	Zend\Memory	1771
375.30	Zend\Mime	1771
375.31	Zend\ModuleManager	1772
375.32	Zend\Mvc	1772
375.33	Zend\Navigation	1772
375.34	Zend\Paginator	1772
375.35	Zend\Permissions\Acl	1773
375.36	Zend\Permissions\Rbac	1773
375.37	Zend\ProgressBar	1773
375.38	Zend\Serializer	1773
375.39	Zend\Server	1773
375.40	Zend\ServiceManager	1773
375.41	Zend\Session	1774
375.42	Zend\Soap	1774
375.43	Zend\Stdlib	1774
375.44	Zend\Tag	1774
375.45	Zend\Test	1774
375.46	Zend\Text	1775
375.47	Zend\Uri	1775
375.48	Zend\Validator	1775
375.49	Zend\Version	1776
375.50	Zend\View	1776
375.51	Zend\XmlRpc	1777

376 Services for Zend Framework 2 Reference 1779

376.1	ZendService\Akismet	1779
376.2	ZendService\Amazon	1779
376.3	ZendService\AppleApns	1780
376.4	ZendService\Audioscrobbler	1780
376.5	ZendService\Delicious	1780
376.6	ZendService\DeveloperGarden	1780
376.7	ZendService\Flickr	1780
376.8	ZendService\Google\Gcm	1780
376.9	ZendService\LiveDocx	1780
376.10	ZendService\Rackspace	1780
376.11	ZendService\ReCaptcha	1780
376.12	ZendService\SlideShare	1781
376.13	ZendService\StrikeIron	1781
376.14	ZendService\Technorati	1781
376.15	ZendService\Twitter	1781
376.16	ZendService\WindowsAzure	1781

377	Copyright	1783
378	Indices and tables	1785

CHAPTER 1

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.

Using Composer

The recommended way to start a new Zend Framework project is to clone the skeleton application and use `composer` to install dependencies using the `create-project` command:

```
1 curl -s https://getcomposer.org/installer | php --
2 php composer.phar create-project -sdev --repository-url="https://packages.
  ↪ zendframework.com" zendframework/skeleton-application path/to/install
```

Alternately, clone the repository and manually invoke `composer` using the shipped `composer.phar`:

```
1 cd my/project/dir
2 git clone git://github.com/zendframework/ZendSkeletonApplication.git
3 cd ZendSkeletonApplication
4 php composer.phar self-update
5 php composer.phar install
```

(The `self-update` directive is to ensure you have an up-to-date `composer.phar` available.)

Another alternative for downloading the project is to grab it via `curl`, and then pass it to `tar`:

```
1 cd my/project/dir
2 curl -#L https://github.com/zendframework/ZendSkeletonApplication/tarball/master |
  ↪ tar xz --strip-components=1
```

You would then invoke `composer` to install dependencies per the previous example.

Using Git submodules

Alternatively, you can install using native git submodules:

```
1 git clone git://github.com/zendframework/ZendSkeletonApplication.git --recursive
```

Web Server Setup

PHP CLI Server

The simplest way to get started if you are using PHP 5.4 or above is to start the internal PHP cli-server in the root directory:

```
1 php -S 0.0.0.0:8080 -t public/ public/index.php
```

This will start the cli-server on port 8080, and bind it to all network interfaces.

Note: The built-in CLI server is *for development only*.

Apache Setup

To use Apache, setup a virtual host to point to the `public/` directory of the project. It should look something like below:

```
1 <VirtualHost *:80>
2     ServerName zf2-tutorial.localhost
3     DocumentRoot /path/to/zf2-tutorial/public
4
5     <Directory /path/to/zf2-tutorial/public>
6         AllowOverride All
7         Order allow,deny
8         Allow from all
9     </Directory>
10 </VirtualHost>
```

or, if you are using Apache 2.4 or above:

```
1 <VirtualHost *:80>
2     ServerName zf2-tutorial.localhost
3     DocumentRoot /path/to/zf2-tutorial/public
4
5     <Directory /path/to/zf2-tutorial/public>
6         AllowOverride All
7         Require all granted
8     </Directory>
9 </VirtualHost>
```

Rewrite Configuration

URL rewriting is a common function of *HTTP* servers, and allows all HTTP requests to be routed through the `index.php` entry point of a Zend Framework Application.

Apache comes bundled with the module ‘‘mod_rewrite’’ for URL rewriting. To use it, `mod_rewrite` must either be included at compile time or enabled as a Dynamic Shared Object (*DSO*). Please consult the [Apache documentation](#) for your version for more information.

The Zend Framework Skeleton Application comes with a `.htaccess` that includes rewrite rules to cover most use cases:

```

1 RewriteEngine On
2 # The following rule tells Apache that if the requested filename
3 # exists, simply serve it.
4 RewriteCond %{REQUEST_FILENAME} -s [OR]
5 RewriteCond %{REQUEST_FILENAME} -l [OR]
6 RewriteCond %{REQUEST_FILENAME} -d
7 RewriteRule ^.*$ - [NC,L]
8 # The following rewrites all other queries to index.php. The
9 # condition ensures that if you are using Apache aliases to do
10 # mass virtual hosting, the base path will be prepended to
11 # allow proper resolution of the index.php file; it will work
12 # in non-aliased environments as well, providing a safe, one-size
13 # fits all solution.
14 RewriteCond %{REQUEST_URI}::$1 ^(/.+)(.+)::\2$
15 RewriteRule ^(.*) - [E=BASE:%1]
16 RewriteRule ^(.*)$ %{ENV:BASE}index.php [NC,L]
```

Microsoft Internet Information Services

As of version 7.0, *IIS* ships with a standard rewrite engine. You may use the following configuration to create the appropriate rewrite rules.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3   <system.webServer>
4     <rewrite>
5       <rules>
6         <rule name="Imported Rule 1" stopProcessing="true">
7           <match url="^.*$" />
8           <conditions logicalGrouping="MatchAny">
9             <add input="{REQUEST_FILENAME}"
10               matchType="IsFile" pattern=""
11               ignoreCase="false" />
12             <add input="{REQUEST_FILENAME}"
13               matchType="IsDirectory"
14               pattern=""
15               ignoreCase="false" />
16           </conditions>
17           <action type="None" />
18         </rule>
19         <rule name="Imported Rule 2" stopProcessing="true">
20           <match url="^.*$" />
21           <action type="Rewrite" url="index.php" />
22         </rule>
23       </rules>
24     </rewrite>
25   </system.webServer>
26 </configuration>
```

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.

Some assumptions

This tutorial assumes that you are running at least PHP 5.3.23 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.

Note: Alternatively, if you are using PHP 5.4+ you may use the built-in web server instead of Apache for development.

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 4

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:

```
1 php composer.phar create-project --stability="dev" zendframework/skeleton-  
  ↪application path/to/install
```

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:

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

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

```
1 Installing dependencies from lock file  
2 - Installing zendframework/zendframework (dev-master)  
3   Cloning 18c8e223f070deb07c17543ed938b54542aa0ed8  
4  
5 Generating autoload files
```

Note: If you see this message:

```
1 [RuntimeException]
2   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:

```
1 php composer.phar install
```

run instead:

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

Note: For windows users with wamp:

1. Install composer for windows Check composer is properly installed by running

```
1 composer
```

2. Install git for windows. Also need to add git path in windows environment variable Check git is properly installed by running

```
1 git
```

3. Now install zf2 using command

```
1 composer create-project -s dev zendframework/skeleton-application path/to/install
```

We can now move on to the web server setup.

Using the Apache Web Server

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:

```
1 <VirtualHost *:80>
2     ServerName zf2-tutorial.localhost
3     DocumentRoot /path/to/zf2-tutorial/public
4     SetEnv APPLICATION_ENV "development"
5     <Directory /path/to/zf2-tutorial/public>
6         DirectoryIndex index.php
7         AllowOverride All
8         Order allow,deny
9         Allow from all
```

```
10     </Directory>
11 </VirtualHost>
```

or, if you are using Apache 2.4 or above:

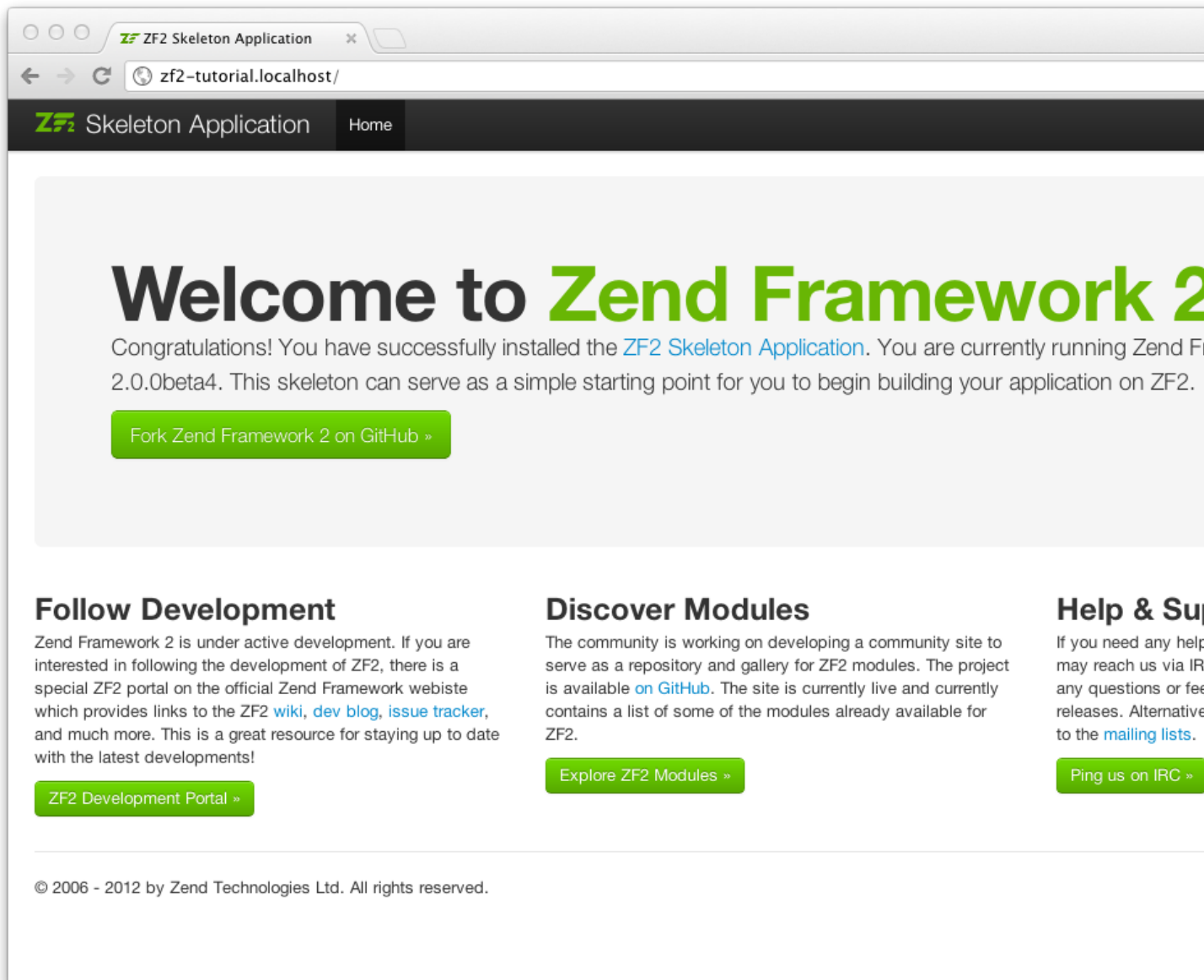
```
1 <VirtualHost *:80>
2     ServerName zf2-tutorial.localhost
3     DocumentRoot /path/to/zf2-tutorial/public
4     SetEnv APPLICATION_ENV "development"
5     <Directory /path/to/zf2-tutorial/public>
6         DirectoryIndex index.php
7         AllowOverride All
8         Require all granted
9     </Directory>
10 </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`.

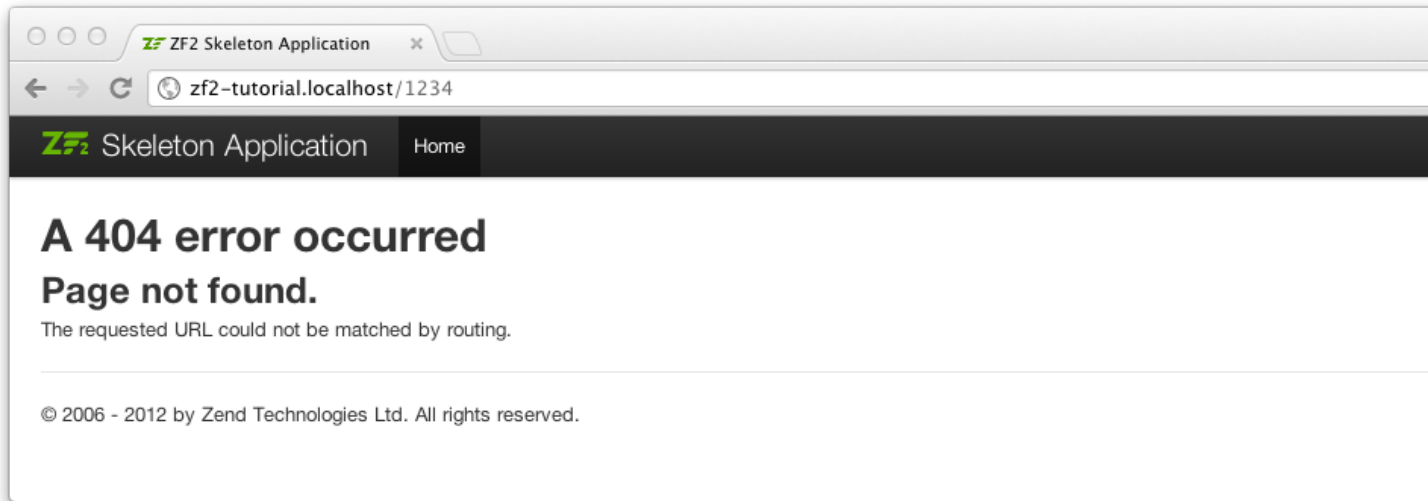
```
1 127.0.0.1          zf2-tutorial.localhost localhost
```

Restart Apache.

If you've done it correctly, it should look 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 using IIS with the URL Rewrite Module, import the following:

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

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

Using the Built-in PHP CLI Server

Alternatively — if you are using PHP 5.4 or above — you can use the built-in CLI server (`cli-server`). To do this, you just start the server in the root directory:

```
1 php -S 0.0.0.0:8080 -t public/ public/index.php
```

This will make the website available on port 8080 on all network interfaces, using `public/index.php` to handle routing. This means the site is accessible via <http://localhost:8080> or <http://<your-local-IP>:8080>.

If you've done it right, you should see the same result as with Apache above.

To test that your routing is working, navigate to <http://localhost:8080/1234> and you should see the same error page as with Apache above.

Note: The built-in CLI server is **for development only**.

Error reporting

Optionally, *when using Apache*, you can use the `APPLICATION_ENV` setting in your `VirtualHost` to let PHP output all its errors to the browser. This can be useful during the development of your application.

Edit `index.php` from the `zf2-tutorial/public/` directory and change it to the following:

```
1 <?php
2
3 /**
4  * Display all errors when APPLICATION_ENV is development.
5  */
6 if ($_SERVER['APPLICATION_ENV'] == 'development') {
7     error_reporting(E_ALL);
8     ini_set("display_errors", 1);
9 }
10
11 /**
12  * This makes our life easier when dealing with paths. Everything is relative
13  * to the application root now.
14  */
15 chdir(dirname(__DIR__));
16
17 // Decline static file requests back to the PHP built-in webserver
18 if (php_sapi_name() === 'cli-server' && is_file(__DIR__ . parse_url($_SERVER[
19 ↪ 'REQUEST_URI'], PHP_URL_PATH))) {
20     return false;
21 }
22
23 // Setup autoloading
24 require 'init_autoloader.php';
25
26 // Run the application!
27 Zend\Mvc\Application::init(require 'config/application.config.php')->run();
```


Zend Framework 2 uses a module system to 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.

Setting up the Album module

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

```
1  zf2-tutorial/  
2      /module  
3          /Album  
4              /config  
5              /src  
6                  /Album  
7                      /Controller  
8                      /Form  
9                      /Model  
10             /view  
11                 /album  
12                 /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`:

```
1 namespace Album;
2
3 use Zend\ModuleManager\Feature\AutoloaderProviderInterface;
4 use Zend\ModuleManager\Feature\ConfigProviderInterface;
5
6 class Module implements AutoloaderProviderInterface, ConfigProviderInterface
7 {
8     public function getAutoloaderConfig()
9     {
10         return array(
11             'Zend\Loader\ClassMapAutoloader' => array(
12                 __DIR__ . '/autoload_classmap.php',
13             ),
14             'Zend\Loader\StandardAutoloader' => array(
15                 'namespaces' => array(
16                     __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
17                 ),
18             ),
19         );
20     }
21
22     public function getConfig()
23     {
24         return include __DIR__ . '/config/module.config.php';
25     }
26 }
```

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

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`:

```
1 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 `StandardAutoloader` for us.

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

```

1  "autoload": {
2      "psr-0": { "Album": "module/Album/src/" }
3  },

```

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

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`:

```

1  return array(
2      'controllers' => array(
3          'invokables' => array(
4              'Album\Controller\Album' => 'Album\Controller\AlbumController',
5          ),
6      ),
7      'view_manager' => array(
8          'template_path_stack' => array(
9              'album' => __DIR__ . '/../view',
10          ),
11      ),
12 );

```

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.

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.)

```

1  return array(
2      'modules' => array(
3          'Application',
4          'Album',                                // <-- Add this line
5      ),
6      'module_listener_options' => array(
7          'config_glob_paths'    => array(
8              'config/autoload/{,*.}global,{,*.}local.php',
9          ),
10         'module_paths' => array(

```

```
11         './module',
12         './vendor',
13     ),
14 ),
15 );
```

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 6

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	<code>AlbumController</code>	<code>index</code>
Add new album	<code>AlbumController</code>	<code>add</code>
Edit album	<code>AlbumController</code>	<code>edit</code>
Delete album	<code>AlbumController</code>	<code>delete</code>

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.

```
1 return array(  
2     'controllers' => array(  
3         'invokables' => array(  
4             'Album\Controller\Album' => 'Album\Controller\AlbumController',  
5         ),  
    ),
```

```

6      ),
7
8      // The following section is new and should be added to your file
9      'router' => array(
10         'routes' => array(
11             'album' => array(
12                 'type' => 'segment',
13                 'options' => array(
14                     'route' => '/album[:action][:id]',
15                     'constraints' => array(
16                         'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
17                         'id' => '[0-9]+',
18                     ),
19                     'defaults' => array(
20                         'controller' => 'Album\Controller\Album',
21                         'action' => 'index',
22                     ),
23                 ),
24             ),
25         ),
26     ),
27
28     'view_manager' => array(
29         'template_path_stack' => array(
30             'album' => __DIR__ . '/../view',
31         ),
32     ),
33 );

```

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

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`:

```
1 namespace Album\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\View\Model\ViewModel;
5
6 class AlbumController extends AbstractActionController
7 {
8     public function indexAction()
9     {
10    }
11
12     public function addAction()
13     {
14    }
15 }
```

```
16     public function editAction()  
17     {  
18     }  
19  
20     public function deleteAction()  
21     {  
22     }  
23 }
```

Note: Make sure to register the new Album module in the “modules” section of your `config/application.config.php`. You also have to provide a *Module Class* for the Album module to be recognized by the MVC.

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
<code>http://zf2-tutorial.localhost/album</code>	<code>Album\Controller\AlbumController::indexAction</code>
<code>http://zf2-tutorial.localhost/album/add</code>	<code>Album\Controller\AlbumController::addAction</code>
<code>http://zf2-tutorial.localhost/album/edit</code>	<code>Album\Controller\AlbumController::editAction</code>
<code>http://zf2-tutorial.localhost/album/delete</code>	<code>Album\Controller\AlbumController::deleteAction</code>

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.

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.

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.

```
1 CREATE TABLE album (  
2     id int(11) NOT NULL auto_increment,  
3     artist varchar(100) NOT NULL,  
4     title varchar(100) NOT NULL,  
5     PRIMARY KEY (id)  
6 );  
7 INSERT INTO album (artist, title)  
8     VALUES ('The Military Wives', 'In My Dreams');  
9 INSERT INTO album (artist, title)  
10    VALUES ('Adele', '21');  
11 INSERT INTO album (artist, title)  
12    VALUES ('Bruce Springsteen', 'Wrecking Ball (Deluxe)');  
13 INSERT INTO album (artist, title)  
14    VALUES ('Lana Del Rey', 'Born To Die');  
15 INSERT INTO album (artist, title)  
16    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.

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`:

```
1 namespace Album\Model;
2
3 class Album
4 {
5     public $id;
6     public $artist;
7     public $title;
8
9     public function exchangeArray($data)
10    {
11        $this->id      = (!empty($data['id'])) ? $data['id'] : null;
12        $this->artist  = (!empty($data['artist'])) ? $data['artist'] : null;
13        $this->title   = (!empty($data['title'])) ? $data['title'] : null;
14    }
15 }
```

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:

```
1 namespace Album\Model;
2
3 use Zend\Db\TableGateway\TableGateway;
4
5 class AlbumTable
6 {
7     protected $tableGateway;
8
9     public function __construct(TableGateway $tableGateway)
10    {
11        $this->tableGateway = $tableGateway;
12    }
13
14     public function fetchAll()
15    {
16        $resultSet = $this->tableGateway->select();
17        return $resultSet;
18    }
19 }
```

```

20 public function getAlbum($id)
21 {
22     $id = (int) $id;
23     $rowset = $this->tableGateway->select(array('id' => $id));
24     $row = $rowset->current();
25     if (!$row) {
26         throw new \Exception("Could not find row $id");
27     }
28     return $row;
29 }
30
31 public function saveAlbum(Album $album)
32 {
33     $data = array(
34         'artist' => $album->artist,
35         'title' => $album->title,
36     );
37
38     $id = (int) $album->id;
39     if ($id == 0) {
40         $this->tableGateway->insert($data);
41     } else {
42         if ($this->getAlbum($id)) {
43             $this->tableGateway->update($data, array('id' => $id));
44         } else {
45             throw new \Exception('Album id does not exist');
46         }
47     }
48 }
49
50 public function deleteAlbum($id)
51 {
52     $this->tableGateway->delete(array('id' => (int) $id));
53 }
54 }

```

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.

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.

```

1  namespace Album;
2
3  // Add these import statements:
4  use Album\Model\Album;
5  use Album\Model\AlbumTable;
6  use Zend\Db\ResultSet\ResultSet;
7  use Zend\Db\TableGateway\TableGateway;
8
9  class Module
10 {
11     // getAutoloaderConfig() and getConfig() methods here
12
13     // Add this method:
14     public function getServiceConfig()
15     {
16         return array(
17             'factories' => array(
18                 'Album\Model\AlbumTable' => function($sm) {
19                     $tableGateway = $sm->get('AlbumTableGateway');
20                     $table = new AlbumTable($tableGateway);
21                     return $table;
22                 },
23                 'AlbumTableGateway' => function ($sm) {
24                     $dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');
25                     $resultSetPrototype = new ResultSet();
26                     $resultSetPrototype->setArrayObjectPrototype(new Album());
27                     return new TableGateway('album', $dbAdapter, null,
28 =>$resultSetPrototype);
29                 },
30             ),
31         );
32     }

```

This method returns an array of factories that are all merged together by the ModuleManager before passing them 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:

```

1  return array(
2      'db' => array(
3          'driver'         => 'Pdo',
4          'dsn'            => 'mysql:dbname=zf2tutorial;host=localhost',

```

```

5         'driver_options' => array(
6             PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
7         ),
8     ),
9     'service_manager' => array(
10         'factories' => array(
11             'Zend\Db\Adapter\Adapter'
12                 => 'Zend\Db\Adapter\AdapterServiceFactory',
13         ),
14     ),
15 );

```

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):

```

1 return array(
2     'db' => array(
3         'username' => 'YOUR USERNAME HERE',
4         'password' => 'YOUR PASSWORD HERE',
5     ),
6 );

```

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:

```

1 // module/Album/src/Album/Controller/AlbumController.php:
2 public function getAlbumTable()
3 {
4     if (!$this->albumTable) {
5         $sm = $this->getServiceLocator();
6         $this->albumTable = $sm->get('Album\Model\AlbumTable');
7     }
8     return $this->albumTable;
9 }

```

You should also add:

```

1 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()`.

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:

```
1 // module/Album/src/Album/Controller/AlbumController.php:
2 // ...
3     public function indexAction()
4     {
5         return new ViewModel(array(
6             'albums' => $this->getAlbumTable()->fetchAll(),
7         ));
8     }
9 // ...
```

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:

```
1 <?php
2 // module/Album/view/album/album/index.phtml:
3
4 $title = 'My albums';
5 $this->headTitle($title);
6 ?>
7 <h1><?php echo $this->escapeHtml($title); ?></h1>
8 <p>
9     <a href="<?php echo $this->url('album', array('action'=>'add')); ?>">Add new album
10 </a>
11 </p>
12
13 <table class="table">
14 <tr>
15     <th>Title</th>
16     <th>Artist</th>
17     <th>&nbsp;</th>
18 </tr>
19 <?php foreach ($albums as $album) : ?>
20 <tr>
21     <td><?php echo $this->escapeHtml($album->title); ?></td>
22     <td><?php echo $this->escapeHtml($album->artist); ?></td>
23     <td>
24         <a href="<?php echo $this->url('album',
25             array('action'=>'edit', 'id' => $album->id)); ?>">Edit</a>
26         <a href="<?php echo $this->url('album',
27             array('action'=>'delete', 'id' => $album->id)); ?>">Delete</a>
28     </td>
29 </tr>
30 <?php endforeach; ?>
31 </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`.

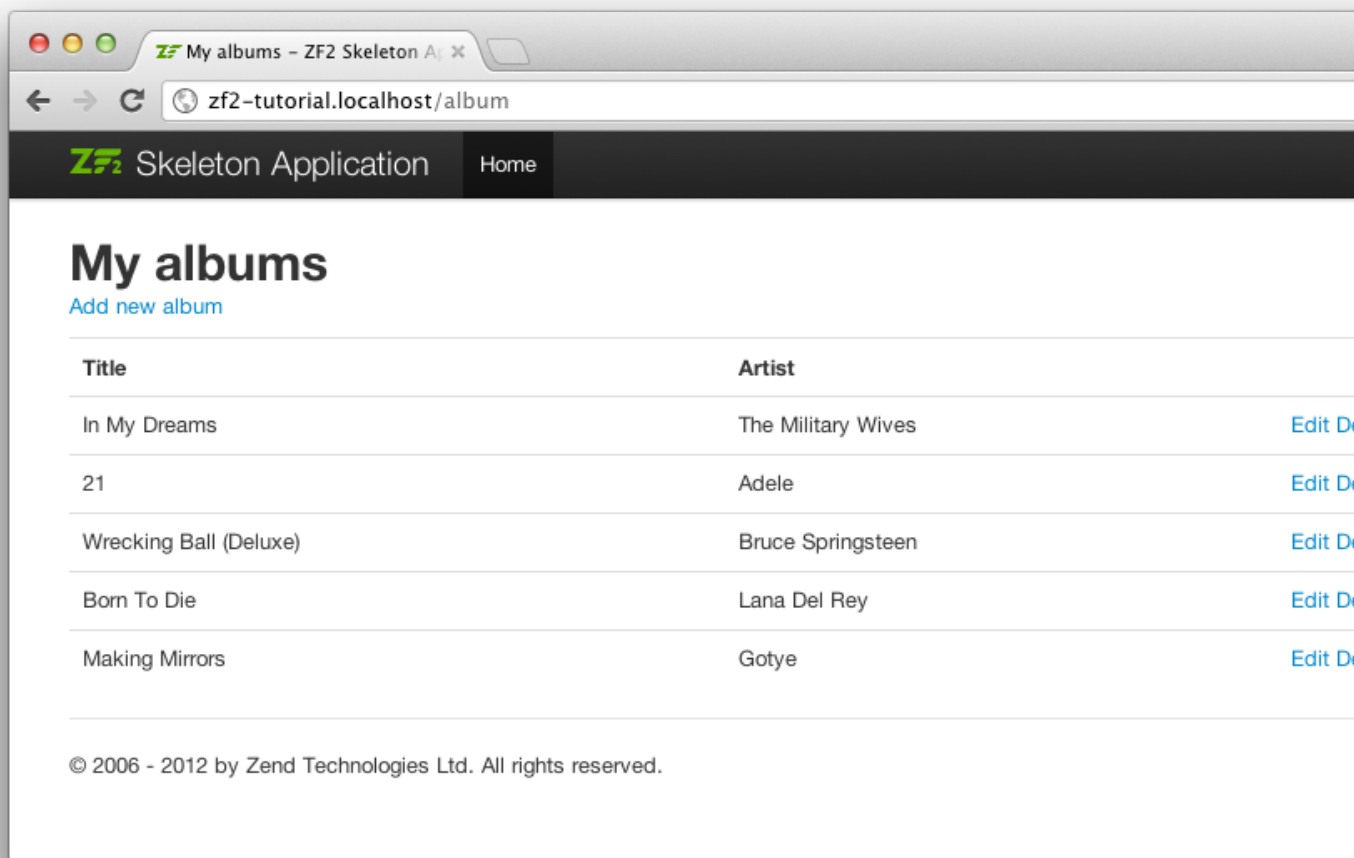
We iterate over the `$albums` that we assigned from the controller action. The Zend Framework 2 view system automatically ensures that these variables are extracted into the scope of the view script, so that we don't have to

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).

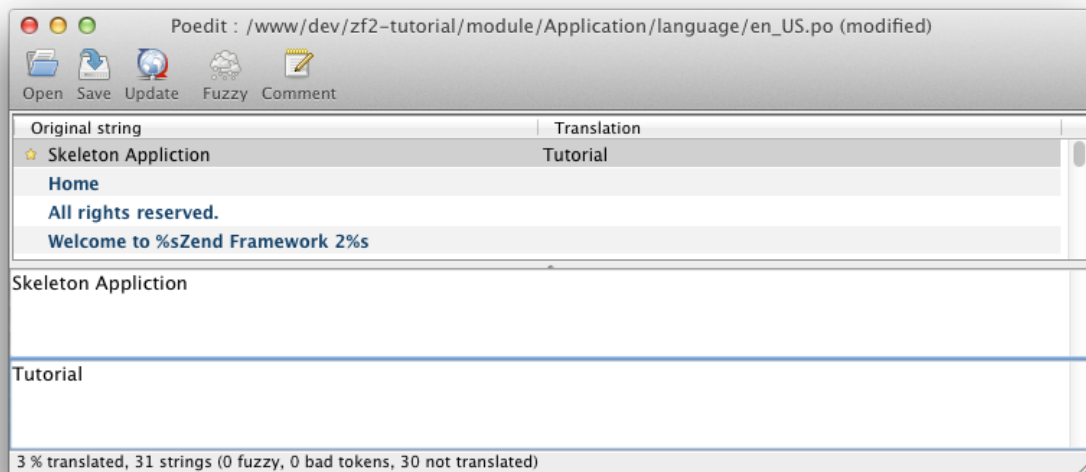
If you open `http://zf2-tutorial.localhost/album` you should see this:



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 `module/Application/language`, and you need to use [poedit](#) to change the text. Start poedit and open `module/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.

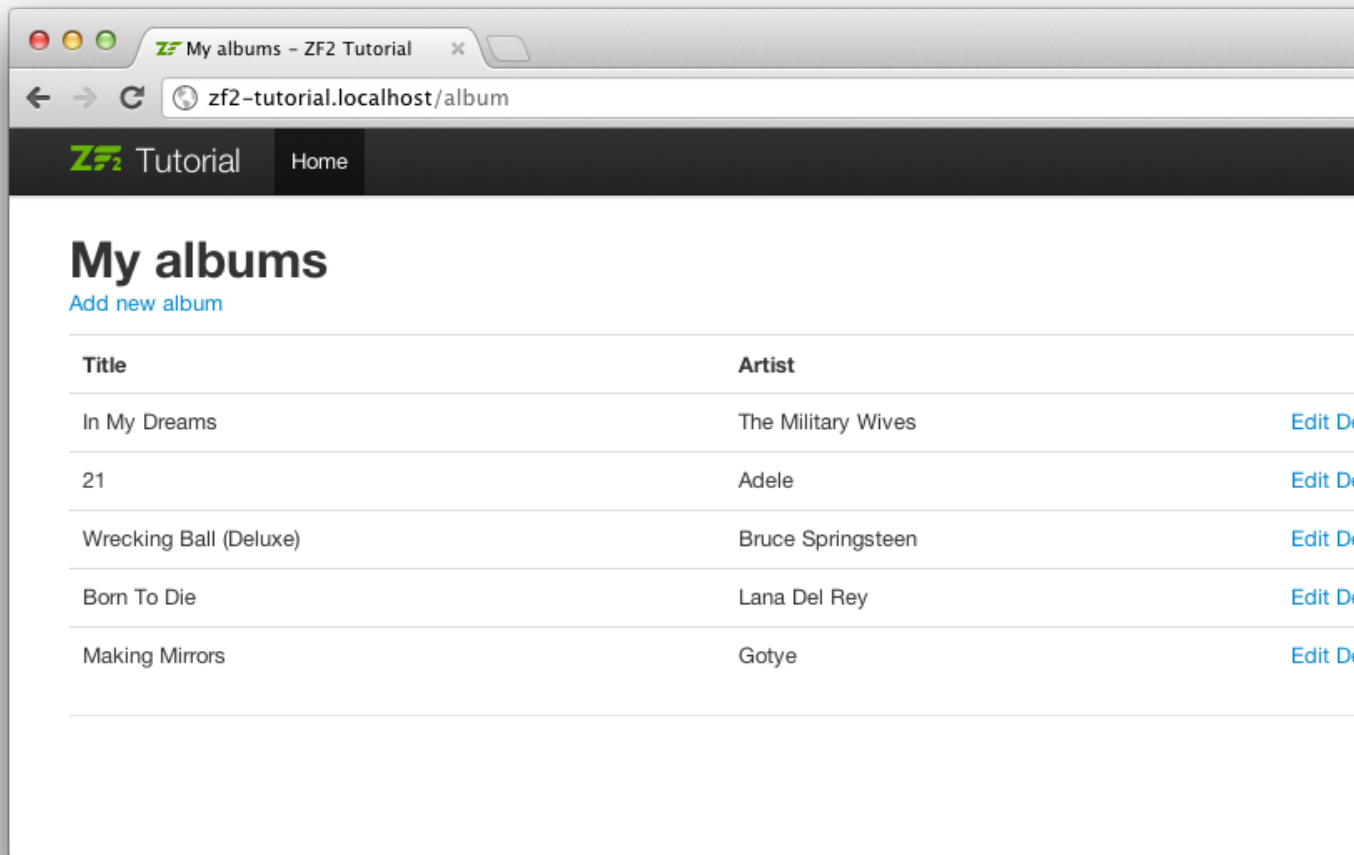
To remove the copyright message, we need to edit the Application module’s `layout.phtml` view script:

```

1 // module/Application/view/layout/layout.phtml:
2 // Remove this line:
3 <p>&copy; 2005 - 2014 by Zend Technologies Ltd. <?php echo $this->translate('All
4 rights reserved.') ?></p>

```

The page now looks ever so slightly better now!



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, form 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`:

```
1  namespace Album\Form;
2
3  use Zend\Form\Form;
4
5  class AlbumForm extends Form
6  {
7      public function __construct($name = null)
8      {
9          // we want to ignore the name passed
10         parent::__construct('album');
11
12         $this->add(array(
13             'name' => 'id',
14             'type' => 'Hidden',
15         ));
16         $this->add(array(
17             'name' => 'title',
18             'type' => 'Text',
19             'options' => array(
20                 'label' => 'Title',
21             ),
22         ));
```

```
23         $this->add(array(  
24             'name' => 'artist',  
25             'type' => 'Text',  
26             'options' => array(  
27                 'label' => 'Artist',  
28             ),  
29         ));  
30         $this->add(array(  
31             'name' => 'submit',  
32             'type' => 'Submit',  
33             'attributes' => array(  
34                 'value' => 'Go',  
35                 'id' => 'submitButton',  
36             ),  
37         ));  
38     }  
39 }
```

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 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.

Note: HTML-Forms can be sent using POST and GET. ZF2s default is POST, therefore you don't have to be explicit in setting this option. If you want to change it to GET though, all you have to do is set the specific attribute in the constructor.

```
$this->setAttribute('method', 'GET');
```

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`:

```
1  namespace Album\Model;  
2  
3  // Add these import statements  
4  use Zend\InputFilter\InputFilter;  
5  use Zend\InputFilter\InputFilterAwareInterface;  
6  use Zend\InputFilter\InputFilterInterface;  
7  
8  class Album implements InputFilterAwareInterface  
9  {  
10     public $id;  
11     public $artist;  
12     public $title;  
13     protected $inputFilter; // <-- Add this variable  
14  
15     public function exchangeArray($data)  
16     {  
17         $this->id      = (isset($data['id'])) ? $data['id']      : null;  
18         $this->artist  = (isset($data['artist'])) ? $data['artist'] : null;  
19         $this->title   = (isset($data['title'])) ? $data['title'] : null;  
20     }  
21  
22     // Add content to these methods:  
23     public function setInputFilter(InputFilterInterface $inputFilter)
```

```

24     {
25         throw new \Exception("Not used");
26     }
27
28     public function getInputFilter()
29     {
30         if (!$this->inputFilter) {
31             $inputFilter = new InputFilter();
32
33             $inputFilter->add(array(
34                 'name'      => 'id',
35                 'required' => true,
36                 'filters' => array(
37                     array('name' => 'Int'),
38                 ),
39             ));
40
41             $inputFilter->add(array(
42                 'name'      => 'artist',
43                 'required' => true,
44                 'filters' => array(
45                     array('name' => 'StripTags'),
46                     array('name' => 'StringTrim'),
47                 ),
48                 'validators' => array(
49                     array(
50                         'name'      => 'StringLength',
51                         'options' => array(
52                             'encoding' => 'UTF-8',
53                             'min'      => 1,
54                             'max'      => 100,
55                         ),
56                     ),
57                 ),
58             ));
59
60             $inputFilter->add(array(
61                 'name'      => 'title',
62                 'required' => true,
63                 'filters' => array(
64                     array('name' => 'StripTags'),
65                     array('name' => 'StringTrim'),
66                 ),
67                 'validators' => array(
68                     array(
69                         'name'      => 'StringLength',
70                         'options' => array(
71                             'encoding' => 'UTF-8',
72                             'min'      => 1,
73                             'max'      => 100,
74                         ),
75                     ),
76                 ),
77             ));
78
79             $this->inputFilter = $inputFilter;
80         }
81     }

```

```

82         return $this->inputFilter;
83     }
84 }

```

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()`:

```

1  // module/Album/src/Album/Controller/AlbumController.php:
2
3  //...
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6  use Album\Model\Album;           // <-- Add this import
7  use Album\Form\AlbumForm;       // <-- Add this import
8  //...
9
10     // Add content to this method:
11     public function addAction()
12     {
13         $form = new AlbumForm();
14         $form->get('submit')->setValue('Add');
15
16         $request = $this->getRequest();
17         if ($request->isPost()) {
18             $album = new Album();
19             $form->setInputFilter($album->getInputFilter());
20             $form->setData($request->getPost());
21
22             if ($form->isValid()) {
23                 $album->exchangeArray($form->getData());
24                 $this->getAlbumTable()->saveAlbum($album);
25
26                 // Redirect to list of albums
27                 return $this->redirect()->toRoute('album');
28             }
29         }
30         return array('form' => $form);
31     }
32     //...

```

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

```

1  $form = new AlbumForm();
2  $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.

```

1  $request = $this->getRequest();
2  if ($request->isPost()) {
3      $album = new Album();
4      $form->setInputFilter($album->getInputFilter());
5      $form->setData($request->getPost());
6      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.

```

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

```

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

```

1  // Redirect to list of albums
2  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.

```

1  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.

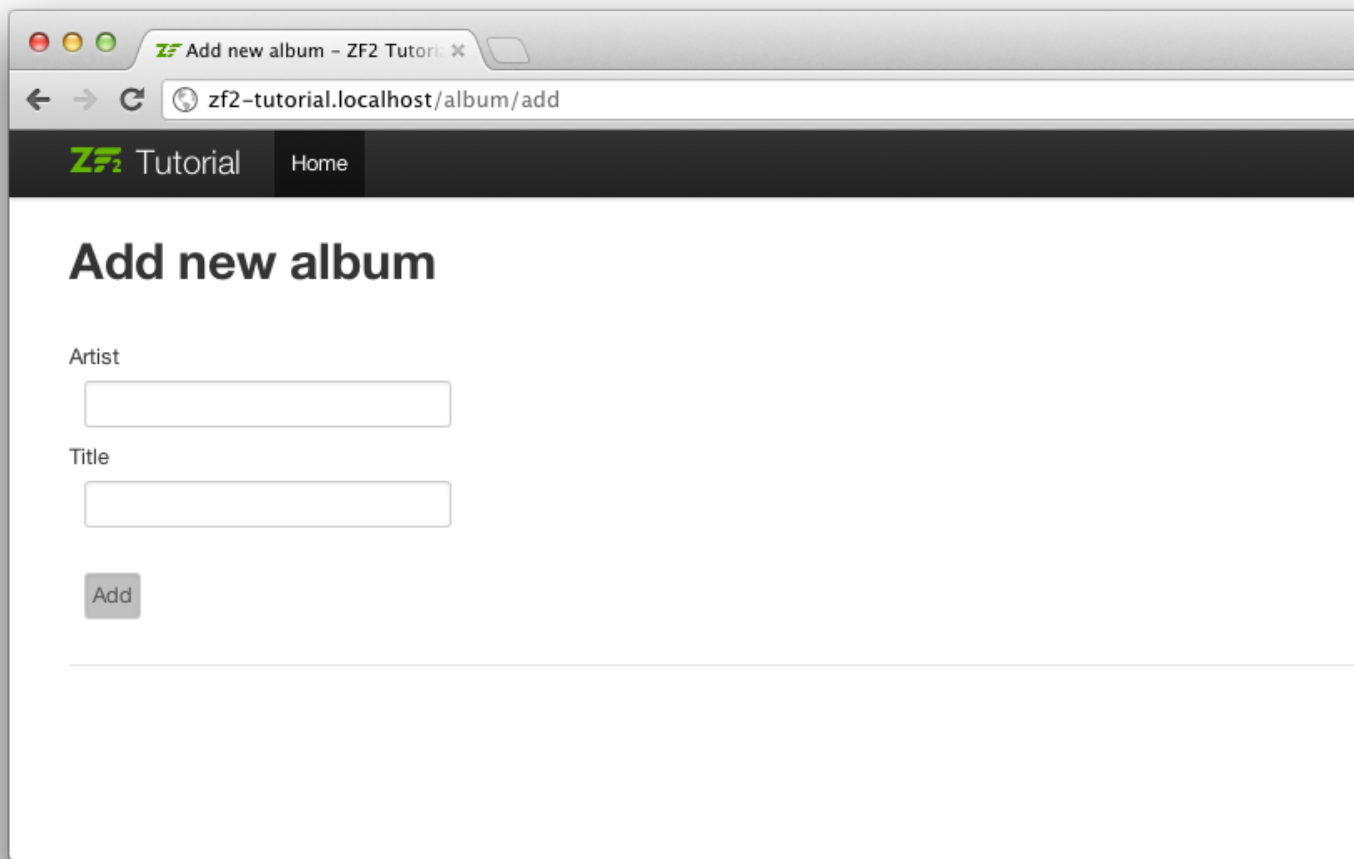
We now need to render the form in the `add.phtml` view script:

```

1  <?php
2  // module/Album/view/album/album/add.phtml:
3
4  $title = 'Add new album';
5  $this->headTitle($title);
6  ?>
7  <h1><?php echo $this->escapeHtml($title); ?></h1>
8  <?php
9  $form->setAttribute('action', $this->url('album', array('action' => 'add')));
10 $form->prepare();
11
12 echo $this->form()->openTag($form);
13 echo $this->formHidden($form->get('id'));
14 echo $this->formRow($form->get('title'));
15 echo $this->formRow($form->get('artist'));
16 echo $this->formSubmit($form->get('submit'));
17 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.

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`:


```

1 // module/Album/src/Album/Controller/AlbumController.php:
2 //...
3
4 // Add content to this method:
5 public function editAction()
6 {
7     $id = (int) $this->params()->fromRoute('id', 0);
8     if (!$id) {
9         return $this->redirect()->toRoute('album', array(
10             'action' => 'add'
11         ));
12     }
13
14     // Get the Album with the specified id. An exception is thrown
15     // if it cannot be found, in which case go to the index page.
16     try {
17         $album = $this->getAlbumTable()->getAlbum($id);
18     }
19     catch (\Exception $ex) {
20         return $this->redirect()->toRoute('album', array(
21             'action' => 'index'
22         ));
23     }
24
25     $form = new AlbumForm();
26     $form->bind($album);
27     $form->get('submit')->setAttribute('value', 'Edit');
28
29     $request = $this->getRequest();
30     if ($request->isPost()) {
31         $form->setInputFilter($album->getInputFilter());
32         $form->setData($request->getPost());
33
34         if ($form->isValid()) {
35             $this->getAlbumTable()->saveAlbum($album);
36
37             // Redirect to list of albums
38             return $this->redirect()->toRoute('album');
39         }
40     }
41
42     return array(
43         'id' => $id,
44         'form' => $form,
45     );
46 }
47 //...

```

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:

```

1 $id = (int) $this->params()->fromRoute('id', 0);
2 if (!$id) {
3     return $this->redirect()->toRoute('album', array(
4         'action' => 'add'
5     ));
6 }
7

```

```

8 // Get the album with the specified id. An exception is thrown
9 // if it cannot be found, in which case go to the index page.
10 try {
11     $album = $this->getAlbumTable()->getAlbum($id);
12 }
13 catch (\Exception $ex) {
14     return $this->redirect()->toRoute('album', array(
15         'action' => 'index'
16     ));
17 }

```

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.

```

1 $form = new AlbumForm();
2 $form->bind($album);
3 $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()`:

```

1 // module/Album/src/Album/Model/Album.php:
2 // ...
3 public function exchangeArray($data)
4 {
5     $this->id      = (isset($data['id'])) ? $data['id']      : null;
6     $this->artist  = (isset($data['artist'])) ? $data['artist'] : null;
7     $this->title   = (isset($data['title'])) ? $data['title'] : null;
8 }
9
10 // Add the following method:
11 public function getArrayCopy()
12 {
13     return get_object_vars($this);
14 }
15 // ...

```

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:

```

1 <?php
2 // module/Album/view/album/album/edit.phtml:
3
4 $title = 'Edit album';
5 $this->headTitle($title);

```

```

6  ?>
7  <h1><?php echo $this->escapeHtml($title); ?></h1>
8
9  <?php
10 $form = $this->form;
11 $form->setAttribute('action', $this->url(
12     'album',
13     array(
14         'action' => 'edit',
15         'id'     => $this->id,
16     )
17 ));
18 $form->prepare();
19
20 echo $this->form()->openTag($form);
21 echo $this->formHidden($form->get('id'));
22 echo $this->formRow($form->get('title'));
23 echo $this->formRow($form->get('artist'));
24 echo $this->formSubmit($form->get('submit'));
25 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.

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()`:

```

1  // module/Album/src/Album/Controller/AlbumController.php:
2  //...
3  // Add content to the following method:
4  public function deleteAction()
5  {
6      $id = (int) $this->params()->fromRoute('id', 0);
7      if (!$id) {
8          return $this->redirect()->toRoute('album');
9      }
10
11     $request = $this->getRequest();
12     if ($request->isPost()) {
13         $del = $request->getPost('del', 'No');
14
15         if ($del == 'Yes') {
16             $id = (int) $request->getPost('id');
17             $this->getAlbumTable()->deleteAlbum($id);
18         }
19
20         // Redirect to list of albums
21         return $this->redirect()->toRoute('album');

```

```

22     }
23
24     return array(
25         'id' => $id,
26         'album' => $this->getAlbumTable()->getAlbum($id)
27     );
28 }
29 //...

```

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:

```

1  <?php
2  // module/Album/view/album/album/delete.phtml:
3
4  $title = 'Delete album';
5  $this->headTitle($title);
6  ?>
7  <h1><?php echo $this->escapeHtml($title); ?></h1>
8
9  <p>Are you sure that you want to delete
10     '<?php echo $this->escapeHtml($album->title); ?>' by
11     '<?php echo $this->escapeHtml($album->artist); ?>'?
12 </p>
13 <?php
14 $url = $this->url('album', array(
15     'action' => 'delete',
16     'id' => $this->id,
17 ));
18 ?>
19 <form action="<?php echo $url; ?>" method="post">
20 <div>
21     <input type="hidden" name="id" value="<?php echo (int) $album->id; ?>" />
22     <input type="submit" name="del" value="Yes" />
23     <input type="submit" name="del" value="No" />
24 </div>
25 </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.

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:

```

1  'home' => array(
2      'type' => 'Zend\Mvc\Router\Http\Literal',
3      'options' => array(

```

```
4     'route'      => '/',
5     'defaults' => array(
6         'controller' => 'Application\Controller\Index',
7         'action'     => 'index',
8     ),
9 ),
10 ),
```

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

```
1  'home' => array(
2      'type' => 'Zend\Mvc\Router\Http\Literal',
3      'options' => array(
4          'route'      => '/',
5          'defaults' => array(
6              'controller' => 'Album\Controller\Album', // <-- change here
7              'action'     => 'index',
8          ),
9      ),
10 ),
```

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

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*.

Introducing our first “Blog” Module

Now that we know about the basics of the Zend Framework 2 Skeleton Application, let’s continue and create our very own module. We will create a module named “Blog”. This module will display a list of database entries that represent a single blog post. Each post will have three properties: `id`, `text` and `title`. We will create forms to enter new posts into our database and to edit existing posts. Furthermore we will do so by using best-practices throughout the whole QuickStart.

Writing a new Module

Let’s start by creating a new folder under the `/module` directory called `Blog`.

To be recognized as a module by the `ModuleManager` all we need to do is create a PHP class named `Module` under our module’s namespace, which is `Blog`. Create the file `/module/Blog/Module.php`

```
1 <?php
2 // Filename: /module/Blog/Module.php
3 namespace Blog;
4
5 class Module
6 {
7 }
```

We now have a module that can be detected by ZF2s `ModuleManager`. Let’s add this module to our application. Although our module doesn’t do anything yet, just having the `Module.php` class allows it to be loaded by ZF2s `ModuleManager`. To do this, add an entry for `Blog` to the modules array inside the main application config file at `/config/application.config.php`:

```
1 <?php
2 // Filename: /config/application.config.php
3 return array(
4     'modules' => array(
5         'Application',
6         'Blog'
```

```
7         ),
8
9         // ...
10    );
```

If you refresh your application you should see no change at all (but also no errors).

At this point it's worth taking a step back to discuss what modules are for. In short, a module is an encapsulated set of features for your application. A module might add features to the application that you can see, like our Blog module; or it might provide background functionality for other modules in the application to use, such as interacting with a third party API.

Organizing your code into modules makes it easier for you to reuse functionality in other application, or to use modules written by the community.

Configuring the Module

The next thing we're going to do is add a route to our application so that our module can be accessed through the URL `localhost:8080/blog`. We do this by adding router configuration to our module, but first we need to let the `ModuleManager` know that our module has configuration that it needs to load.

This is done by adding a `getConfig()` function to the `Module` class that returns the configuration. (This function is defined in the `ConfigProviderInterface` although actually implementing this interface in the module class is optional.) This function should return either an array or a `Traversable` object. Continue by editing your `/module/Blog/Module.php`:

```
1 <?php
2 // Filename: /module/Blog/Module.php
3 namespace Blog;
4
5 use Zend\ModuleManager\Feature\ConfigProviderInterface;
6
7 class Module implements ConfigProviderInterface
8 {
9     public function getConfig()
10     {
11         return array();
12     }
13 }
```

With this our `Module` is now able to be configured. Configuration files can become quite big though and keeping everything inside the `getConfig()` function won't be optimal. To help keep our project organized we're going to put our array configuration in a separate file. Go ahead and create this file at `/module/Blog/config/module.config.php`:

```
1 <?php
2 // Filename: /module/Blog/config/module.config.php
3 return array();
```

Now we will rewrite the `getConfig()` function to include this newly created file instead of directly returning the array.

```
1 <?php
2 // Filename: /module/Blog/Module.php
3 namespace Blog;
4
```

```

5 use Zend\ModuleManager\Feature\ConfigProviderInterface;
6
7 class Module implements ConfigProviderInterface
8 {
9     public function getConfig()
10     {
11         return include __DIR__ . '/config/module.config.php';
12     }
13 }

```

Reload your application and you'll see that everything remains as it was. Next we add the new route to our configuration file:

```

1 <?php
2 // Filename: /module/Blog/config/module.config.php
3 return array(
4     // This line opens the configuration for the RouteManager
5     'router' => array(
6         // Open configuration for all possible routes
7         'routes' => array(
8             // Define a new route called "post"
9             'post' => array(
10                 // Define the routes type to be "Zend\Mvc\Router\Http\Literal",
11                 // which is basically just a string
12                 'type' => 'literal',
13                 // Configure the route itself
14                 'options' => array(
15                     // Listen to "/blog" as uri
16                     'route' => '/blog',
17                     // Define default controller and action to be called when this
18                     // route is matched
19                     'defaults' => array(
20                         'controller' => 'Blog\Controller\List',
21                         'action' => 'index',
22                     )
23                 )
24             )
25         )
26     );

```

We've now created a route called post that listens to the URL localhost:8080/blog. Whenever someone accesses this route, the `indexAction()` function of the class `Blog\Controller\List` will be executed. However, this controller does not exist yet, so if you reload the page you will see this error message:

```

1 A 404 error occurred
2 Page not found.
3 The requested controller could not be mapped to an existing controller class.
4
5 Controller:
6 Blog\Controller\List(resolves to invalid controller class or alias:
7 Blog\Controller\List)
8 No Exception available

```

We now need to tell our module where to find this controller named `Blog\Controller\List`. To achieve this we have to add this key to the controllers configuration key inside your `/module/Blog/config/module.config.php`.

```

1 <?php
2 // Filename: /module/Blog/config/module.config.php
3 return array(
4     'controllers' => array(
5         'invokables' => array(
6             'Blog\Controller\List' => 'Blog\Controller\ListController'
7         )
8     ),
9     'router' => array( /** Route Configuration */ )
10 );

```

This configuration defines `Blog\Controller\List` as an alias for the `ListController` under the namespace `Blog\Controller`. Reloading the page should then give you:

```

1 ( ! ) Fatal error: Class 'Blog\Controller\ListController' not found in {libPath}/
↳ Zend/ServiceManager/AbstractPluginManager.php on line {lineNumber}

```

This error tells us that the application knows what class to load, but not where to find it. To fix this, we need to configure [autoloading](#) for our Module. Autoloading is a process to allow PHP to automatically load classes on demand. For our Module we set this up by adding a `getAutoloaderConfig()` function to our Module class. (This function is defined in the [AutoloaderProviderInterface](#), although the presence of the function is enough, actually implementing the interface is optional.)

```

1 <?php
2 // Filename: /module/Blog/Module.php
3 namespace Blog;
4
5 use Zend\ModuleManager\Feature\AutoloaderProviderInterface;
6 use Zend\ModuleManager\Feature\ConfigProviderInterface;
7
8 class Module implements
9     AutoloaderProviderInterface,
10    ConfigProviderInterface
11 {
12    /**
13     * Return an array for passing to Zend\Loader\AutoloaderFactory.
14     *
15     * @return array
16     */
17    public function getAutoloaderConfig()
18    {
19        return array(
20            'Zend\Loader\StandardAutoloader' => array(
21                'namespaces' => array(
22                    // Autoload all classes from namespace 'Blog' from '/module/Blog/
↳ src/Blog'
23                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
24                )
25            )
26        );
27    }
28
29    /**
30     * Returns configuration to merge with application configuration
31     *
32     * @return array|\Traversable
33     */

```

```

34     public function getConfig()
35     {
36         return include __DIR__ . '/config/module.config.php';
37     }
38 }

```

Now this looks like a lot of change but don't be afraid. We've added an `getAutoloaderConfig()` function which provides configuration for the `Zend\Loader\StandardAutoloader`. This configuration tells the application that classes in `__NAMESPACE__` (Blog) can be found inside `__DIR__ . '/src/' . __NAMESPACE__ (/module/Blog/src/Blog)`.

The `Zend\Loader\StandardAutoloader` uses a PHP community driven standard called *PSR-0* <<https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md>>_. Amongst other things, this standard defines a way for PHP to map class names to the file system. So with this configured, the application knows that our `Blog\Controller\ListController` class should exist at `/module/Blog/src/Blog/Controller/ListController.php`.

If you refresh the browser now you'll see the same error, as even though we've configured the autoloader, we still need to create the controller class. Let's create this file now:

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Controller/ListController.php
3 namespace Blog\Controller;
4
5 class ListController
6 {
7 }

```

Reloading the page now will finally result into a new screen. The new error message looks like this:

```

1 A 404 error occurred
2 Page not found.
3 The requested controller was not dispatchable.
4
5 Controller:
6 Blog\Controller\List(resolves to invalid controller class or alias:
↳ Blog\Controller\List)
7
8 Additional information:
9 Zend\Mvc\Exception\InvalidControllerException
10
11 File:
12 {libraryPath}/Zend/Mvc/Controller/ControllerManager.php:{lineNumber}
13 Message:
14 Controller of type Blog\Controller\ListController is invalid; must implement
↳ Zend\Stdlib\DispatchableInterface

```

This happens because our controller must implement `ZendStdlibDispatchableInterface` in order to be 'dispatched' (or run) by ZendFramework's MVC layer. ZendFramework provides some base controller implementation of it with `AbstractActionController`, which we are going to use. Let's modify our controller now:

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Controller/ListController.php
3 namespace Blog\Controller;
4
5 use Zend\Mvc\Controller\AbstractActionController;
6
7 class ListController extends AbstractActionController

```

```
{
}
```

It's now time for another refresh of the site. You should now see a new error message:

```
An error occurred
An error occurred during execution; please try again later.

Additional information:
Zend\View\Exception\RuntimeException

File:
{libraryPath}/library/Zend/View/Renderer/PhpRenderer.php:{lineNumber}
Message:
Zend\View\Renderer\PhpRenderer::render: Unable to render template "blog/list/index";
↪resolver could not resolve to a file
```

Now the application tells you that a view template-file can not be rendered, which is to be expected as we've not created it yet. The application is expecting it to be at `/module/Blog/view/blog/list/index.phtml`. Create this file and add some dummy content to it:

```
<!-- Filename: /module/Blog/view/blog/list/index.phtml -->
<h1>Blog\ListController::indexAction()</h1>
```

Before we continue let us quickly take a look at where we placed this file. Note that view files are found within the `/view` subdirectory, not `/src` as they are not PHP class files, but template files for rendering HTML. The following path however deserves some explanation but it's very simple. First we have the lowercased namespace. Followed by the lowercased controller name without the appendix 'controller' and lastly comes the name of the action that we are accessing, again without the appendix 'action'. All in all it looks like this: `/view/{namespace}/{controller}/{action}.phtml`. This has become a community standard but can potentially be changed by you at any time.

However creating this file alone is not enough and this brings us to the final topic of this part of the QuickStart. We need to let the application know where to look for view files. We do this within our modules configuration file `module.config.php`.

```
<?php
// Filename: /module/Blog/config/module.config.php
return array(
    'view_manager' => array(
        'template_path_stack' => array(
            __DIR__ . '/../view',
        ),
    ),
    'controllers' => array( /** Controller Configuration */),
    'router'       => array( /** Route Configuration */ )
);
```

The above configuration tells the application that the folder `/module/Blog/view` has view files in it that match the above described default scheme. It is important to note that with this you can not only ship view files for your module but you can also overwrite view files from other modules.

Reload your site now. Finally we are at a point where we see something different than an error being displayed. Congratulations, not only have you created a simple "Hello World" style module, you also learned about many error messages and their causes. If we didn't exhaust you too much, continue with our QuickStart and let's create a module that actually does something.

Introducing Services and the ServiceManager

In the previous chapter we've learned how to create a simple "Hello World" Application in Zend Framework 2. This is a good start and easy to understand but the application itself doesn't really do anything. In this chapter we will introduce you into the concept of Services and with this the introduction to `Zend\ServiceManager\ServiceManager`.

What is a Service?

A Service is an object that executes complex application logic. It's the part of the application that wires all difficult stuff together and gives you easy to understand results.

For what we're trying to accomplish with our `Blog-Module` this means that we want to have a Service that will give us the data that we want. The Service will get it's data from some source and when writing the Service we don't really care about what the source actually is. The Service will be written against an `Interface` that we define and that future Data-Providers have to implement.

Writing the PostService

When writing a Service it is a common best-practice to define an `Interface` first. `Interfaces` are a good way to ensure that other programmers can easily build extensions for our Services using their own implementations. In other words, they can write Services that have the same function names but internally do completely different things but have the same specified result.

In our case we want to create a `PostService`. This means first we are going to define a `PostServiceInterface`. The task of our Service is to provide us with data of our blog posts. For now we are going to focus on the read-only side of things. We will define a function that will give us all posts and we will define a function that will give us a single post.

Let's start by creating the `Interface` at `/module/Blog/src/Blog/Service/PostServiceInterface.php`

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Service/PostServiceInterface.php
3 namespace Blog\Service;
4
5 use Blog\Model\PostInterface;
6
7 interface PostServiceInterface
8 {
9     /**
10      * Should return a set of all blog posts that we can iterate over. Single_
11      ↪entries of the array are supposed to be
12      * implementing \Blog\Model\PostInterface
13      *
14      * @return array|PostInterface[]
15      */
16     public function findAllPosts();
17
18     /**
19      * Should return a single blog post
20      *
21      * @param int $id Identifier of the Post that should be returned
22      * @return PostInterface
23      */
24     public function findPost($id);
25 }
```

As you can see we define two functions. The first being `findAllPosts()` that is supposed to return all posts and the second one being `findPost($id)` that is supposed to return the post matching the given identifier `$id`. What's new in here is the fact that we actually define a return value that doesn't exist yet. We make the assumption that the return value all in all are of type `Blog\Model\PostInterface`. We will define this class at a later point and for now we simply create the `PostService` first.

Create the class `PostService` at `/module/Blog/src/Blog/Service/PostService.php`, be sure to implement the `PostServiceInterface` and its required functions (we will fill in these functions later). You then should have a class that looks like the following:

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Service/PostService.php
3 namespace Blog\Service;
4
5 class PostService implements PostServiceInterface
6 {
7     /**
8      * {@inheritDoc}
9      */
10     public function findAllPosts()
11     {
12         // TODO: Implement findAllPosts() method.
13     }
14
15     /**
16      * {@inheritDoc}
17      */
18     public function findPost($id)
19     {
20         // TODO: Implement findPost() method.
21     }
22 }
```


Writing the required Model Files

Since our `PostService` will return Models, we should create them, too. Be sure to write an Interface for the Model first! Let's create `/module/Blog/src/Blog/Model/PostInterface.php` and `/module/Blog/src/Blog/Model/Post.php`. First the `PostInterface`:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Model/PostInterface.php
3  namespace Blog\Model;
4
5  interface PostInterface
6  {
7      /**
8       * Will return the ID of the blog post
9       *
10      * @return int
11      */
12     public function getId();
13
14     /**
15      * Will return the TITLE of the blog post
16      *
17      * @return string
18      */
19     public function getTitle();
20
21     /**
22      * Will return the TEXT of the blog post
23      *
24      * @return string
25      */
26     public function getText();
27 }
```

Notice that we only created getter-functions here. This is because right now we don't bother how the data gets inside the `Post`-class. All we care for is that we're able to access the properties through these getter-functions.

And now we'll create the appropriate Model file associated with the interface. Make sure to set the required class properties and fill the getter functions defined by our `PostInterface` with some useful content. Even if our interface doesn't care about setter functions we will write them as we will fill our class with data through these. You then should have a class that looks like the following:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Model/Post.php
3  namespace Blog\Model;
4
5  class Post implements PostInterface
6  {
7      /**
8       * @var int
9       */
10     protected $id;
11
12     /**
```

```
13     * @var string
14     */
15     protected $title;
16
17     /**
18     * @var string
19     */
20     protected $text;
21
22     /**
23     * {@inheritdoc}
24     */
25     public function getId()
26     {
27         return $this->id;
28     }
29
30     /**
31     * @param int $id
32     */
33     public function setId($id)
34     {
35         $this->id = $id;
36     }
37
38     /**
39     * {@inheritdoc}
40     */
41     public function getTitle()
42     {
43         return $this->title;
44     }
45
46     /**
47     * @param string $title
48     */
49     public function setTitle($title)
50     {
51         $this->title = $title;
52     }
53
54     /**
55     * {@inheritdoc}
56     */
57     public function getText()
58     {
59         return $this->text;
60     }
61
62     /**
63     * @param string $text
64     */
65     public function setText($text)
66     {
67         $this->text = $text;
68     }
69 }
```

Bringing Life into our PostService

Now that we have our Model files in place we can actually bring life into our `PostService` class. To keep the Service-Layer easy to understand for now we will only return some hard-coded content from our `PostService` class directly. Create a property inside the `PostService` called `$data` and make this an array of our Model type. Edit `PostService` like this:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Service/PostService.php
3  namespace Blog\Service;
4
5  class PostService implements PostServiceInterface
6  {
7      protected $data = array(
8          array(
9              'id'      => 1,
10             'title'   => 'Hello World #1',
11             'text'    => 'This is our first blog post!'
12         ),
13         array(
14             'id'      => 2,
15             'title'   => 'Hello World #2',
16             'text'    => 'This is our second blog post!'
17         ),
18         array(
19             'id'      => 3,
20             'title'   => 'Hello World #3',
21             'text'    => 'This is our third blog post!'
22         ),
23         array(
24             'id'      => 4,
25             'title'   => 'Hello World #4',
26             'text'    => 'This is our fourth blog post!'
27         ),
28         array(
29             'id'      => 5,
30             'title'   => 'Hello World #5',
31             'text'    => 'This is our fifth blog post!'
32         )
33     );
34
35     /**
36      * {@inheritdoc}
37      */
38     public function findAllPosts()
39     {
40         // TODO: Implement findAllPosts() method.
41     }
42
43     /**
44      * {@inheritdoc}
45      */
46     public function findPost($id)
47     {
48         // TODO: Implement findPost() method.
49     }
50 }
```

After we now have some data, let's modify our `find*()` functions to return the appropriate model files:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Service/PostService.php
3  namespace Blog\Service;
4
5  use Blog\Model\Post;
6
7  class PostService implements PostServiceInterface
8  {
9      protected $data = array(
10         array(
11             'id'      => 1,
12             'title'   => 'Hello World #1',
13             'text'    => 'This is our first blog post!'
14         ),
15         array(
16             'id'      => 2,
17             'title'   => 'Hello World #2',
18             'text'    => 'This is our second blog post!'
19         ),
20         array(
21             'id'      => 3,
22             'title'   => 'Hello World #3',
23             'text'    => 'This is our third blog post!'
24         ),
25         array(
26             'id'      => 4,
27             'title'   => 'Hello World #4',
28             'text'    => 'This is our fourth blog post!'
29         ),
30         array(
31             'id'      => 5,
32             'title'   => 'Hello World #5',
33             'text'    => 'This is our fifth blog post!'
34         )
35     );
36
37     /**
38      * {@inheritDoc}
39      */
40     public function findAllPosts()
41     {
42         $allPosts = array();
43
44         foreach ($this->data as $index => $post) {
45             $allPosts[] = $this->findPost($index);
46         }
47
48         return $allPosts;
49     }
50
51     /**
52      * {@inheritDoc}
53      */
54     public function findPost($id)
55     {
56         $postData = $this->data[$id];
57     }

```

```

58     $model = new Post();
59     $model->setId($postData['id']);
60     $model->setTitle($postData['title']);
61     $model->setText($postData['text']);
62
63     return $model;
64 }
65 }

```

As you can see, both our functions now have appropriate return values. Please note that from a technical point of view the current implementation is far from perfect. We will improve this Service a lot in the future but for now we have a working Service that is able to give us some data in a way that is defined by our `PostServiceInterface`.

Bringing the Service into the Controller

Now that we have our `PostService` written, we want to get access to this Service in our Controllers. For this task we will step foot into a new topic called “Dependency Injection”, short “DI”.

When we’re talking about dependency injection we’re talking about a way to get dependencies into our classes. The most common form, “Constructor Injection”, is used for all dependencies that are required by a class at all times.

In our case we want to have our `Blog-Modules ListController` somehow interact with our `PostService`. This means that the class `PostService` is a dependency of the class `ListController`. Without the `PostService` our `ListController` will not be able to function properly. To make sure that our `ListController` will always get the appropriate dependency, we will first define the dependency inside the `ListController`’s constructor function `__construct()`. Go on and modify the `ListController` like this:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Controller/ListController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Mvc\Controller\AbstractActionController;
7
8  class ListController extends AbstractActionController
9  {
10     /**
11      * @var \Blog\Service\PostServiceInterface
12      */
13     protected $postService;
14
15     public function __construct(PostServiceInterface $postService)
16     {
17         $this->postService = $postService;
18     }
19 }

```

As you can see our `__construct()` function now has a required argument. We will not be able to call this class anymore without passing it an instance of a class that matches our defined `PostServiceInterface`. If you were to go back to your browser and reload your project with the url `localhost:8080/blog`, you’d see the following error message:

```

1  ( ! ) Catchable fatal error: Argument 1 passed to Blog\Controller>ListController::__
   ↪construct()
2      must be an instance of Blog\Service\PostServiceInterface, none given,

```

```

3      called in {libraryPath}\Zend\ServiceManager\AbstractPluginManager.php on line
    ↪{lineNumber}
4      and defined in \module\Blog\src\Blog\Controller\ListController.php on line 15

```

And this error message is expected. It tells you exactly that our `ListController` expects to be passed an implementation of the `PostServiceInterface`. So how do we make sure that our `ListController` will receive such an implementation? To solve this, we need to tell the application how to create instances of the `Blog\Controller\ListController`. If you remember back to when we created the controller, we added an entry to the `invokables` array in the module config:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'view_manager' => array( /** ViewManager Config */ ),
5      'controllers' => array(
6          'invokables' => array(
7              'Blog\Controller\List' => 'Blog\Controller\ListController'
8          )
9      ),
10     'router' => array( /** Router Config */ )
11 );

```

An invokable is a class that can be constructed without any arguments. Since our `Blog\Controller\ListController` now has a required argument, we need to change this. The `ControllerManager`, which is responsible for instantiating controllers, also support using factories. A factory is a class that creates instances of another class. We'll now create one for our `ListController`. Let's modify our configuration like this:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'view_manager' => array( /** ViewManager Config */ ),
5      'controllers' => array(
6          'factories' => array(
7              'Blog\Controller\List' => 'Blog\Factory\ListControllerFactory'
8          )
9      ),
10     'router' => array( /** Router Config */ )
11 );

```

As you can see we no longer have the key `invokables`, instead we now have the key `factories`. Furthermore the value of our controller name `Blog\Controller\List` has been changed to not match the class `Blog\Controller\ListController` directly but to rather call a class called `Blog\Factory\ListControllerFactory`. If you refresh your browser you'll see a different error message:

```

1  An error occurred
2  An error occurred during execution; please try again later.
3
4  Additional information:
5  Zend\ServiceManager\Exception\ServiceNotCreatedException
6
7  File:
8  {libraryPath}\Zend\ServiceManager\AbstractPluginManager.php:{lineNumber}
9
10 Message:
11 While attempting to create blogcontrollerlist(alias: Blog\Controller\List) an_
    ↪invalid factory was registered for this instance type.

```

This message should be quite easy to understand. The `Zend\Mvc\Controller\ControllerManager` is accessing `Blog\Controller\List`, which internally is saved as `blogcontrollerlist`. While it does so it notices that a factory class is supposed to be called for this controller name. However, it doesn't find this factory class so to the Manager it is an invalid factory. Using easy words: the Manager doesn't find the Factory class so that's probably where our error lies. And of course, we have yet to write the factory, so let's go ahead and do this.

Writing a Factory Class

Factory classes within Zend Framework 2 always need to implement the `Zend\ServiceManager\FactoryInterface`. Implementing this class lets the `ServiceManager` know that the function `createService()` is supposed to be called. And `createService()` actually expects to be passed an instance of the `ServiceLocatorInterface` so the `ServiceManager` will always inject this using Dependency Injection as we have learned above. Let's implement our factory class:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Factory/ListControllerFactory.php
3  namespace Blog\Factory;
4
5  use Blog\Controller\ListController;
6  use Zend\ServiceManager\FactoryInterface;
7  use Zend\ServiceManager\ServiceLocatorInterface;
8
9  class ListControllerFactory implements FactoryInterface
10 {
11     /**
12      * Create service
13      *
14      * @param ServiceLocatorInterface $serviceLocator
15      *
16      * @return mixed
17      */
18     public function createService(ServiceLocatorInterface $serviceLocator)
19     {
20         $realServiceLocator = $serviceLocator->getServiceLocator();
21         $postService        = $realServiceLocator->get(
22             'Blog\Service\PostServiceInterface');
23
24         return new ListController($postService);
25     }
26 }
```

Now this looks complicated! Let's start to look at the `$realServiceLocator`. When using a Factory-Class that will be called from the `ControllerManager` it will actually inject **itself** as the `$serviceLocator`. However, we need the real `ServiceManager` to get to our Service-Classes. This is why we call the function `getServiceLocator()` who will give us the real `ServiceManager`.

After we have the `$realServiceLocator` set up we try to get a Service called `Blog\Service\PostServiceInterface`. This name that we're accessing is supposed to return a Service that matches the `PostServiceInterface`. This Service is then passed along to the `ListController` which will directly be returned.

Note though that we have yet to register a Service called `Blog\Service\PostServiceInterface`. There's no magic happening that does this for us just because we give the Service the name of an Interface. Refresh your browser and you will see this error message:

```

1  An error occurred
2  An error occurred during execution; please try again later.
3
4  Additional information:
5  Zend\ServiceManager\Exception\ServiceNotFoundException
6
7  File:
8  {libraryPath}\Zend\ServiceManager\ServiceManager.php:{lineNumber}
9
10 Message:
11 Zend\ServiceManager\ServiceManager::get was unable to fetch or create an instance_
    ↪for Blog\Service\PostServiceInterface

```

Exactly what we expected. Somewhere in our application - currently our factory class - a service called `Blog\Service\PostServiceInterface` is requested but the `ServiceManager` doesn't know about this Service yet. Therefore it isn't able to create an instance for the requested name.

Registering Services

Registering a Service is as simple as registering a Controller. All we need to do is modify our `module.config.php` and add a new key called `service_manager` that then has `invokables` and `factories`, too, the same way like we have it inside our `controllers` array. Check out the new configuration file:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'service_manager' => array(
5          'invokables' => array(
6              'Blog\Service\PostServiceInterface' => 'Blog\Service\PostService'
7          )
8      ),
9      'view_manager' => array( /** View Manager Config */ ),
10     'controllers' => array( /** Controller Config */ ),
11     'router' => array( /** Router Config */ )
12 );

```

As you can see we now have added a new Service that listens to the name `Blog\Service\PostServiceInterface` and points to our own implementation which is `Blog\Service\PostService`. Since our Service has no dependencies we are able to add this Service under the `invokables` array. Try refreshing your browser. You should see no more error messages but rather exactly the page that we have created in the previous chapter of the Tutorial.

Using the Service at our Controller

Let's now use the `PostService` within our `ListController`. For this we will need to overwrite the default `indexAction()` and return the values of our `PostService` into the view. Modify the `ListController` like this:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Controller/ListController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;

```



```

6 use Zend\Mvc\Controller\AbstractActionController;
7 use Zend\View\Model\ViewModel;
8
9 class ListController extends AbstractActionController
10 {
11     /**
12      * @var \Blog\Service\PostServiceInterface
13      */
14     protected $postService;
15
16     public function __construct(PostServiceInterface $postService)
17     {
18         $this->postService = $postService;
19     }
20
21     public function indexAction()
22     {
23         return new ViewModel(array(
24             'posts' => $this->postService->findAllPosts()
25         ));
26     }
27 }

```

First please note that our controller imported another class. We need to import `Zend\View\Model\ViewModel`, which usually is what your Controllers will return. When returning an instance of a `ViewModel` you're able to always assign so called View-Variables. In this case we have assigned a variable called `$posts` with the value of whatever the function `findAllPosts()` of our `PostService` returns. In our case it is an array of `Blog\Model\Post` classes. Refreshing the browser won't change anything yet because we obviously need to modify our view-file to be able to display the data we want to.

Note: You do not actually need to return an instance of `ViewModel`. When you return a normal php array it will internally be converted into a `ViewModel`. So in short:

```
return new ViewModel(array('foo' => 'bar'));
```

equals

```
return array('foo' => 'bar');
```

Accessing View Variables

When pushing variables to the view they are accessible through two ways. Either directly like `$this->posts` or implicitly like `$posts`. Both are the same, however, calling `$posts` implicitly will result in a little round-trip through the `__call()` function.

Let's modify our view to display a table of all blog posts that our `PostService` returns.

```

1 <!-- Filename: /module/Blog/view/blog/list/index.phtml -->
2 <h1>Blog</h1>
3
4 <?php foreach ($this->posts as $post): ?>
5 <article>
6     <h1 id="post<?= $post->getId() ?>"><?= $post->getTitle() ?></h1>
7     <p>
8         <?= $post->getText() ?>

```

```
9         </p>
10     </article>
11 <?php endforeach ?>
```

In here we simply run a `foreach` over the array `$this->posts`. Since every single entry of our array is of type `Blog\Model\Post` we can use the respective getter functions to receive the data we want to get.

Summary

And with this the current chapter is finished. We now have learned how to interact with the `ServiceManager` and we also know what dependency injection is all about. We are now able to pass variables from our services into the view through a controller and we know how to iterate over arrays inside a view-script.

In the next chapter we will take a first look at the things we should do when we want to get data from a database.

Preparing for different Database-Backends

In the previous chapter we have created a `PostService` that returns some data from blog posts. While this served an easy to understand learning purpose it is quite impractical for real world applications. No one would want to modify the source files each time a new post is added. But luckily we all know about databases. All we need to learn is how to interact with databases from our ZF2 application.

But there is a catch. There are many database backend systems, namely SQL and NoSQL databases. While in a real-world you would probably jump right to the solution that fits you the most at the time being, it is a better practice to create another layer in front of the actual database access that abstracts the database interaction. We call this the **Mapper-Layer**.

What is database abstraction?

The term “database abstraction” may sound quite confusing but this is actually a very simple thing. Consider a SQL and a NoSQL database. Both have methods for CRUD (Create, Read, Update, Delete) operations. For example to query the database against a given row in MySQL you’d do a `mysqli_query('SELECT foo FROM bar')`. But using an ORM for MongoDB for example you’d do something like `$mongoODM->getRepository('bar')->find('foo')`. Both engines would give you the same result but the execution is different.

So if we start using a SQL database and write those codes directly into our `PostService` and a year later we decide to switch to a NoSQL database, we would literally have to delete all previously coded lines and write new ones. And in a few years later a new thing pops up and we have to delete and re-write codes again. This isn’t really the best approach and that’s precisely where database abstraction or the Mapper-Layer comes in handy.

Basically what we do is to create a new Interface. This interface then defines **how** our database interaction should function but the actual implementation is left out. But let’s stop the theory and go over to code this thing.

Creating the PostMapperInterface

Let’s first think a bit about what possible database interactions we can think of. We need to be able to:

- find a single blog post
- find all blog posts
- insert new blog post
- update existing blog posts
- delete existing blog posts

Those are the most important ones I'd guess for now. Considering `insert()` and `update()` both write into the database it'd be nice to have just a single `save()` -function that calls the proper function internally.

Start by creating a new file inside a new namespace `Blog\Mapper` called `PostMapperInterface.php` and add the following content to it.

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Mapper/PostMapperInterface.php
3 namespace Blog\Mapper;
4
5 use Blog\Model\PostInterface;
6
7 interface PostMapperInterface
8 {
9     /**
10      * @param int|string $id
11      * @return PostInterface
12      * @throws \InvalidArgumentException
13      */
14     public function find($id);
15
16     /**
17      * @return array|PostInterface[]
18      */
19     public function findAll();
20 }
```

As you can see we define two different functions. We say that a mapper-implementation is supposed to have one `find()` -function that returns a single object implementing the `PostInterface`. Then we want to have one function called `findAll()` that returns an array of objects implementing the `PostInterface`. Definitions for a possible `save()` or `delete()` functionality will not be added to the interface yet since we'll only be looking at the read-only side of things for now. They will be added at a later point though!

Refactoring the PostService

Now that we have defined how our mapper should act we can make use of it inside our `PostService`. To start off the refactoring process let's empty our class and delete all current content. Then implement the functions defined by the `PostServiceInterface` and you should have an empty `PostService` that looks like this:

The first thing we need to keep in mind is that this interface isn't implemented in our `PostService` but is rather used as a dependency. A required dependency, therefore we need to create a `__construct()` that takes any implementation of this interface as a parameter. Also you should create a protected variable to store the parameter into.

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Service/PostService.php
3 namespace Blog\Service;
4
5 use Blog\Mapper\PostMapperInterface;
```

```

6
7 class PostService implements PostServiceInterface
8 {
9     /**
10      * @var \Blog\Mapper\PostMapperInterface
11      */
12     protected $postMapper;
13
14     /**
15      * @param PostMapperInterface $postMapper
16      */
17     public function __construct(PostMapperInterface $postMapper)
18     {
19         $this->postMapper = $postMapper;
20     }
21
22     /**
23      * {@inheritDoc}
24      */
25     public function findAllPosts()
26     {
27     }
28
29     /**
30      * {@inheritDoc}
31      */
32     public function findPost($id)
33     {
34     }
35 }

```

With this we now require an implementation of the `PostMapperInterface` for our `PostService` to function. Since none exists yet we can not get our application to work and we'll be seeing the following PHP error:

```

1 Catchable fatal error: Argument 1 passed to Blog\Service\PostService::__construct()
2 must implement interface Blog\Mapper\PostMapperInterface, none given,
3 called in {path}\module\Blog\src\Blog\Service\PostServiceFactory.php on line 19
4 and defined in {path}\module\Blog\src\Blog\Service\PostService.php on line 17

```

But the power of what we're doing lies within assumptions that we **can** make. This `PostService` will always have a mapper passed as an argument. So in our `find*()`-functions we **can** assume that it is there. Recall that the `PostMapperInterface` defines a `find($id)` and a `findAll()` function. Let's use those within our Service-functions:

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Service/PostService.php
3 namespace Blog\Service;
4
5 use Blog\Mapper\PostMapperInterface;
6
7 class PostService implements PostServiceInterface
8 {
9     /**
10      * @var \Blog\Mapper\PostMapperInterface
11      */
12     protected $postMapper;
13
14     /**

```

```
15     * @param PostMapperInterface $postMapper
16     */
17     public function __construct(PostMapperInterface $postMapper)
18     {
19         $this->postMapper = $postMapper;
20     }
21
22     /**
23     * {@inheritDoc}
24     */
25     public function findAllPosts()
26     {
27         return $this->postMapper->findAll();
28     }
29
30     /**
31     * {@inheritDoc}
32     */
33     public function findPost($id)
34     {
35         return $this->postMapper->find($id);
36     }
37 }
```

Looking at this code you'll see that we use the `postMapper` to get access to the data we want. How this is happening isn't the business of the `PostService` anymore. But the `PostService` does know what data it will receive and that's the only important thing.

The PostService has a dependency

Now that we have introduced the `PostMapperInterface` as a dependency for the `PostService` we are no longer able to define this service as an `invokable` because it has a dependency. So we need to create a factory for the service. Do this by creating a factory the same way we have done for the `ListController`. First change the configuration from an `invokables`-entry to a `factories`-entry and assign the proper factory class:

```
1 <?php
2 // Filename: /module/Blog/config/module.config.php
3 return array(
4     'service_manager' => array(
5         'factories' => array(
6             'Blog\Service\PostServiceInterface' => 'Blog\Factory\PostServiceFactory'
7         )
8     ),
9     'view_manager' => array( /** ViewManager Config */ ),
10    'controllers' => array( /** ControllerManager Config */ ),
11    'router' => array( /** Router Config */ )
12 );
```

Going by the above configuration we now need to create the class `Blog\Factory\PostServiceFactory` so let's go ahead and create it:

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Factory/PostServiceFactory.php
3 namespace Blog\Factory;
4
```

```

5  use Blog\Service\PostService;
6  use Zend\ServiceManager\FactoryInterface;
7  use Zend\ServiceManager\ServiceLocatorInterface;
8
9  class PostServiceFactory implements FactoryInterface
10 {
11     /**
12      * Create service
13      *
14      * @param ServiceLocatorInterface $serviceLocator
15      * @return mixed
16      */
17     public function createService(ServiceLocatorInterface $serviceLocator)
18     {
19         return new PostService(
20             $serviceLocator->get('Blog\Mapper\PostMapperInterface')
21         );
22     }
23 }

```

With this in place you should now be able to see the `ServiceNotFoundException`, thrown by the `ServiceManager`, saying that the requested service cannot be found.

```

1  Additional information:
2  Zend\ServiceManager\Exception\ServiceNotFoundException
3  File:
4  {libraryPath}\Zend\ServiceManager\ServiceManager.php:529
5  Message:
6  Zend\ServiceManager\ServiceManager::get was unable to fetch or create an instance_
  ↳for Blog\Mapper\PostMapperInterface

```

Conclusion

We finalize this chapter with the fact that we successfully managed to keep the database-logic outside of our service. Now we are able to implement different database solution depending on our need and change them easily when the time requires it.

In the next chapter we will create the actual implementation of our `PostMapperInterface` using `Zend\Db\Sql`.

Introducing Zend\Db\Sql and Zend\Stdlib\Hydrator

In the last chapter we have introduced the mapping layer and created the `PostMapperInterface`. Now it is time to create an implementation of this interface so that we can make use of our `PostService` again. As an introductory example we will be using the `Zend\Db\Sql` classes. So let's jump right into it.

Preparing the Database

Before we can start using a database we should prepare one. In this example we'll be using a MySQL-Database called `blog` which is accessible on the `localhost`. The database will have one table called `posts` with three columns `id`, `title` and `text` with the `id` being the primary key. For demo purpose, please use this database-dump.

```
1 CREATE TABLE posts (  
2     id int(11) NOT NULL auto_increment,  
3     title varchar(100) NOT NULL,  
4     text TEXT NOT NULL,  
5     PRIMARY KEY (id)  
6 );  
7  
8 INSERT INTO posts (title, text)  
9     VALUES ('Blog #1', 'Welcome to my first blog post');  
10 INSERT INTO posts (title, text)  
11     VALUES ('Blog #2', 'Welcome to my second blog post');  
12 INSERT INTO posts (title, text)  
13     VALUES ('Blog #3', 'Welcome to my third blog post');  
14 INSERT INTO posts (title, text)  
15     VALUES ('Blog #4', 'Welcome to my fourth blog post');  
16 INSERT INTO posts (title, text)  
17     VALUES ('Blog #5', 'Welcome to my fifth blog post');
```

Quick Facts Zend\Db\Sql

To create queries against a database using Zend\Db\Sql you need to have a database connection available. This connection is served through any class implementing the Zend\Db\Adapter\AdapterInterface. The most handy way to create such a class is through the use of the Zend\Db\Adapter\AdapterServiceFactory which listens to the config-key db. Let's start by creating the required configuration entries and modify your module.config.php adding a new top-level key called db:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'db' => array(
5          'driver'           => 'Pdo',
6          'username'        => 'SECRET_USERNAME', //edit this
7          'password'        => 'SECRET_PASSWORD', //edit this
8          'dsn'             => 'mysql:dbname=blog;host=localhost',
9          'driver_options'  => array(
10             \PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
11         )
12     ),
13     'service_manager' => array( /** ServiceManager Config */ ),
14     'view_manager'   => array( /** ViewManager Config */ ),
15     'controllers'    => array( /** ControllerManager Config */ ),
16     'router'         => array( /** Router Config */ )
17 );

```

As you can see we've added the db-key and inside we create the parameters required to create a driver instance.

Note: One important thing to note is that in general you **do not** want to have your credentials inside the normal configuration file but rather in a local configuration file like /config/autoload/db.local.php, that will **not** be pushed to servers using zend-skeletons .gitignore file. Keep this in mind when you share your codes!

Taking this example you would have this file:

```

1  <?php
2  // Filename: /config/autoload/db.local.php
3  return array(
4      'db' => array(
5          'driver'           => 'Pdo',
6          'username'        => 'SECRET_USERNAME', //edit this
7          'password'        => 'SECRET_PASSWORD', //edit this
8          'dsn'             => 'mysql:dbname=blog;host=localhost',
9          'driver_options'  => array(
10             \PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
11         )
12     ),
13 );

```

The next thing we need to do is by making use of the AdapterServiceFactory. This is a ServiceManager entry that will look like the following:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'db' => array(
5          'driver'           => 'Pdo',

```

```

6      'username'      => 'SECRET_USERNAME', //edit this
7      'password'     => 'SECRET_PASSWORD', //edit this
8      'dsn'          => 'mysql:dbname=blog;host=localhost',
9      'driver_options' => array(
10         \PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
11     )
12 ),
13 'service_manager' => array(
14     'factories' => array(
15         'Blog\Service\PostServiceInterface' => 'Blog\Factory\PostServiceFactory',
16         'Zend\Db\Adapter\Adapter'           =>
17         ↪ 'Zend\Db\Adapter\AdapterServiceFactory'
18     )
19 ),
19 'view_manager'     => array( /** ViewManager Config */ ),
20 'controllers'      => array( /** ControllerManager Config */ ),
21 'router'           => array( /** Router Config */ )
22 );

```

Note the new Service that we called `Zend\Db\Adapter\Adapter`. Calling this Service will now always give back a running instance of the `Zend\Db\Adapter\AdapterInterface` depending on what driver we assign.

With the adapter in place we're now able to run queries against the database. The construction of queries is best done through the “QueryBuilder” features of `Zend\Db\Sql` which are `Zend\Db\Sql\Sql` for select queries, `Zend\Db\Sql\Insert` for insert queries, `Zend\Db\Sql\Update` for update queries and `Zend\Db\Sql\Delete` for delete queries. The basic workflow of these components is:

1. Build a query using `Sql`, `Insert`, `Update` or `Delete`
2. Create an `Sql-Statement` from the `Sql` object
3. Execute the query
4. Do something with the result

Knowing this we can now write the implementation for the `PostMapperInterface`.

Writing the mapper implementation

Our mapper implementation will reside inside the same namespace as its interface. Go ahead and create a class called `ZendDbSqlMapper` and implement the `PostMapperInterface`.

Now recall what we have learned earlier. For `Zend\Db\Sql` to function we will need a working implementation of the `AdapterInterface`. This is a requirement and therefore will be injected using constructor-injection. Create a `__construct()` function that accepts an `AdapterInterface` as parameter and store it within the class.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6  use Zend\Db\Adapter\AdapterInterface;
7
8  class ZendDbSqlMapper implements PostMapperInterface
9  {
10     /**
11      * @var \Zend\Db\Adapter\AdapterInterface
12      */

```

```

13     protected $dbAdapter;
14
15     /**
16      * @param AdapterInterface $dbAdapter
17      */
18     public function __construct(AdapterInterface $dbAdapter)
19     {
20         $this->dbAdapter = $dbAdapter;
21     }
22
23     /**
24      * @param int|string $id
25      *
26      * @return PostInterface
27      * @throws \InvalidArgumentException
28      */
29     public function find($id)
30     {
31     }
32
33     /**
34      * @return array|PostInterface[]
35      */
36     public function findAll()
37     {
38     }
39 }

```

As you know from previous chapters, whenever we have a required parameter we need to write a factory for the class. Go ahead and create a factory for our mapper implementation.

We're now able to register our mapper implementation as a service. If you recall from the previous chapter, or if you were to look at the current error message, you'll note that we call the Service `Blog\Mapper\PostMapperInterface` to get a mapper implementation. Modify the configuration so that this key will call the newly called factory class.

```

1 <?php
2 // Filename: /module/Blog/config/module.config.php
3 return array(
4     'db' => array( /** Db Config */ ),
5     'service_manager' => array(
6         'factories' => array(
7             'Blog\Mapper\PostMapperInterface' =>
8             ↪ 'Blog\Factory\ZendDbSqlMapperFactory',
9             'Blog\Service\PostServiceInterface' =>
10            ↪ 'Blog\Service\Factory\PostServiceFactory',
11             'Zend\Db\Adapter\Adapter' =>
12            ↪ 'Zend\Db\Adapter\AdapterServiceFactory'
13         )
14     ),
15     'view_manager' => array( /** ViewManager Config */ ),
16     'controllers' => array( /** ControllerManager Config */ ),
17     'router' => array( /** Router Config */ )
18 );

```

With the adapter in place you're now able to refresh the blog index at `localhost:8080/blog` and you'll notice that the `ServiceNotFoundException` is gone and we get the following PHP Warning:

```

1 Warning: Invalid argument supplied for foreach() in /module/Blog/view/blog/list/
  ↳ index.phtml on line 13
2 ID Text Title

```

This is due to the fact that our mapper doesn't return anything yet. Let's modify the `findAll()` function to return all blogs from the database table.

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3 namespace Blog\Mapper;
4
5 use Zend\Db\Adapter\AdapterInterface;
6
7 class ZendDbSqlMapper implements PostMapperInterface
8 {
9     /**
10      * @var \Zend\Db\Adapter\AdapterInterface
11      */
12     protected $dbAdapter;
13
14     /**
15      * @param AdapterInterface $dbAdapter
16      */
17     public function __construct(AdapterInterface $dbAdapter)
18     {
19         $this->dbAdapter = $dbAdapter;
20     }
21
22     /**
23      * @param int|string $id
24      *
25      * @return \Blog\Entity\PostInterface
26      * @throws \InvalidArgumentException
27      */
28     public function find($id)
29     {
30     }
31
32     /**
33      * @return array|\Blog\Entity\PostInterface[]
34      */
35     public function findAll()
36     {
37         $sql = new Sql($this->dbAdapter);
38         $select = $sql->select('posts');
39
40         $stmt = $sql->prepareStatementForSqlObject($select);
41         $result = $stmt->execute();
42
43         return $result;
44     }
45 }

```

The above code should look fairly straight forward to you. Sadly, though, a refresh of the application reveals another error message.

Let's not return the `$result` variable for now and do a dump of it to see what we get here. Change the `findAll()` function and do a data dumping of the `$result` variable:

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3 namespace Blog\Mapper;
4
5 use Blog\Model\PostInterface;
6 use Zend\Db\Adapter\AdapterInterface;
7 use Zend\Db\Sql\Sql;
8
9 class ZendDbSqlMapper implements PostMapperInterface
10 {
11     /**
12      * @var \Zend\Db\Adapter\AdapterInterface
13      */
14     protected $dbAdapter;
15
16     /**
17      * @param AdapterInterface $dbAdapter
18      */
19     public function __construct(AdapterInterface $dbAdapter)
20     {
21         $this->dbAdapter = $dbAdapter;
22     }
23
24     /**
25      * @param int|string $id
26      *
27      * @return PostInterface
28      * @throws \InvalidArgumentException
29      */
30     public function find($id)
31     {
32     }
33
34     /**
35      * @return array|PostInterface[]
36      */
37     public function findAll()
38     {
39         $sql = new Sql($this->dbAdapter);
40         $select = $sql->select('posts');
41
42         $stmt = $sql->prepareStatementForSqlObject($select);
43         $result = $stmt->execute();
44
45         \Zend\Debug\Debug::dump($result); die();
46     }
47 }

```

Refreshing the application you should now see the following output:

```

1 object (Zend\Db\Adapter\Driver\Pdo\Result)#303 (8) {
2     ["statementMode":protected] => string(7) "forward"
3     ["resource":protected] => object (PDOStatement)#296 (1) {
4         ["queryString"] => string(29) "SELECT `posts`. * FROM `posts`"
5     }
6     ["options":protected] => NULL
7     ["currentComplete":protected] => bool(false)
8     ["currentData":protected] => NULL

```

```

9      ["position":protected] => int(-1)
10     ["generatedValue":protected] => string(1) "0"
11     ["rowCount":protected] => NULL
12 }
    
```

As you can see we do not get any data returned. Instead we are presented with a dump of some `Result` object that appears to have no data in it whatsoever. But this is a faulty assumption. This `Result` object only has information available for you when you actually try to access it. To make use of the data within the `Result` object the best approach would be to pass the `Result` object over into a `ResultSet` object, as long as the query was successful.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6  use Zend\Db\Adapter\AdapterInterface;
7  use Zend\Db\Adapter\Driver\ResultInterface;
8  use Zend\Db\ResultSet\ResultSet;
9  use Zend\Db\Sql\Sql;
10
11  class ZendDbSqlMapper implements PostMapperInterface
12  {
13      /**
14       * @var \Zend\Db\Adapter\AdapterInterface
15       */
16      protected $dbAdapter;
17
18      /**
19       * @param AdapterInterface $dbAdapter
20       */
21      public function __construct(AdapterInterface $dbAdapter)
22      {
23          $this->dbAdapter = $dbAdapter;
24      }
25
26      /**
27       * @param int|string $id
28       *
29       * @return PostInterface
30       * @throws \InvalidArgumentException
31       */
32      public function find($id)
33      {
34      }
35
36      /**
37       * @return array|PostInterface[]
38       */
39      public function findAll()
40      {
41          $sql = new Sql($this->dbAdapter);
42          $select = $sql->select('posts');
43
44          $stmt = $sql->prepareStatementForSqlObject($select);
45          $result = $stmt->execute();
46
47          if ($result instanceof ResultInterface && $result->isQueryResult()) {
48              $resultSet = new ResultSet();
    
```

```

49         \Zend\Debug\Debug::dump($resultSet->initialize($result));die();
50     }
51
52     die("no data");
53 }
54
55 }

```

Refreshing the page you should now see the dump of a ResultSet object that has a property ["count":protected] => int(5). Meaning we have five rows inside our database.

```

1  object (Zend\Db\ResultSet\ResultSet)#304 (8) {
2      ["allowedReturnTypes":protected] => array(2) {
3          [0] => string(11) "arrayobject"
4          [1] => string(5) "array"
5      }
6      ["arrayObjectPrototype":protected] => object (ArrayObject)#305 (1) {
7          ["storage":"ArrayObject":private] => array(0) {
8          }
9      }
10     ["returnType":protected] => string(11) "arrayobject"
11     ["buffer":protected] => NULL
12     ["count":protected] => int(2)
13     ["dataSource":protected] => object (Zend\Db\Adapter\Driver\Pdo\Result)#303 (8) {
14         ["statementMode":protected] => string(7) "forward"
15         ["resource":protected] => object (PDOStatement)#296 (1) {
16             ["queryString"] => string(29) "SELECT `posts`.* FROM `posts`"
17         }
18         ["options":protected] => NULL
19         ["currentComplete":protected] => bool(false)
20         ["currentData":protected] => NULL
21         ["position":protected] => int(-1)
22         ["generatedValue":protected] => string(1) "0"
23         ["rowCount":protected] => int(2)
24     }
25     ["fieldCount":protected] => int(3)
26     ["position":protected] => int(0)
27 }

```

Another very interesting property is ["returnType":protected] => string(11) "arrayobject". This tells us that all database entries will be returned as an ArrayObject. And this is a little problem as the PostMapperInterface requires us to return an array of PostInterface objects. Luckily there is a very simple option for us available to make this happen. In the examples above we have used the default ResultSet object. There is also a HydratingResultSet which will hydrate the given data into a provided object.

This means: if we tell the HydratingResultSet to use the database data to create Post objects for us, then it will do exactly this. Let's modify our code:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6  use Zend\Db\Adapter\AdapterInterface;
7  use Zend\Db\Adapter\Driver\ResultInterface;
8  use Zend\Db\ResultSet\HydratingResultSet;
9  use Zend\Db\Sql\Sql;
10

```



```

11 class ZendDbSqlMapper implements PostMapperInterface
12 {
13     /**
14      * @var \Zend\Db\Adapter\AdapterInterface
15      */
16     protected $dbAdapter;
17
18     /**
19      * @param AdapterInterface $dbAdapter
20      */
21     public function __construct(AdapterInterface $dbAdapter)
22     {
23         $this->dbAdapter = $dbAdapter;
24     }
25
26     /**
27      * @param int|string $id
28      *
29      * @return PostInterface
30      * @throws \InvalidArgumentException
31      */
32     public function find($id)
33     {
34     }
35
36     /**
37      * @return array|PostInterface[]
38      */
39     public function findAll()
40     {
41         $sql = new Sql($this->dbAdapter);
42         $select = $sql->select('posts');
43
44         $stmt = $sql->prepareStatementForSqlObject($select);
45         $result = $stmt->execute();
46
47         if ($result instanceof ResultInterface && $result->isQueryResult()) {
48             $resultSet = new HydratingResultSet(new \Zend\Stdlib\Hydrator\ClassMethods(), new \Blog\Model\Post());
49
50             return $resultSet->initialize($result);
51         }
52
53         return array();
54     }
55 }

```

We have changed a couple of things here. Firstly instead of a normal `ResultSet` we are using the `HydratingResultSet`. This Object requires two parameters, the second one being the object to hydrate into and the first one being the hydrator that will be used. A hydrator, in short, is an object that changes any sort of data from one format to another. The InputFormat that we have is an `ArrayObject` but we want `Post-Models`. The `ClassMethods`-hydrator will take care of this using the setter- and getter functions of our `Post-model`.

Instead of dumping the `$result` variable we now directly return the initialized `HydratingResultSet` so we'll be able to access the data stored within. In case we get something else returned that is not an instance of a `ResultInterface` we return an empty array.

Refreshing the page you will now see all your blog posts listed on the page. Great!

Refactoring hidden dependencies

There's one little thing that we have done that's not a best-practice. We use both a Hydrator and an Object inside our ZendDbSqlMapper

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6  use Zend\Db\Adapter\AdapterInterface;
7  use Zend\Db\Adapter\Driver\ResultInterface;
8  use Zend\Db\ResultSet\HydratingResultSet;
9  use Zend\Db\Sql\Sql;
10 use Zend\Stdlib\Hydrator\HydratorInterface;
11
12 class ZendDbSqlMapper implements PostMapperInterface
13 {
14     /**
15      * @var \Zend\Db\Adapter\AdapterInterface
16      */
17     protected $dbAdapter;
18
19     /**
20      * @var \Zend\Stdlib\Hydrator\HydratorInterface
21      */
22     protected $hydrator;
23
24     /**
25      * @var \Blog\Model\PostInterface
26      */
27     protected $postPrototype;
28
29     /**
30      * @param AdapterInterface $dbAdapter
31      * @param HydratorInterface $hydrator
32      * @param PostInterface $postPrototype
33      */
34     public function __construct(
35         AdapterInterface $dbAdapter,
36         HydratorInterface $hydrator,
37         PostInterface $postPrototype
38     ) {
39         $this->dbAdapter      = $dbAdapter;
40         $this->hydrator        = $hydrator;
41         $this->postPrototype   = $postPrototype;
42     }
43
44     /**
45      * @param int|string $id
46      *
47      * @return PostInterface
48      * @throws \InvalidArgumentException
49      */
50     public function find($id)
51     {
52     }
53

```

```

54     /**
55      * @return array|PostInterface[]
56      */
57     public function findAll()
58     {
59         $sql = new Sql($this->dbAdapter);
60         $select = $sql->select('posts');
61
62         $stmt = $sql->prepareStatementForSqlObject($select);
63         $result = $stmt->execute();
64
65         if ($result instanceof ResultInterface && $result->isQueryResult()) {
66             $resultSet = new HydratingResultSet($this->hydrator, $this->
↪postPrototype);
67
68             return $resultSet->initialize($result);
69         }
70
71         return array();
72     }
73 }

```

Now that our mapper requires more parameters we need to update the `ZendDbSqlMapperFactory` and inject those parameters.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Factory/ZendDbSqlMapperFactory.php
3  namespace Blog\Factory;
4
5  use Blog\Mapper\ZendDbSqlMapper;
6  use Blog\Model\Post;
7  use Zend\ServiceManager\FactoryInterface;
8  use Zend\ServiceManager\ServiceLocatorInterface;
9  use Zend\Stdlib\Hydrator\ClassMethods;
10
11  class ZendDbSqlMapperFactory implements FactoryInterface
12  {
13      /**
14       * Create service
15       *
16       * @param ServiceLocatorInterface $serviceLocator
17       *
18       * @return mixed
19       */
20      public function createService(ServiceLocatorInterface $serviceLocator)
21      {
22          return new ZendDbSqlMapper(
23              $serviceLocator->get('Zend\Db\Adapter\Adapter'),
24              new ClassMethods(false),
25              new Post()
26          );
27      }
28  }

```

With this in place you can refresh the application again and you'll see your blog posts listed once again. Our Mapper has now a really good architecture and no more hidden dependencies.

Finishing the mapper

Before we jump into the next chapter let's quickly finish the mapper by writing an implementation for the `find()` method.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6  use Zend\Db\Adapter\AdapterInterface;
7  use Zend\Db\Adapter\Driver\ResultInterface;
8  use Zend\Db\ResultSet\HydratingResultSet;
9  use Zend\Db\Sql\Sql;
10 use Zend\Stdlib\Hydrator\HydratorInterface;
11
12 class ZendDbSqlMapper implements PostMapperInterface
13 {
14     /**
15      * @var \Zend\Db\Adapter\AdapterInterface
16      */
17     protected $dbAdapter;
18
19     /**
20      * @var \Zend\Stdlib\Hydrator\HydratorInterface
21      */
22     protected $hydrator;
23
24     /**
25      * @var \Blog\Model\PostInterface
26      */
27     protected $postPrototype;
28
29     /**
30      * @param AdapterInterface $dbAdapter
31      * @param HydratorInterface $hydrator
32      * @param PostInterface $postPrototype
33      */
34     public function __construct(
35         AdapterInterface $dbAdapter,
36         HydratorInterface $hydrator,
37         PostInterface $postPrototype
38     ) {
39         $this->dbAdapter = $dbAdapter;
40         $this->hydrator = $hydrator;
41         $this->postPrototype = $postPrototype;
42     }
43
44     /**
45      * @param int|string $id
46      *
47      * @return PostInterface
48      * @throws \InvalidArgumentException
49      */
50     public function find($id)
51     {
52         $sql = new Sql($this->dbAdapter);
53         $select = $sql->select('posts');
```

```

54         $select->where(array('id = ?' => $id));
55
56         $stmt    = $sql->prepareStatementForSqlObject($select);
57         $result  = $stmt->execute();
58
59         if ($result instanceof ResultInterface && $result->isQueryResult() &&
↪$result->getAffectedRows()) {
60             return $this->hydrator->hydrate($result->current(), $this->
↪postPrototype);
61         }
62
63         throw new \InvalidArgumentException("Blog with given ID:{$id} not found.");
64     }
65
66     /**
67      * @return array|PostInterface[]
68      */
69     public function findAll()
70     {
71         $sql    = new Sql($this->dbAdapter);
72         $select = $sql->select('posts');
73
74         $stmt    = $sql->prepareStatementForSqlObject($select);
75         $result  = $stmt->execute();
76
77         if ($result instanceof ResultInterface && $result->isQueryResult()) {
78             $resultSet = new HydratingResultSet($this->hydrator, $this->
↪postPrototype);
79
80             return $resultSet->initialize($result);
81         }
82
83         return array();
84     }
85 }

```

The `find()` function looks really similar to the `findAll()` function. There's just three simple differences. Firstly we need to add a condition to the query to only select one row. This is done using the `where()` function of the `Sql` object. Then we also check if the `$result` has a row in it through `getAffectedRows()`. The return statement then will be hydrated using the injected hydrator into the prototype that has also been injected.

This time, when we do not find a row we will throw an `\InvalidArgumentException` so that the application will easily be able to handle the scenario.

Conclusion

Finishing this chapter you now know how to query for data using the `Zend\Db\Sql` classes. You have also learned about the `Zend\Stdlib\Hydrator-Component` which is one of the new key components of ZF2. Furthermore you have once again proven that you are able to manage proper dependency injection.

In the next chapter we'll take a closer look at the router so we'll be able to do some more action within our Module.

Understanding the Router

Right now we have a pretty solid set up for our module. However, we're not really doing all too much yet, to be precise, all we do is display all `Blog` entries on one page. In this chapter you will learn everything you need to know about the `Router` to create other routes to be able to display only a single blog, to add new blogs to your application and to edit and delete existing blogs.

Different route types

Before we go into details on our application, let's take a look at the most important route types that Zend Framework offers.

Zend\Mvc\Router\Http\Literal

The first common route type is the `Literal-Route`. As mentioned in a previous chapter a literal route is one that matches a specific string. Examples for URLs that are usually literal routes are:

- `http://domain.com/blog`
- `http://domain.com/blog/add`
- `http://domain.com/about-me`
- `http://domain.com/my/very/deep/page`
- `http://domain.com/my/very/deep/page`

Configuration for a literal route requires you to set up the route that should be matched and needs you to define some defaults to be used, for example which controller and which action to call. A simple configuration for a literal route looks like this:

```
1 'router' => array(  
2     'routes' => array(  
3         'about' => array(  
4             'type' => 'literal',
```

```
5         'options' => array(
6             'route' => '/about-me',
7             'defaults' => array(
8                 'controller' => 'AboutMeController',
9                 'action' => 'aboutme',
10            ),
11        ),
12    ),
13 )
14 )
```

Zend\Mvc\Router\Http\Segment

The second most commonly used route type is the `Segment-Route`. A segmented route is used for whenever your url is supposed to contain variable parameters. Pretty often those parameters are used to identify certain objects within your application. Some examples for URLs that contain parameters and are usually segment routes are:

Configuring a `Segment-Route` takes a little more effort but isn't difficult to understand. The tasks you have to do are similar at first, you have to define the route-type, just be sure to make it `Segment`. Then you have to define the route and add parameters to it. Then as usual you define the defaults to be used, the only thing that differs in this part is that you can assign defaults for your parameters, too. The new part that is used on routes of the `Segment` type is to define so called `constraints`. They are used to tell the Router what “rules” are given for parameters. For example, an `id-parameter` is only allowed to be of type `integer`, the `year-parameter` is only allowed to be of type `integer` and may only contain exactly four digits. A sample configuration can look like this:

```
1  'router' => array(
2      'routes' => array(
3          'archives' => array(
4              'type' => 'segment',
5              'options' => array(
6                  'route' => '/news/archive/:year',
7                  'defaults' => array(
8                      'controller' => 'ArchiveController',
9                      'action' => 'byYear',
10                 ),
11                  'constraints' => array(
12                      'year' => '\d{4}'
13                 )
14             ),
15         ),
16     )
17 )
```

This configuration defines a route for a URL like `domain.com/news/archive/2014`. As you can see, our route now contains the part `:year`. This is called a route-parameter. Route parameters for `Segment-Routes` are defined by a full-colon (“:”) in front of a string; the string is the parameter name.

Under `constraints` you see that we have another array. This array contains regular expression rules for each parameter of your route. In our example case the regex uses two parts, the first one being `\d` which means “a digit”, so any number from 0-9. The second part is `{4}` which means that the part before this has to match exactly four times. So in easy words we say “four digits”.

If now you call the URL `domain.com/news/archive/123`, the router will not match the URL because we only support years with four digits.

You may notice that we did not define any defaults for the parameter `year`. This is because the parameter is currently set up as a `required` parameter. If a parameter is supposed to be `optional` we need to define this inside

the route definition. This is done by adding square brackets around the parameter. Let's modify the above example route to have the `year` parameter optional and use the current year as default:

```

1  'router' => array(
2      'routes' => array(
3          'archives' => array(
4              'type' => 'segment',
5              'options' => array(
6                  'route' => '/news/archive[:year]',
7                  'defaults' => array(
8                      'controller' => 'ArchiveController',
9                      'action' => 'byYear',
10                     'year' => date('Y')
11                 ),
12                 'constraints' => array(
13                     'year' => '\d{4}'
14                 )
15             ),
16         )
17     )
18 )

```

Notice that now we have a part in our route that is optional. Not only the parameter `year` is optional. The slash that is separating the `year` parameter from the URL string `archive` is optional, too, and may only be there whenever the `year` parameter is present.

Different routing concepts

When thinking about the whole application it becomes clear that there are a lot of routes to be matched. When writing these routes you have two options. One option is to spend less time writing routes that in turn are a little slow in matching. Another option is to write very explicit routes that match a little faster but require more work to define. Let's take a look at both of them.

Generic routes

A generic route is one that matches many URLs. You may remember this concept from Zend Framework 1 where basically you didn't even bother about routes because we had one "god route" that was used for everything. You define the controller, the action, and all parameters within just one single route.

The big advantage of this approach is the immense time you save when developing your application. The downside, however, is that matching such a route can take a little bit longer due to the fact that so many variables need to be checked. However, as long as you don't overdo it, this is a viable concept. For this reason the `ZendSkeletonApplication` uses a very generic route, too. Let's take a look at a generic route:

```

1  'router' => array(
2      'routes' => array(
3          'default' => array(
4              'type' => 'segment',
5              'options' => array(
6                  'route' => '/[:controller[:action]]',
7                  'defaults' => array(
8                      '__NAMESPACE__' => 'Application\Controller',
9                      'controller' => 'Index',
10                     'action' => 'index',

```

```
11         ),
12         'constraints' => [
13             'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
14             'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
15         ]
16     ),
17 )
18 )
19 )
```

Let's take a closer look as to what has been defined in this configuration. The `route` part now contains two optional parameters, `controller` and `action`. The `action` parameter is optional only when the `controller` parameter is present.

Within the `defaults`-section it looks a little bit different, too. The `__NAMESPACE__` will be used to concatenate with the `controller` parameter at all times. So for example when the `controller` parameter is "news" then the controller to be called from the Router will be `Application\Controller\news`, if the parameter is "archive" the Router will call the controller `Application\Controller\archive`.

The `defaults`-section then is pretty straight forward again. Both parameters, `controller` and `action`, only have to follow the conventions given by PHP-Standards. They have to start with a letter from `a-z`, upper- or lowercase and after that first letter there can be an (almost) infinite amount of letters, digits, underscores or dashes.

The big downside to this approach not only is that matching this route is a little slower, it is that there is no error-checking going on. For example, when you were to call a URL like `domain.com/weird/doesntExist` then the controller would be "Application\Controller\weird" and the action would be "doesntExistAction". As you can guess by the names let's assume neither controller nor action does exist. The route will still match but an Exception will be thrown because the Router will be unable to find the requested resources and we'll receive a 404-Response.

Explicit routes using `child_routes`

Explicit routing is done by defining all possible routes yourself. For this method you actually have two options available, too.

Without config structure

The probably most easy to understand way to write explicit routes would be to write many top level routes like in the following configuration:

As you can see with this little example, all routes have an explicit name and there's lots of repetition going on. We have to redefine the default `controller` to be used every single time and we don't really have any structure within the configuration. Let's take a look at how we could bring more structure into a configuration like this.

Using `child_routes` for more structure

Another option to define explicit routes is to be using `child_routes`. Child routes inherit all options from their respective parents. Meaning: when the `controller` doesn't change, you do not need to redefine it. Let's take a look at a child routes configuration using the same example as above:

```
1  'router' => array(
2      'routes' => array(
3          'news' => array(
4              'type' => 'literal',
5              'options' => array(
6                  'route' => '/news',
7                  'defaults' => array(
8                      'controller' => 'NewsController',
```

```

9         'action'      => 'showAll',
10     ),
11 ),
12 // Defines that "/news" can be matched on its own without a child route_
↳being matched
13 'may_terminate' => true,
14 'child_routes' => array(
15     'archive' => array(
16         'type' => 'segment',
17         'options' => array(
18             'route'      => '/archive[:year]',
19             'defaults' => array(
20                 'action'      => 'archive',
21             ),
22             'constraints' => array(
23                 'year' => '\d{4}'
24             )
25         ),
26     ),
27     'single' => array(
28         'type' => 'segment',
29         'options' => array(
30             'route'      => '/:id',
31             'defaults' => array(
32                 'action'      => 'detail',
33             ),
34             'constraints' => array(
35                 'id' => '\d+'
36             )
37         ),
38     ),
39 ),
40 ),
41 ),
42 )

```

This routing configuration requires a little more explanation. First of all we have a new configuration entry which is called `may_terminate`. This property defines that the parent route can be matched alone, without child routes needing to be matched, too. In other words all of the following routes are valid:

- `/news`
- `/news/archive`
- `/news/archive/2014`
- `/news/42`

If, however, you were to set `may_terminate => false`, then the parent route would only be used for global defaults that all `child_routes` were to inherit. In other words: only `child_routes` can be matched, so the only valid routes would be:

- `/news/archive`
- `/news/archive/2014`
- `/news/42`

The parent route would not be able to be matched on its own.

Next to that we have a new entry called `child_routes`. In here we define new routes that will be appended to the

parent route. There's no real difference in configuration from routes you define as a child route to routes that are on the top level of the configuration. The only thing that may fall away is the re-definition of shared default values.

The big advantage you have with this kind of configuration is the fact that you explicitly define the routes and therefore you will never run into problems of non-existing controllers like you would with generic routes like described above. The second advantage would be that this kind of routing is a little bit faster than generic routes and the last advantage would be that you can easily see all possible URLs that start with `/news`.

While ultimately this falls into the category of personal preference bare in mind that debugging of explicit routes is significantly easier than debugging generic routes.

A practical example for our Blog Module

Now that we know how to configure new routes, let's first create a route to display only a single Blog from our Database. We want to be able to identify blog posts by their internal ID. Given that ID is a variable parameter we need a route of type `Segment`. Furthermore we want to put this route as a child route to the route of name `blog`.

```
1 <?php
2 // FileName: /module/Blog/config/module.config.php
3 return array(
4     'db' => array( /** DB Config */ ),
5     'service_manager' => array( /* ServiceManager Config */ ),
6     'view_manager' => array( /* ViewManager Config */ ),
7     'controllers' => array( /* ControllerManager Config */ ),
8     'router' => array(
9         'routes' => array(
10             'blog' => array(
11                 'type' => 'literal',
12                 'options' => array(
13                     'route' => '/blog',
14                     'defaults' => array(
15                         'controller' => 'Blog\Controller\List',
16                         'action' => 'index',
17                     ),
18                 ),
19                 'may_terminate' => true,
20                 'child_routes' => array(
21                     'detail' => array(
22                         'type' => 'segment',
23                         'options' => array(
24                             'route' => '/:id',
25                             'defaults' => array(
26                                 'action' => 'detail'
27                             ),
28                             'constraints' => array(
29                                 'id' => '[1-9]\d*'
30                             )
31                         )
32                     )
33                 )
34             )
35         )
36     );
37
```

With this we have set up a new route that we use to display a single blog entry. We have assigned a parameter called `id` that needs to be a positive digit excluding 0. Database entries usually start with a 0 when it comes to primary ID

keys and therefore our regular expression constraints for the `id` fields looks a little more complicated. Basically we tell the router that the parameter `id` has to start with an integer between 1 and 9, that's the `[1-9]` part, and after that zero or more digits can follow (that's the `\d*` part).

The route will call the same controller like the parent route but it will call the `detailAction()` instead. Go to your browser and request the URL `http://localhost:8080/blog/2`. You'll see the following error message:

```

1  A 404 error occurred
2
3  Page not found.
4  The requested controller was unable to dispatch the request.
5
6  Controller:
7  Blog\Controller\List
8
9  No Exception available

```

This is due to the fact that the controller tries to access the `detailAction()` which does not yet exist. Let's go ahead and create this action now. Go to your `ListController` and add the action. Return an empty `ViewModel` and then refresh the page.

```

1  <?php
2  // FileName: /module/Blog/src/Blog/Controller/ListController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Mvc\Controller\AbstractActionController;
7  use Zend\View\Model\ViewModel;
8
9  class ListController extends AbstractActionController
10 {
11     /**
12      * @var \Blog\Service\PostServiceInterface
13      */
14     protected $postService;
15
16     public function __construct(PostServiceInterface $postService)
17     {
18         $this->postService = $postService;
19     }
20
21     public function indexAction()
22     {
23         return new ViewModel(array(
24             'posts' => $this->postService->findAllPosts()
25         ));
26     }
27
28     public function detailAction()
29     {
30         return new ViewModel();
31     }
32 }

```

Now you'll see the all familiar message that a template was unable to be rendered. Let's create this template now and assume that we will get one `Post`-Object passed to the template to see the details of our blog. Create a new view file under `/view/blog/list/detail.phtml`:

```
1 <!-- FileName: /module/Blog/view/blog/list/detail.phtml -->
2 <h1>Post Details</h1>
3
4 <dl>
5     <dt>Post Title</dt>
6     <dd><?php echo $this->escapeHtml($this->post->getTitle());?></dd>
7     <dt>Post Text</dt>
8     <dd><?php echo $this->escapeHtml($this->post->getText());?></dd>
9 </dl>
```

Looking at this template we're expecting the variable `$this->post` to be an instance of our `Post-Model`. Let's now modify our `ListController` so that a `Post` will be passed.

```
1 <?php
2 // FileName: /module/Blog/src/Blog/Controller/ListController.php
3 namespace Blog\Controller;
4
5 use Blog\Service\PostServiceInterface;
6 use Zend\Mvc\Controller\AbstractActionController;
7 use Zend\View\Model\ViewModel;
8
9 class ListController extends AbstractActionController
10 {
11     /**
12      * @var \Blog\Service\PostServiceInterface
13      */
14     protected $postService;
15
16     public function __construct(PostServiceInterface $postService)
17     {
18         $this->postService = $postService;
19     }
20
21     public function indexAction()
22     {
23         return new ViewModel(array(
24             'posts' => $this->postService->findAllPosts()
25         ));
26     }
27
28     public function detailAction()
29     {
30         $id = $this->params()->fromRoute('id');
31
32         return new ViewModel(array(
33             'post' => $this->postService->findPost($id)
34         ));
35     }
36 }
```

If you refresh your application now you'll see the details for our `Post` to be displayed. However, there is one little Problem with what we have done. While we do have our `Service` set up to throw an `\InvalidArgumentException` whenever no `Post` matching a given `id` is found, we don't make use of this just yet. Go to your browser and open the URL `http://localhost:8080/blog/99`. You will see the following error message:

```

1  An error occurred
2  An error occurred during execution; please try again later.
3
4  Additional information:
5  InvalidArgumentException
6
7  File:
8  {rootPath}/module/Blog/src/Blog/Service/PostService.php:40
9
10 Message:
11 Could not find row 99

```

This is kind of ugly, so our `ListController` should be prepared to do something whenever an `InvalidArgumentException` is thrown by the `PostService`. Whenever an invalid Post is requested we want the User to be redirected to the Post-Overview. Let's do this by putting the call against the `PostService` in a try-catch statement.

```

1  <?php
2  // FileName: /module/Blog/src/Blog/Controller/ListController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Mvc\Controller\AbstractActionController;
7  use Zend\View\Model\ViewModel;
8
9  class ListController extends AbstractActionController
10 {
11     /**
12      * @var \Blog\Service\PostServiceInterface
13      */
14     protected $postService;
15
16     public function __construct(PostServiceInterface $postService)
17     {
18         $this->postService = $postService;
19     }
20
21     public function indexAction()
22     {
23         return new ViewModel(array(
24             'posts' => $this->postService->findAllPosts()
25         ));
26     }
27
28     public function detailAction()
29     {
30         $id = $this->params()->fromRoute('id');
31
32         try {
33             $post = $this->postService->findPost($id);
34         } catch (\InvalidArgumentException $ex) {
35             return $this->redirect()->toRoute('blog');
36         }
37
38         return new ViewModel(array(
39             'post' => $post
40         ));
41     }

```

42

```
}
```

Now whenever you access an invalid `id` you'll be redirected to the route `blog` which is our list of blog posts, perfect!

Making use of Forms and Fieldsets

So far all we did was read data from the database. In a real-life-application this won't get us very far as very often the least we need to do is to support full `Create`, `Read`, `Update` and `Delete` operations (CRUD). Most often the process of getting data into our database is that a user enters the data into a web `<form>` and the application then uses the user input and saves it into our backend.

Core components

We want to be able to do exactly this and Zend Framework provides us with all the tools we need to achieve our goal. Before we jump into coding, we need to understand the two core components for this task first. So let's take a look at what these components are and what they are used for.

Zend\Form\Fieldset

The first component that you have to know about is `Zend\Form\Fieldset`. A `Fieldset` is a component that contains a reusable set of elements. You will use the `Fieldset` to create the frontend-input for your backend-models. It is considered good practice to have one `Fieldset` for every `Model` of your application.

The `Fieldset`-component, however, is no `Form`, meaning you will not be able to use a `Fieldset` without attaching it to the `Form`-component. The advantage here is that you have one set of elements that you can re-use for as many `Forms` as you like without having to re-declare all the inputs for the `Model` that's represented by the `Fieldset`.

Zend\Form\Form

The main component you'll need and that most probably you've heard about already is `Zend\Form\Form`. The `Form`-component is the main container for all elements of your web `<form>`. You are able to add single elements or a set of elements in the form of a `Fieldset`, too.

Creating your first Fieldset

Explaining how the `Zend\Form` component works is best done by giving you real code to work with. So let's jump right into it and create all the forms we need to finish our `Blog` module. We start by creating a `Fieldset` that contains all the input elements that we need to work with our `Blog`-data.

- You will need one hidden input for the `id` property, which is only needed for editing and deleting data.
- You will need one text input for the `text` property
- You will need one text input for the `title` property

Create the file `/module/Blog/src/Blog/Form/PostFieldset.php` and add the following code:

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Form/PostFieldset.php
3 namespace Blog\Form;
4
5 use Zend\Form\Fieldset;
6
7 class PostFieldset extends Fieldset
8 {
9     public function __construct()
10     {
11         $this->add(array(
12             'type' => 'hidden',
13             'name' => 'id'
14         ));
15
16         $this->add(array(
17             'type' => 'text',
18             'name' => 'text',
19             'options' => array(
20                 'label' => 'The Text'
21             )
22         ));
23
24         $this->add(array(
25             'type' => 'text',
26             'name' => 'title',
27             'options' => array(
28                 'label' => 'Blog Title'
29             )
30         ));
31     }
32 }
```

As you can see this class is pretty handy. All we do is to have our class extend `Zend\Form\Fieldset` and then we write a `__construct()` method and add all the elements we need to the fieldset. This `Fieldset` can now be used by as many forms as we want. So let's go ahead and create our first `Form`.

Creating the PostForm

Now that we have our `PostFieldset` in place, we need to use it inside a `Form`. We then need to add a `Submit-Button` to the form so that the user will be able to submit the data and we're done. So create the `PostForm` within the same directory under `/module/Blog/src/Blog/Form/PostForm` and add the `PostFieldset` to it:

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Form/PostForm.php
3 namespace Blog\Form;
4
5 use Zend\Form\Form;
6
7 class PostForm extends Form
8 {
9     public function __construct()
10     {
11         $this->add(array(
12             'name' => 'post-fieldset',
13             'type' => 'Blog\Form\PostFieldset'
14         ));
15
16         $this->add(array(
17             'type' => 'submit',
18             'name' => 'submit',
19             'attributes' => array(
20                 'value' => 'Insert new Post'
21             )
22         ));
23     }
24 }

```

And that's our form. Nothing special here, we add our `PostFieldset` to the `Form`, we add a submit button to the form and nothing more. Let's now make use of the `Form`.

Adding a new Post

Now that we have the `PostForm` written we want to use it. But there are a couple more tasks that you need to do. The tasks that are standing right in front of you are:

- create a new controller `WriteController`
- add `PostService` as a dependency to the `WriteController`
- add `PostForm` as a dependency to the `WriteController`
- create a new route `blog/add` that routes to the `WriteController` and its `addAction()`
- create a new view that displays the form

Creating the WriteController

As you can see from the task-list we need a new controller and this controller is supposed to have two dependencies. One dependency being the `PostService` that's also being used within our `ListController` and the other dependency being the `PostForm` which is new. Since the `PostForm` is a dependency that the `ListController` doesn't need to display blog-data, we will create a new controller to keep things properly separated. First, register a controller-factory within the configuration:

```

1 <?php
2 // Filename: /module/Blog/config/module.config.php
3 return array(
4     'db' => array( /** DB Config */ ),
5     'service_manager' => array( /** ServiceManager Config */),

```

```

6     'view_manager'    => array( /** ViewManager Config */ ),
7     'controllers'    => array(
8         'factories' => array(
9             'Blog\Controller\List'    => 'Blog\Factory\ListControllerFactory',
10            'Blog\Controller\Write'   => 'Blog\Factory\WriteControllerFactory'
11        )
12    ),
13    'router'           => array( /** Router Config */ )
14 );
    
```

Next step would be to write the WriteControllerFactory. Have the factory return the WriteController and add the required dependencies within the constructor.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Factory/WriteControllerFactory.php
3  namespace Blog\Factory;
4
5  use Blog\Controller\WriteController;
6  use Zend\ServiceManager\FactoryInterface;
7  use Zend\ServiceManager\ServiceLocatorInterface;
8
9  class WriteControllerFactory implements FactoryInterface
10 {
11     public function createService(ServiceLocatorInterface $serviceLocator)
12     {
13         $realServiceLocator = $serviceLocator->getServiceLocator();
14         $postService        = $realServiceLocator->get (
15             ↪ 'Blog\Service\PostServiceInterface');
16         $postInsertForm     = $realServiceLocator->get ('FormElementManager')->get (
17             ↪ 'Blog\Form\PostForm');
18
19         return new WriteController(
20             $postService,
21             $postInsertForm
22         );
23     }
24 }
    
```

In this code-example there are a couple of things to be aware of. First, the WriteController doesn't exist yet, but we will create this in the next step so we're just assuming that it will exist later on. Second, we access the FormElementManager to get access to our PostForm. All forms should be accessed through the FormElementManager. Even though we haven't registered the PostForm in our config files yet the FormElementManager automatically knows about forms that act as invocables. As long as you have no dependencies you don't need to register them explicitly.

Next up is the creation of our controller. Be sure to type hint the dependencies by their interfaces and to add the addAction()!

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Controller/WriteController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Form\FormInterface;
7  use Zend\Mvc\Controller\AbstractActionController;
8
9  class WriteController extends AbstractActionController
10 {
    
```

```

11     protected $postService;
12
13     protected $postForm;
14
15     public function __construct(
16         PostServiceInterface $postService,
17         FormInterface $postForm
18     ) {
19         $this->postService = $postService;
20         $this->postForm     = $postForm;
21     }
22
23     public function addAction()
24     {
25     }
26 }

```

Right on to creating the new route:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'db' => array( /** Db Config */ ),
5      'service_manager' => array( /** ServiceManager Config */ ),
6      'view_manager' => array( /** ViewManager Config */ ),
7      'controllers' => array( /** Controller Config */ ),
8      'router' => array(
9          'routes' => array(
10             'blog' => array(
11                 'type' => 'literal',
12                 'options' => array(
13                     'route' => '/blog',
14                     'defaults' => array(
15                         'controller' => 'Blog\Controller\List',
16                         'action' => 'index',
17                     )
18                 ),
19                 'may_terminate' => true,
20                 'child_routes' => array(
21                     'detail' => array(
22                         'type' => 'segment',
23                         'options' => array(
24                             'route' => '/:id',
25                             'defaults' => array(
26                                 'action' => 'detail'
27                             )
28                         ),
29                         'constraints' => array(
30                             'id' => '\d+'
31                         )
32                     )
33                 ),
34                 'add' => array(
35                     'type' => 'literal',
36                     'options' => array(
37                         'route' => '/add',
38                         'defaults' => array(
39                             'controller' => 'Blog\Controller\Write',
40                             'action' => 'add'

```

```

40         )
41     )
42 )
43 )
44 )
45 )
46 )
47 );

```

And lastly let's create a dummy template:

```

1 <!-- Filename: /module/Blog/view/blog/write/add.phtml -->
2 <h1>WriteController::addAction()</h1>

```

Checking the current status

If you try to access the new route `localhost:8080/blog/add` you're supposed to see the following error message:

```

1 Fatal error: Call to a member function insert() on a non-object in
2 {libraryPath}/Zend/Form/Fieldset.php on line {lineNumber}

```

If this is not the case, be sure to follow the tutorial correctly and carefully check all your files. Assuming you are getting this error, let's find out what it means and fix it!

The above error message is very common and its solution isn't that intuitive. It appears that there is an error within the `Zend/Form/Fieldset.php` but that's not the case. The error message let's you know that something didn't go right while you were creating your form. In fact, while creating both the `PostForm` as well as the `PostFieldset` we have forgotten something very, very important.

Note: When overwriting a `__construct()` method within the `Zend\Form`-component, be sure to always call `parent::__construct()`!

Without this, forms and fieldsets will not be able to get initiated correctly. Let's now fix the problem by calling the parents constructor in both form and fieldset. To have more flexibility we will also include the signature of the `__construct()` function which accepts a couple of parameters.

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Form/PostForm.php
3 namespace Blog\Form;
4
5 use Zend\Form\Form;
6
7 class PostForm extends Form
8 {
9     public function __construct($name = null, $options = array())
10     {
11         parent::__construct($name, $options);
12
13         $this->add(array(
14             'name' => 'post-fieldset',
15             'type' => 'Blog\Form\PostFieldset'
16         ));
17
18         $this->add(array(
19             'type' => 'submit',

```

```

20         'name' => 'submit',
21         'attributes' => array(
22             'value' => 'Insert new Post'
23         )
24     ));
25 }
26 }

```

As you can see our `PostForm` now accepts two parameters to give our form a name and to set a couple of options. Both parameters will be passed along to the parent. If you look closely at how we add the `PostFieldset` to the form you'll notice that we assign a name to the fieldset. Those options will be passed from the `FormElementManager` when the `PostFieldset` is created. But for this to function we need to do the same step inside our fieldset, too:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Form/PostFieldset.php
3  namespace Blog\Form;
4
5  use Zend\Form\Fieldset;
6
7  class PostFieldset extends Fieldset
8  {
9      public function __construct($name = null, $options = array())
10     {
11         parent::__construct($name, $options);
12
13         $this->add(array(
14             'type' => 'hidden',
15             'name' => 'id'
16         ));
17
18         $this->add(array(
19             'type' => 'text',
20             'name' => 'text',
21             'options' => array(
22                 'label' => 'The Text'
23             )
24         ));
25
26         $this->add(array(
27             'type' => 'text',
28             'name' => 'title',
29             'options' => array(
30                 'label' => 'Blog Title'
31             )
32         ));
33     }
34 }

```

Reloading your application now will yield you the desired result.

Displaying the form

Now that we have our `PostForm` within our `WriteController` it's time to pass this form to the view and have it rendered using the provided `ViewHelpers` from the `Zend\Form` component. First change your controller so that the form is passed to the view.

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Controller/WriteController.php
3 namespace Blog\Controller;
4
5 use Blog\Service\PostServiceInterface;
6 use Zend\Form\FormInterface;
7 use Zend\Mvc\Controller\AbstractActionController;
8 use Zend\View\Model\ViewModel;
9
10 class WriteController extends AbstractActionController
11 {
12     protected $postService;
13
14     protected $postForm;
15
16     public function __construct(
17         PostServiceInterface $postService,
18         FormInterface $postForm
19     ) {
20         $this->postService = $postService;
21         $this->postForm     = $postForm;
22     }
23
24     public function addAction()
25     {
26         return new ViewModel(array(
27             'form' => $this->postForm
28         ));
29     }
30 }
```

And then we need to modify our view to have the form rendered.

```
1 <!-- Filename: /module/Blog/view/blog/write/add.phtml -->
2 <h1>WriteController::addAction()</h1>
3 <?php
4 $form = $this->form;
5 $form->setAttribute('action', $this->url());
6 $form->prepare();
7
8 echo $this->form()->openTag($form);
9
10 echo $this->formCollection($form);
11
12 echo $this->form()->closeTag();
```

Firstly, we tell the form that it should send its data to the current URL and then we tell the form to `prepare()` itself which triggers a couple of internal things.

Note: HTML-Forms can be sent using POST and GET. ZF2s default is POST, therefore you don't have to be explicit in setting this option. If you want to change it to GET though, all you have to do is set the specific attribute prior to the `prepare()` call.

```
$form->setAttribute('method', 'GET');
```

Next we're using a couple of `ViewHelpers` which take care of rendering the form for us. There are many different ways to render a form within Zend Framework but using `formCollection()` is probably the fastest one.

Refreshing the browser you will now see your form properly displayed. However, if we're submitting the form all we see is our form being displayed again. And this is due to the simple fact that we didn't add any logic to the controller yet.

Note: Keep in mind that this tutorial focuses solely on the OOP aspect of things. Rendering the form like this, without any stylesheets added doesn't really reflect most designers' idea of a beautiful form. You'll find out more about the rendering of forms in the chapter of *Zend\Form\View\Helper*.

Controller Logic for basically all Forms

Writing a Controller that handles a form workflow is pretty simple and it's basically identical for each and every form you have within your application.

1. You want to check if the current request is a POST-Request, meaning if the form has been sent
2. **If the form has been sent, you want to:**
 - store the POST-Data within the Form
 - check if the form passes validation
3. **If the form passes validation, you want to:**
 - pass the form data to your service to have it stored
 - redirect the user to either the detail page of the entered data or to some overview page
4. In all other cases, you want the form displayed, sometimes alongside given error messages

And all of this is really not that much code. Modify your `WriteController` to the following code:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Controller/WriteController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Form\FormInterface;
7  use Zend\Mvc\Controller\AbstractActionController;
8  use Zend\View\Model\ViewModel;
9
10 class WriteController extends AbstractActionController
11 {
12     protected $postService;
13
14     protected $postForm;
15
16     public function __construct (
17         PostServiceInterface $postService,
18         FormInterface $postForm
19     ) {
20         $this->postService = $postService;
21         $this->postForm     = $postForm;
22     }
23
24     public function addAction()
25     {
26         $request = $this->getRequest();

```

```

27
28     if ($request->isPost()) {
29         $this->postForm->setData($request->getPost());
30
31         if ($this->postForm->isValid()) {
32             try {
33                 $this->postService->savePost($this->postForm->getData());
34
35                 return $this->redirect()->toRoute('blog');
36             } catch (\Exception $e) {
37                 // Some DB Error happened, log it and let the user know
38             }
39         }
40     }
41
42     return new ViewModel(array(
43         'form' => $this->postForm
44     ));
45 }
46

```

This example code should be pretty straight forward. First we save the current request into a local variable. Then we check if the current request is a POST-Request and if so, we store the requests POST-data into the form. If the form turns out to be valid we try to save the form data through our service and then redirect the user to the route `blog`. If any error occurred at any point we simply display the form again.

Submitting the form right now will return into the following error

```

1 Fatal error: Call to undefined method Blog\Service\PostService::savePost() in
2 /module/Blog/src/Blog/Controller/WriteController.php on line 33

```

Let's fix this by extending our `PostService`. Be sure to also change the signature of the `PostServiceInterface`!

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Service/PostServiceInterface.php
3 namespace Blog\Service;
4
5 use Blog\Model\PostInterface;
6
7 interface PostServiceInterface
8 {
9     /**
10      * Should return a set of all blog posts that we can iterate over. Single
11      * entries of the array are supposed to be
12      * implementing \Blog\Model\PostInterface
13      * @return array|PostInterface[]
14      */
15     public function findAllPosts();
16
17     /**
18      * Should return a single blog post
19      *
20      * @param int $id Identifier of the Post that should be returned
21      * @return PostInterface
22      */
23     public function findPost($id);

```

```

24
25     /**
26      * Should save a given implementation of the PostInterface and return it. If it
27      * is an existing Post the Post
28      * should be updated, if it's a new Post it should be created.
29      *
30      * @param PostInterface $blog
31      * @return PostInterface
32      */
33     public function savePost(PostInterface $blog);

```

As you can see the `savePost()` function has been added and needs to be implemented within the `PostService` now.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Service/PostService.php
3  namespace Blog\Service;
4
5  use Blog\Mapper\PostMapperInterface;
6
7
8  class PostService implements PostServiceInterface
9  {
10     /**
11      * @var \Blog\Mapper\PostMapperInterface
12      */
13     protected $postMapper;
14
15     /**
16      * @param PostMapperInterface $postMapper
17      */
18     public function __construct(PostMapperInterface $postMapper)
19     {
20         $this->postMapper = $postMapper;
21     }
22
23     /**
24      * {@inheritDoc}
25      */
26     public function findAllPosts()
27     {
28         return $this->postMapper->findAll();
29     }
30
31     /**
32      * {@inheritDoc}
33      */
34     public function findPost($id)
35     {
36         return $this->postMapper->find($id);
37     }
38
39     /**
40      * {@inheritDoc}
41      */
42     public function savePost(PostInterface $post)
43     {

```

```
44         return $this->postMapper->save($post);
45     }
46 }
```

And now that we're making an assumption against our postMapper we need to extend the PostMapperInterface and its implementation, too. Start by extending the interface:

```
1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/PostMapperInterface.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6
7  interface PostMapperInterface
8  {
9      /**
10       * @param int|string $id
11       * @return PostInterface
12       * @throws \InvalidArgumentException
13       */
14     public function find($id);
15
16     /**
17      * @return array|PostInterface[]
18      */
19     public function findAll();
20
21     /**
22      * @param PostInterface $postObject
23      *
24      * @param PostInterface $postObject
25      * @return PostInterface
26      * @throws \Exception
27      */
28     public function save(PostInterface $postObject);
29 }
```

And now the implementation of the save function.

```
1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6  use Zend\Db\Adapter\AdapterInterface;
7  use Zend\Db\Adapter\Driver\ResultInterface;
8  use Zend\Db\ResultSet\HydratingResultSet;
9  use Zend\Db\Sql\Insert;
10 use Zend\Db\Sql\Sql;
11 use Zend\Db\Sql\Update;
12 use Zend\Stdlib\Hydrator\HydratorInterface;
13
14 class ZendDbSqlMapper implements PostMapperInterface
15 {
16     /**
17      * @var \Zend\Db\Adapter\AdapterInterface
18      */
19     protected $dbAdapter;
```

```

20
21  /**
22   * @var \Zend\Stdlib\Hydrator\HydratorInterface
23   */
24  protected $hydrator;
25
26  /**
27   * @var \Blog\Model\PostInterface
28   */
29  protected $postPrototype;
30
31  /**
32   * @param AdapterInterface $dbAdapter
33   * @param HydratorInterface $hydrator
34   * @param PostInterface $postPrototype
35   */
36  public function __construct(
37      AdapterInterface $dbAdapter,
38      HydratorInterface $hydrator,
39      PostInterface $postPrototype
40  ) {
41      $this->dbAdapter      = $dbAdapter;
42      $this->hydrator        = $hydrator;
43      $this->postPrototype  = $postPrototype;
44  }
45
46  /**
47   * @param int|string $id
48   *
49   * @return PostInterface
50   * @throws \InvalidArgumentException
51   */
52  public function find($id)
53  {
54      $sql      = new Sql($this->dbAdapter);
55      $select    = $sql->select('posts');
56      $select->where(array('id = ?' => $id));
57
58      $stmt      = $sql->prepareStatementForSqlObject($select);
59      $result     = $stmt->execute();
60
61      if ($result instanceof ResultInterface && $result->isQueryResult() && $result->
↪getAffectedRows()) {
62          return $this->hydrator->hydrate($result->current(), $this->postPrototype);
63      }
64
65      throw new \InvalidArgumentException("Blog with given ID:{$id} not found.");
66  }
67
68  /**
69   * @return array|PostInterface[]
70   */
71  public function findAll()
72  {
73      $sql      = new Sql($this->dbAdapter);
74      $select    = $sql->select('posts');
75
76      $stmt      = $sql->prepareStatementForSqlObject($select);

```

```

77     $result = $stmt->execute();
78
79     if ($result instanceof ResultInterface && $result->isQueryResult()) {
80         $resultSet = new HydratingResultSet($this->hydrator, $this->postPrototype);
81
82         return $resultSet->initialize($result);
83     }
84
85     return array();
86 }
87
88 /**
89  * @param PostInterface $postObject
90  *
91  * @return PostInterface
92  * @throws \Exception
93  */
94 public function save(PostInterface $postObject)
95 {
96     $postData = $this->hydrator->extract($postObject);
97     unset($postData['id']); // Neither Insert nor Update needs the ID in the array
98
99     if ($postObject->getId()) {
100         // ID present, it's an Update
101         $action = new Update('posts');
102         $action->set($postData);
103         $action->where(array('id = ?' => $postObject->getId()));
104     } else {
105         // ID NOT present, it's an Insert
106         $action = new Insert('posts');
107         $action->values($postData);
108     }
109
110     $sql = new Sql($this->dbAdapter);
111     $stmt = $sql->prepareStatementForSqlObject($action);
112     $result = $stmt->execute();
113
114     if ($result instanceof ResultInterface) {
115         if ($newId = $result->getGeneratedValue()) {
116             // When a value has been generated, set it on the object
117             $postObject->setId($newId);
118         }
119
120         return $postObject;
121     }
122
123     throw new \Exception("Database error");
124 }
125 }

```

The `save()` function handles two cases. The insert and update routine. Firstly we extract the `Post`-Object since we need array data to work with `Insert` and `Update`. Then we remove the `id` from the array since this field is not wanted. When we do an update of a row, we don't update the `id` property itself and therefore it isn't needed. On the insert routine we don't need an `id` either so we can simply strip it away.

After the `id` field has been removed we check what action is supposed to be called. If the `Post`-Object has an `id` set we create a new `Update`-Object and if not we create a new `Insert`-Object. We set the data for both actions accordingly and after that the data is passed over to the `Sql`-Object for the actual query into the database.

At last we check if we receive a valid result and if there has been an `id` generated. If it's the case we call the `setId()`-function of our `blog` and return the object in the end.

Let's submit our form again and see what we get.

```
1 Catchable fatal error: Argument 1 passed to Blog\Service\PostService::savePost()
2 must implement interface Blog\Model\PostInterface, array given,
3 called in /module/Blog/src/Blog/Controller/InsertController.php on line 33
4 and defined in /module/Blog/src/Blog/Service/PostService.php on line 49
```

Forms, per default, give you data in an array format. But our `PostService` expects the format to be an implementation of the `PostInterface`. This means we need to find a way to have this array data become object data. If you recall the previous chapter, this is done through the use of hydrators.

Note: On the Update-Query you'll notice that we have assigned a condition to only update the row matching a given `id`

```
$action->where(array('id = ?' => $postObject->getId()));
```

You'll see here that the condition is: **id equals ?**. With the question-mark being the `id` of the post-object. In the same way you could assign a condition to update (or select) rows with all entries higher than a given `id`:

```
$action->where(array('id > ?' => $postObject->getId()));
```

This works for all conditions. `=`, `>`, `<`, `>=` and `<=`

Zend\Form and Zend\Stdlib\Hydrator working together

Before we go ahead and put the hydrator into the form, let's first do a data-dump of the data coming from the form. That way we can easily notice all changes that the hydrator does. Modify your `WriteController` to the following:

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Controller/WriteController.php
3 namespace Blog\Controller;
4
5 use Blog\Service\PostServiceInterface;
6 use Zend\Form\FormInterface;
7 use Zend\Mvc\Controller\AbstractActionController;
8 use Zend\View\Model\ViewModel;
9
10 class WriteController extends AbstractActionController
11 {
12     protected $postService;
13
14     protected $postForm;
15
16     public function __construct(
17         PostServiceInterface $postService,
18         FormInterface $postForm
19     ) {
20         $this->postService = $postService;
21         $this->postForm     = $postForm;
22     }
23
24     public function addAction()
25     {
```

```

26         $request = $this->getRequest();
27
28         if ($request->isPost()) {
29             $this->postForm->setData($request->getPost());
30
31             if ($this->postForm->isValid()) {
32                 try {
33                     \Zend\Debug\Debug::dump($this->postForm->getData()); die();
34                     $this->postService->savePost($this->postForm->getData());
35
36                     return $this->redirect()->toRoute('blog');
37                 } catch (\Exception $e) {
38                     // Some DB Error happened, log it and let the user know
39                 }
40             }
41         }
42
43         return new ViewModel(array(
44             'form' => $this->postForm
45         ));
46     }
47 }

```

With this set up go ahead and submit the form once again. You should now see a data dump like the following:

```

1 array(2) {
2     ["submit"] => string(16) "Insert new Post"
3     ["post-fieldset"] => array(3) {
4         ["id"] => string(0) ""
5         ["text"] => string(3) "foo"
6         ["title"] => string(3) "bar"
7     }
8 }

```

Now telling your fieldset to hydrate its data into an Post-object is very simple. All you need to do is to assign the hydrator and the object prototype like this:

```

1 <?php
2 // Filename: /module/Blog/src/Blog/Form/PostFieldset.php
3 namespace Blog\Form;
4
5 use Blog\Model\Post;
6 use Zend\Form\Fieldset;
7 use Zend\Stdlib\Hydrator\ClassMethods;
8
9 class PostFieldset extends Fieldset
10 {
11     public function __construct($name = null, $options = array())
12     {
13         parent::__construct($name, $options);
14
15         $this->setHydrator(new ClassMethods(false));
16         $this->setObject(new Post());
17
18         $this->add(array(
19             'type' => 'hidden',
20             'name' => 'id'
21         ));
22     }
23 }

```



```

22
23     $this->add(array(
24         'type' => 'text',
25         'name' => 'text',
26         'options' => array(
27             'label' => 'The Text'
28         )
29     ));
30
31     $this->add(array(
32         'type' => 'text',
33         'name' => 'title',
34         'options' => array(
35             'label' => 'Blog Title'
36         )
37     ));
38 }
39 }

```

As you can see we're doing two things. We tell the fieldset to be using the `ClassMethods` hydrator and then we tell the fieldset that the default object to be returned is our `Blog-Model`. However, when you're re-submitting the form now you'll notice that nothing has changed. We're still only getting array data returned and no object.

This is due to the fact that the form itself doesn't know that it has to return an object. When the form doesn't know that it's supposed to return an object it uses the `ArraySerializable` hydrator recursively. To change this, all we need to do is to make our `PostFieldset` a so-called `base_fieldset`.

A `base_fieldset` basically tells the form "this form is all about me, don't worry about other data, just worry about me". And when the form knows that this fieldset is the real deal, then the form will use the hydrator presented by the fieldset and return the object that we desire. Modify your `PostForm` and assign the `PostFieldset` as `base_fieldset`:

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Form/PostForm.php
3  namespace Blog\Form;
4
5  use Zend\Form\Form;
6
7  class PostForm extends Form
8  {
9      public function __construct($name = null, $options = array())
10     {
11         parent::__construct($name, $options);
12
13         $this->add(array(
14             'name' => 'post-fieldset',
15             'type' => 'Blog\Form\PostFieldset',
16             'options' => array(
17                 'use_as_base_fieldset' => true
18             )
19         ));
20
21         $this->add(array(
22             'type' => 'submit',
23             'name' => 'submit',
24             'attributes' => array(
25                 'value' => 'Insert new Post'
26             )

```

```

27         ));
28     }
29 }

```

Now submit your form again. You should see the following output:

```

1  object(Blog\Model\Post)#294 (3) {
2      ["id":protected] => string(0) ""
3      ["title":protected] => string(3) "foo"
4      ["text":protected] => string(3) "bar"
5  }

```

You can now revert back your WriteController to its previous form to have the form-data passed through the PostService.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Controller/WriteController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Form\FormInterface;
7  use Zend\Mvc\Controller\AbstractActionController;
8  use Zend\View\Model\ViewModel;
9
10 class WriteController extends AbstractActionController
11 {
12     protected $postService;
13
14     protected $postForm;
15
16     public function __construct(
17         PostServiceInterface $postService,
18         FormInterface $postForm
19     ) {
20         $this->postService = $postService;
21         $this->postForm     = $postForm;
22     }
23
24     public function addAction()
25     {
26         $request = $this->getRequest();
27
28         if ($request->isPost()) {
29             $this->postForm->setData($request->getPost());
30
31             if ($this->postForm->isValid()) {
32                 try {
33                     $this->postService->savePost($this->postForm->getData());
34
35                     return $this->redirect()->toRoute('blog');
36                 } catch (\Exception $e) {
37                     // Some DB Error happened, log it and let the user know
38                 }
39             }
40         }
41
42         return new ViewModel(array(
43             'form' => $this->postForm

```

```
44         ));  
45     }  
46 }
```

If you send the form now you'll now be able to add as many new blogs as you want. Great!

Conclusion

In this chapter you've learned a great deal about the `Zend\Form` component. You've learned that `Zend\Stdlib\Hydrator` takes a big part within the `Zend\Form` component and by making use of both components you've been able to create an insert form for the blog module.

In the next chapter we will finalize the CRUD functionality by creating the update and delete routines for the blog module.

Editing and Deleting Data

In the previous chapter we've come to learn how we can use the `Zend\Form`- and `Zend\Db`-components to create the functionality of creating new data-sets. This chapter will focus on finalizing the CRUD functionality by introducing the concepts for editing and deleting data. We start by editing the data.

Binding Objects to Forms

The one fundamental difference between an insert- and an edit-form is the fact that inside an edit-form there is already data preset. This means we need to find a way to get data from our database into the form. Luckily `Zend\Form` provides us with a very handy way of doing so and it's called **data-binding**.

All you need to do when providing an edit-form is to get the object of interest from your service and bind it to the form. This is done the following way inside your controller.

```
1  <?php
2  // Filename: /module/Blog/src/Blog/Controller/WriteController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Form\FormInterface;
7  use Zend\Mvc\Controller\AbstractActionController;
8  use Zend\View\Model\ViewModel;
9
10 class WriteController extends AbstractActionController
11 {
12     protected $postService;
13
14     protected $postForm;
15
16     public function __construct(
17         PostServiceInterface $postService,
18         FormInterface $postForm
19     ) {
20         $this->postService = $postService;
```

```
21     $this->postForm    = $postForm;
22 }
23
24 public function addAction()
25 {
26     $request = $this->getRequest();
27
28     if ($request->isPost()) {
29         $this->postForm->setData($request->getPost());
30
31         if ($this->postForm->isValid()) {
32             try {
33                 $this->postService->savePost($this->postForm->getData());
34
35                 return $this->redirect()->toRoute('blog');
36             } catch (\Exception $e) {
37                 die($e->getMessage());
38                 // Some DB Error happened, log it and let the user know
39             }
40         }
41     }
42
43     return new ViewModel(array(
44         'form' => $this->postForm
45     ));
46 }
47
48 public function editAction()
49 {
50     $request = $this->getRequest();
51     $post    = $this->postService->findPost($this->params('id'));
52
53     $this->postForm->bind($post);
54
55     if ($request->isPost()) {
56         $this->postForm->setData($request->getPost());
57
58         if ($this->postForm->isValid()) {
59             try {
60                 $this->postService->savePost($post);
61
62                 return $this->redirect()->toRoute('blog');
63             } catch (\Exception $e) {
64                 die($e->getMessage());
65                 // Some DB Error happened, log it and let the user know
66             }
67         }
68     }
69
70     return new ViewModel(array(
71         'form' => $this->postForm
72     ));
73 }
74 }
```

Compared to the `addAction()` the `editAction()` has only three different lines. The first one is used to simply get the relevant `Post`-object from the service identified by the `id`-parameter of the route (which we'll be writing soon).

The second line then shows you how you can bind data to the `Zend\Form-Component`. We're able to use an object here because our `PostFieldset` will use the hydrator to display the data coming from the object.

Lastly instead of actually doing `$form->getData()` we simply use the previous `$post`-variable since it will be updated with the latest data from the form thanks to the data-binding. And that's all there is to it. The only things we need to add now is the new edit-route and the view for it.

Adding the edit-route

The edit route is a normal segment route just like the route `blog/detail`. Configure your route config to include the new route:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'db' => array( /** Db Config */ ),
5      'service_manager' => array( /** ServiceManager Config */ ),
6      'view_manager' => array( /** ViewManager Config */ ),
7      'controllers' => array( /** ControllerManager Config* */ ),
8      'router' => array(
9          'routes' => array(
10             'blog' => array(
11                 'type' => 'literal',
12                 'options' => array(
13                     'route' => '/blog',
14                     'defaults' => array(
15                         'controller' => 'Blog\Controller\List',
16                         'action' => 'index',
17                     )
18                 ),
19                 'may_terminate' => true,
20                 'child_routes' => array(
21                     'detail' => array(
22                         'type' => 'segment',
23                         'options' => array(
24                             'route' => '/:id',
25                             'defaults' => array(
26                                 'action' => 'detail'
27                             )
28                         ),
29                         'constraints' => array(
30                             'id' => '\d+'
31                         )
32                     )
33                 ),
34                 'add' => array(
35                     'type' => 'literal',
36                     'options' => array(
37                         'route' => '/add',
38                         'defaults' => array(
39                             'controller' => 'Blog\Controller\Write',
40                             'action' => 'add'
41                         )
42                     )
43                 ),
44                 'edit' => array(
45                     'type' => 'segment',

```

```

45         'options' => array(
46             'route' => '/edit/:id',
47             'defaults' => array(
48                 'controller' => 'Blog\Controller\Write',
49                 'action' => 'edit'
50             ),
51             'constraints' => array(
52                 'id' => '\d+'
53             )
54         ),
55     ),
56 ),
57 ),
58 ),
59 );
60

```

Creating the edit-template

Next in line is the creation of the new template `blog/write/edit`:

All that is really changing on the view-end is that you need to pass the current `id` to the `url()` view helper. To achieve this you have two options. The first one would be to pass the ID to the parameters array like

```
1 $this->url('blog/edit', array('id' => $id));
```

The downside is that `$id` is not available as we have not assigned it to the view. The `Zend\Mvc\Router`-component however provides us with a nice functionality to re-use the currently matched parameters. This is done by setting the last parameter of the view-helper to `true`.

```
1 $this->url('blog/edit', array(), true);
```

Checking the status

If you go to your browser and open up the edit form at `localhost:8080/blog/edit/1` you'll see that the form contains the data from your selected blog. And when you submit the form you'll notice that the data has been changed successfully. However sadly the submit-button still contains the text `Insert new Post`. This can be changed inside the view, too.

```

1 <!-- Filename: /module/Blog/view/blog/write/add.phtml -->
2 <h1>WriteController::editAction()</h1>
3 <?php
4 $form = $this->form;
5 $form->setAttribute('action', $this->url('blog/edit', array(), true));
6 $form->prepare();
7
8 $form->get('submit')->setValue('Update Post');
9
10 echo $this->form()->openTag($form);
11
12 echo $this->formCollection($form);
13
14 echo $this->form()->closeTag();

```


Implementing the delete functionality

Last but not least it's time to delete some data. We start this process by creating a new route and adding a new controller:

```

1  <?php
2  // Filename: /module/Blog/config/module.config.php
3  return array(
4      'db'                => array( /** Db Config */ ),
5      'service_manager' => array( /** ServiceManager Config */ ),
6      'view_manager'    => array( /** ViewManager Config */ ),
7      'controllers'     => array(
8          'factories' => array(
9              'Blog\Controller\List'    => 'Blog\Factory\ListControllerFactory',
10             'Blog\Controller\Write'   => 'Blog\Factory\WriteControllerFactory',
11             'Blog\Controller\Delete'  => 'Blog\Factory\DeleteControllerFactory'
12         )
13     ),
14     'router'            => array(
15         'routes' => array(
16             'post' => array(
17                 'type' => 'literal',
18                 'options' => array(
19                     'route' => '/blog',
20                     'defaults' => array(
21                         'controller' => 'Blog\Controller\List',
22                         'action' => 'index',
23                     )
24                 ),
25                 'may_terminate' => true,
26                 'child_routes' => array(
27                     'detail' => array(
28                         'type' => 'segment',
29                         'options' => array(
30                             'route' => '/:id',
31                             'defaults' => array(
32                                 'action' => 'detail'
33                             ),
34                             'constraints' => array(
35                                 'id' => '\d+'
36                             )
37                         )
38                     ),
39                     'add' => array(
40                         'type' => 'literal',
41                         'options' => array(
42                             'route' => '/add',
43                             'defaults' => array(
44                                 'controller' => 'Blog\Controller\Write',
45                                 'action' => 'add'
46                             )
47                         )
48                     ),
49                     'edit' => array(
50                         'type' => 'segment',
51                         'options' => array(
52                             'route' => '/edit/:id',
53                             'defaults' => array(

```

```

54         'controller' => 'Blog\Controller\Write',
55         'action'      => 'edit'
56     ),
57     'constraints' => array(
58         'id' => '\d+'
59     )
60 )
61 ),
62 'delete' => array(
63     'type' => 'segment',
64     'options' => array(
65         'route' => '/delete/:id',
66         'defaults' => array(
67             'controller' => 'Blog\Controller\Delete',
68             'action' => 'delete'
69         ),
70         'constraints' => array(
71             'id' => '\d+'
72         )
73     )
74 ),
75 )
76 )
77 )
78 )
79 );
    
```

Notice here that we have assigned yet another controller `Blog\Controller\Delete`. This is due to the fact that this controller will **not** require the `PostForm`. A `DeleteForm` is a perfect example for when you do not even need to make use of the `Zend\Form` component. Let's go ahead and create our controller first:

The Factory

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Factory/DeleteControllerFactory.php
3  namespace Blog\Factory;
4
5  use Blog\Controller\DeleteController;
6  use Zend\ServiceManager\FactoryInterface;
7  use Zend\ServiceManager\ServiceLocatorInterface;
8
9  class DeleteControllerFactory implements FactoryInterface
10 {
11     /**
12      * Create service
13      *
14      * @param ServiceLocatorInterface $serviceLocator
15      *
16      * @return mixed
17      */
18     public function createService(ServiceLocatorInterface $serviceLocator)
19     {
20         $realServiceLocator = $serviceLocator->getServiceLocator();
21         $postService        = $realServiceLocator->get(
22             'Blog\Service\PostServiceInterface');
23
24         return new DeleteController($postService);
25     }
26 }
    
```

25 }
}

The Controller

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Controller/DeleteController.php
3  namespace Blog\Controller;
4
5  use Blog\Service\PostServiceInterface;
6  use Zend\Mvc\Controller\AbstractActionController;
7  use Zend\View\Model\ViewModel;
8
9  class DeleteController extends AbstractActionController
10 {
11     /**
12      * @var \Blog\Service\PostServiceInterface
13      */
14     protected $postService;
15
16     public function __construct(PostServiceInterface $postService)
17     {
18         $this->postService = $postService;
19     }
20
21     public function deleteAction()
22     {
23         try {
24             $post = $this->postService->findPost($this->params('id'));
25         } catch (\InvalidArgumentException $e) {
26             return $this->redirect()->toRoute('blog');
27         }
28
29         $request = $this->getRequest();
30
31         if ($request->isPost()) {
32             $del = $request->getPost('delete_confirmation', 'no');
33
34             if ($del === 'yes') {
35                 $this->postService->deletePost($post);
36             }
37
38             return $this->redirect()->toRoute('blog');
39         }
40
41         return new ViewModel(array(
42             'post' => $post
43         ));
44     }
45 }
```

As you can see this is nothing new. We inject the `PostService` into the controller and inside the action we first check if the blog exists. If so we check if it's a post request and inside there we check if a certain post parameter called `delete_confirmation` is present. If the value of that then is `yes` we delete the blog through the `PostService's deletePost()` function.

When you're writing this code you'll notice that you don't get typehints for the `deletePost()` function because we haven't added it to the service / interface yet. Go ahead and add the function to the interface and implement it inside the service.

The Interface

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Service/PostServiceInterface.php
3 namespace Blog\Service;
4
5 use Blog\Model\PostInterface;
6
7 interface PostServiceInterface
8 {
9     /**
10      * Should return a set of all blog posts that we can iterate over. Single_
11      * entries of the array are supposed to be
12      * implementing \Blog\Model\PostInterface
13      * @return array|PostInterface[]
14      */
15     public function findAllPosts();
16
17     /**
18      * Should return a single blog post
19      *
20      * @param int $id Identifier of the Post that should be returned
21      * @return PostInterface
22      */
23     public function findPost($id);
24
25     /**
26      * Should save a given implementation of the PostInterface and return it. If it_
27      * is an existing Post the Post
28      * should be updated, if it's a new Post it should be created.
29      *
30      * @param PostInterface $blog
31      * @return PostInterface
32      */
33     public function savePost(PostInterface $blog);
34
35     /**
36      * Should delete a given implementation of the PostInterface and return true if_
37      * the deletion has been
38      * successful or false if not.
39      *
40      * @param PostInterface $blog
41      * @return bool
42      */
43     public function deletePost(PostInterface $blog);
44 }
```

The Service

```
1 <?php
2 // Filename: /module/Blog/src/Blog/Service/PostService.php
3 namespace Blog\Service;
4
5 use Blog\Mapper\PostMapperInterface;
6 use Blog\Model\PostInterface;
7
8 class PostService implements PostServiceInterface
9 {
```

```

10  /**
11   * @var \Blog\Mapper\PostMapperInterface
12   */
13  protected $postMapper;
14
15  /**
16   * @param PostMapperInterface $postMapper
17   */
18  public function __construct(PostMapperInterface $postMapper)
19  {
20      $this->postMapper = $postMapper;
21  }
22
23  /**
24   * {@inheritdoc}
25   */
26  public function findAllPosts()
27  {
28      return $this->postMapper->findAll();
29  }
30
31  /**
32   * {@inheritdoc}
33   */
34  public function findPost($id)
35  {
36      return $this->postMapper->find($id);
37  }
38
39  /**
40   * {@inheritdoc}
41   */
42  public function savePost(PostInterface $post)
43  {
44      return $this->postMapper->save($post);
45  }
46
47  /**
48   * {@inheritdoc}
49   */
50  public function deletePost(PostInterface $post)
51  {
52      return $this->postMapper->delete($post);
53  }
54  }

```

Now we assume that the `PostMapperInterface` has a `delete()`-function. We haven't yet implemented this one so go ahead and add it to the `PostMapperInterface`.

```

1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/PostMapperInterface.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6
7  interface PostMapperInterface
8  {
9      /**

```

```
10      * @param int|string $id
11      * @return PostInterface
12      * @throws \InvalidArgumentException
13      */
14      public function find($id);
15
16      /**
17       * @return array|PostInterface[]
18       */
19      public function findAll();
20
21      /**
22       * @param PostInterface $postObject
23       *
24       * @param PostInterface $postObject
25       * @return PostInterface
26       * @throws \Exception
27       */
28      public function save(PostInterface $postObject);
29
30      /**
31       * @param PostInterface $postObject
32       *
33       * @return bool
34       * @throws \Exception
35       */
36      public function delete(PostInterface $postObject);
37  }
```

Now that we have declared the function inside the interface it's time to implement it inside our `ZendDbSqlMapper`:

```
1  <?php
2  // Filename: /module/Blog/src/Blog/Mapper/ZendDbSqlMapper.php
3  namespace Blog\Mapper;
4
5  use Blog\Model\PostInterface;
6  use Zend\Db\Adapter\AdapterInterface;
7  use Zend\Db\Adapter\Driver\ResultInterface;
8  use Zend\Db\ResultSet\HydratingResultSet;
9  use Zend\Db\Sql\Delete;
10 use Zend\Db\Sql\Insert;
11 use Zend\Db\Sql\Sql;
12 use Zend\Db\Sql\Update;
13 use Zend\Stdlib\Hydrator\HydratorInterface;
14
15 class ZendDbSqlMapper implements PostMapperInterface
16 {
17     /**
18      * @var \Zend\Db\Adapter\AdapterInterface
19      */
20     protected $dbAdapter;
21
22     protected $hydrator;
23
24     protected $postPrototype;
25
26     /**
27      * @param AdapterInterface $dbAdapter
```

```

28      * @param HydratorInterface $hydrator
29      * @param PostInterface      $postPrototype
30      */
31      public function __construct(
32          AdapterInterface $dbAdapter,
33          HydratorInterface $hydrator,
34          PostInterface $postPrototype
35      ) {
36          $this->dbAdapter      = $dbAdapter;
37          $this->hydrator       = $hydrator;
38          $this->postPrototype = $postPrototype;
39      }
40
41      /**
42       * {@inheritdoc}
43       */
44      public function find($id)
45      {
46          $sql      = new Sql($this->dbAdapter);
47          $select    = $sql->select('posts');
48          $select->where(array('id = ?' => $id));
49
50          $stmt      = $sql->prepareStatementForSqlObject($select);
51          $result     = $stmt->execute();
52
53          if ($result instanceof ResultInterface && $result->isQueryResult() &&
↳$result->getAffectedRows()) {
54              return $this->hydrator->hydrate($result->current(), $this->
↳postPrototype);
55          }
56
57          throw new \InvalidArgumentException("Blog with given ID:{$id} not found.");
58      }
59
60      /**
61       * {@inheritdoc}
62       */
63      public function findAll()
64      {
65          $sql      = new Sql($this->dbAdapter);
66          $select    = $sql->select('posts');
67
68          $stmt      = $sql->prepareStatementForSqlObject($select);
69          $result     = $stmt->execute();
70
71          if ($result instanceof ResultInterface && $result->isQueryResult()) {
72              $resultSet = new HydratingResultSet($this->hydrator, $this->
↳postPrototype);
73
74              return $resultSet->initialize($result);
75          }
76
77          return array();
78      }
79
80      /**
81       * {@inheritdoc}
82       */

```

```

83     public function save(PostInterface $postObject)
84     {
85         $postData = $this->hydrator->extract($postObject);
86         unset($postData['id']); // Neither Insert nor Update needs the ID in the_
        ↪array
87
88         if ($postObject->getId()) {
89             // ID present, it's an Update
90             $action = new Update('post');
91             $action->set($postData);
92             $action->where(array('id = ?' => $postObject->getId()));
93         } else {
94             // ID NOT present, it's an Insert
95             $action = new Insert('post');
96             $action->values($postData);
97         }
98
99         $sql = new Sql($this->dbAdapter);
100         $stmt = $sql->prepareStatementForSqlObject($action);
101         $result = $stmt->execute();
102
103         if ($result instanceof ResultInterface) {
104             if ($newId = $result->getGeneratedValue()) {
105                 // When a value has been generated, set it on the object
106                 $postObject->setId($newId);
107             }
108
109             return $postObject;
110         }
111
112         throw new \Exception("Database error");
113     }
114
115     /**
116     * {@inheritdoc}
117     */
118     public function delete(PostInterface $postObject)
119     {
120         $action = new Delete('posts');
121         $action->where(array('id = ?' => $postObject->getId()));
122
123         $sql = new Sql($this->dbAdapter);
124         $stmt = $sql->prepareStatementForSqlObject($action);
125         $result = $stmt->execute();
126
127         return (bool) $result->getAffectedRows();
128     }
129 }
    
```

The Delete statement should look fairly similar to you as this is basically the same deal as all other queries we've created so far. With all of this set up now we're good to go ahead and write our view file so we can delete blogs.

```

1  <!-- Filename: /module/Blog/view/blog/delete/delete.phtml -->
2  <h1>DeleteController::deleteAction()</h1>
3  <p>
4      Are you sure that you want to delete
5      '<?php echo $this->escapeHtml($this->post->getTitle()); ?>' by
6      '<?php echo $this->escapeHtml($this->post->getText()); ?>'?
    
```



```
7 </p>
8 <form action="<?php echo $this->url('blog/delete', array(), true) ?>" method="post">
9     <input type="submit" name="delete_confirmation" value="yes">
10    <input type="submit" name="delete_confirmation" value="no">
11 </form>
```

Summary

In this chapter we've learned how data binding within the `Zend\Form`-component works and through it we have finished our update-routine. Then we have learned how we can use HTML-Forms and checking it's data without relying on `Zend\Form`, which ultimately lead us to having a full CRUD-Routine for the Blog example.

In the next chapter we'll recapitulate everything we've done. We'll talk about the design-patterns we've used and we're going to cover a couple of questions that highly likely arose during the course of this tutorial.

Reviewing the Blog-application

Throughout the past seven chapters we have created a fully functional CRUD-Application using music-blogs as an example. While doing so we've made use of several different design-patterns and best-practices. Now it's time to reiterate and take a look at some of the code-samples we've written. This is going to be done in a Q&A fashion.

- *Do we always need all the layers and interfaces?*
- *Having many objects, won't there be much code-duplication?*
- *Why are there so many controllers?*

Do we always need all the layers and interfaces?

Short answer: no.

Long answer: The importance of interfaces goes up the bigger your application becomes. If you can foresee that your application will be used by other people or is supposed to be extendable, then you should strongly consider to always code against interfaces. This is a very common best-practice that is not tied to ZF2 specifically but rather aimed at strict OOP programming.

The main role of the multiple layers that we have introduced (**Controller** -> **Service** -> **Mapper** -> **Backend**) are to get a strict separation of concerns for all of our objects. There are many resources who can explain in detail the big advantages of each layer so please go ahead and read up on them.

For a very simple application, though, you're most likely to strip away the **Mapper**-layer. In practice all the code from the mapper layer often resides inside the services directly. And this works for most of the applications but as soon as you plan to support multiple backends (i.e. open source software) or you want to be prepared for changing backends, you should always consider including this layer.

Having many objects, won't there be much code-duplication?

Short answer: yes.

Long answer: there doesn't need to be. Most code-duplication would come from the mapper-layer, too. If you take a closer look at the class you'll notice that there's just two things that are tied to a specific object. First, it is the name of the database-table. Second, it is the object-prototype that's passed into the mapper.

The prototype is already passed into the class from the `__construct()` function so that's already interchangeable. If you want to make the table-name interchangeable, too, all you need to do is to provide the table-name from the constructor, too, and you have a fully versatile db-mapper-implementation that can be used for pretty much every object of your application.

You could then write a factory class that could look like this:

```
1 <?php
2
3 class NewsMapperFactory implements FactoryInterface
4 {
5     public function createService(ServiceLocatorInterface $serviceLocator)
6     {
7         return new ZendDbSqlMapper(
8             $serviceLocator->get('Zend\Db\Adapter\Adapter'), // DB-Adapter
9             'news', // Table-Name
10            new ClassMethods(false), // Object-Hydrator
11            new News() // Object-Prototype
12        );
13    }
14 }
```

Why are there so many controllers?

Looking back at code-examples from a couple of years back you'll notice that there was a lot of code inside each controller. This has become a bad-practice that's known as Fat Controllers or Bloated Controllers.

The major difference about each controller we have created is that there are different dependencies. For example, the `WriteController` required the `PostForm` as well as the `PostService` while the `DeleteController` only required the `PostService`. In this example it wouldn't make sense to write the `deleteAction()` into the `WriteController` because we then would needlessly create an instance of the `PostForm` which is not required. In large scale applications this would create a huge bottleneck that would slow down the application.

Looking at the `DeleteController` as well as the `ListController` you'll notice that both controllers have the same dependency. Both require only the `PostService` so why not merge them into one controller? The reason here is for semantical reasons. Would you look for a `deleteAction()` in a `ListController`? Most of us wouldn't and therefore we have created a new class for that.

In applications where the `InsertForm` differs from the `UpdateForm` you'd always want to have two different controllers for each of them instead of one united `WriteController` like we have in our example. These things heavily differ from application to application but the general intent always is: **keep your controllers slim / lightweight!**

Do you have more questions? PR them!

If there's anything you feel that's missing in this FAQ, please PR your question and we will give you the answer that you need!

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.

We will develop this application using Zend Studio 10 and run the application on Zend Server 6.

Zend Server is a PHP application server that includes the PHP runtime. It comes in both free and paid editions, both of which provide lots of features; however the most interesting ones for developers are the dead-simple environment setup and the ability to investigate application problems, including profiling performance and memory issues with code-tracing abilities. Zend Server also ships with Zend Framework 2, which is convenient.

Zend Studio is a PHP-focused IDE based on Eclipse that comes in two flavours: the free Eclipse PDT and Zend Studio, a paid-for product that provides enhanced features and support options. Usefully, Eclipse PDT provides Zend Framework 2 support out of the box along with Zend Server integration. You don't get the mobile features though, or integrated PHP Documenter & PHPUnit features.

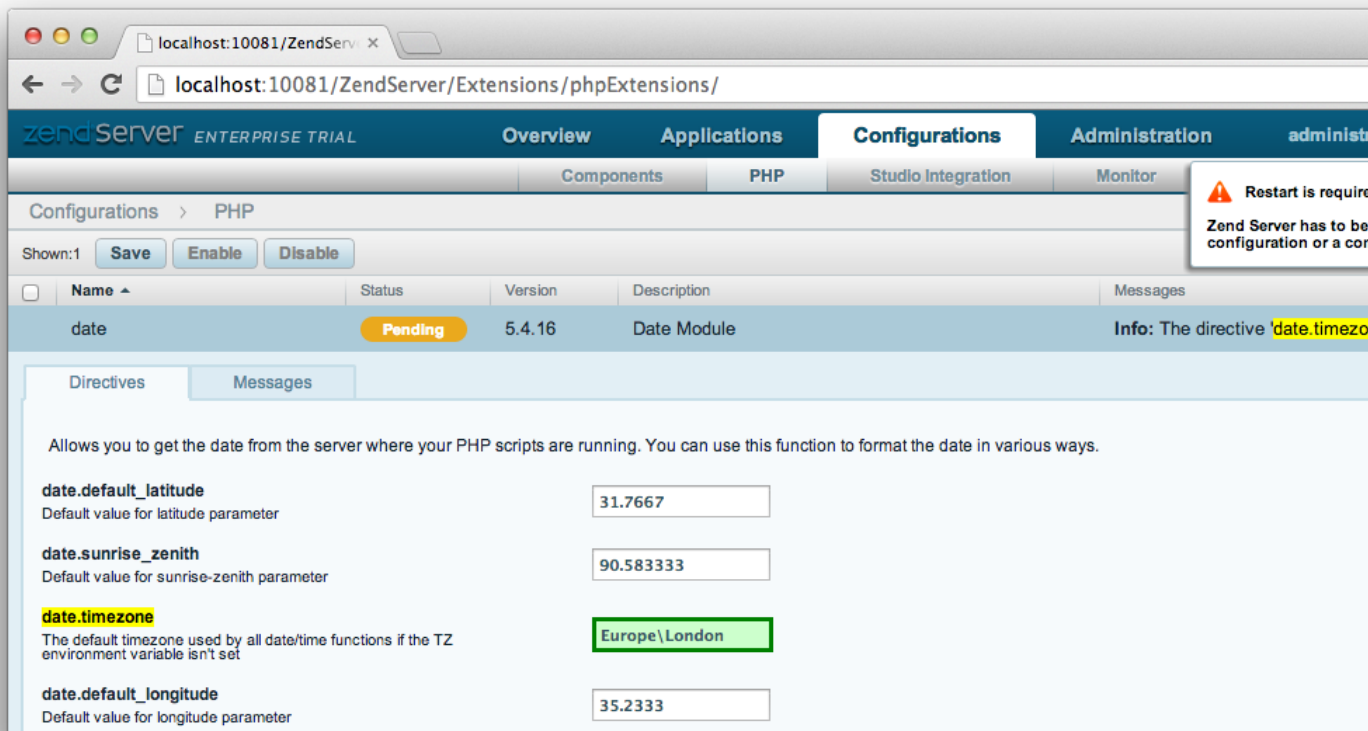
In this tutorial we're going to build a small, simple database application to manage a list of to-do items. We'll need a list of items along with the ability to add, edit and delete items. We'll use a database to store information about each to-do item.

Installation

Firstly you'll need to install Zend Server and Eclipse PDT. If you have a license for Zend Studio 10, you can use that too. You can download the latest version of [Zend Server](#). Grab [Eclipse PDT](#) or [Zend Studio](#) (which comes with a free 30-day trial) and install it. In this tutorial we will use the phrase Zend Studio, but it will all work with Eclipse PDT too.

On Linux, you can install Zend Server with either Apache or Nginx. This tutorial has assumed that you have installed the Apache version. The only important difference for this tutorial is the creation of rewrite rules.

Once you have installed Zend Server, enter the administration application, which can usually be found at <http://localhost:10081/>. Set the time zone in Configuration -> PHP, and then restart the server (third button from the right in the top right corner).



You will also need to install MySQL using your Linux distribution's package manager or from mysql.com if you are on Windows. For OS X users, Zend Server already includes MySQL for you.

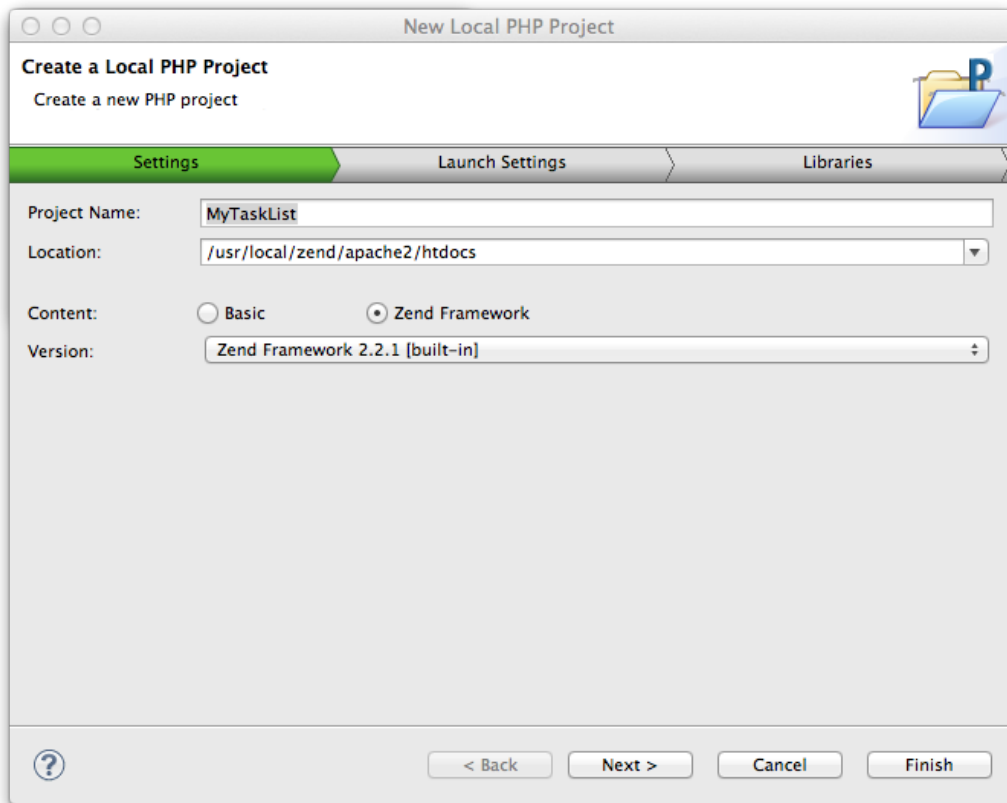
On OS X, the document root for the Zend Server installed Apache is at `/usr/local/zend/apache2/htdocs`. On Linux, Zend Server uses the web server supplied by the distribution. On Ubuntu 12.04, with Apache, it is `/var/www` and with nginx it is at `/usr/share/nginx/html`. On Windows, it is `C:\Program Files (x86)\Zend\Apache2\htdocs`.

Ensure that this folder is writeable by your own user. The easiest way to do this is to change the owner of the html directory. On a Mac, this would be:

```
$ sudo chown {your username} /usr/local/zend/apache2/htdocs
```

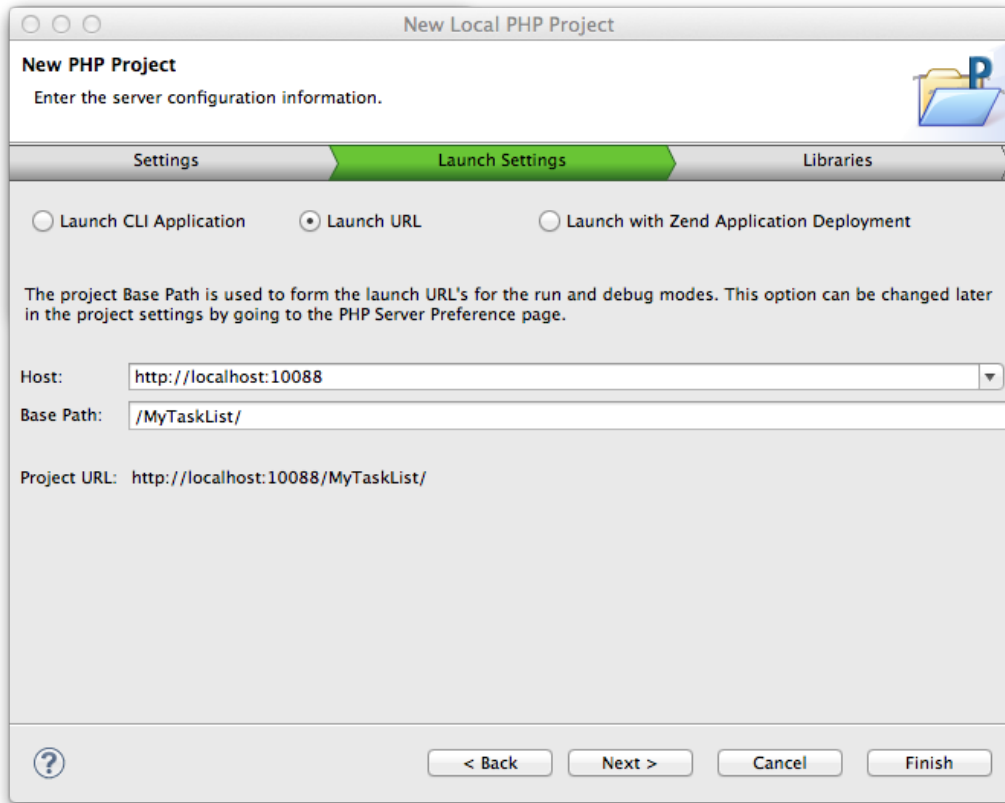
Getting Started

We start by creating a new Local PHP project in Zend Studio. Open Zend Studio and select File -> New -> Local PHP Project. This will display the New Local PHP Project wizard as shown:



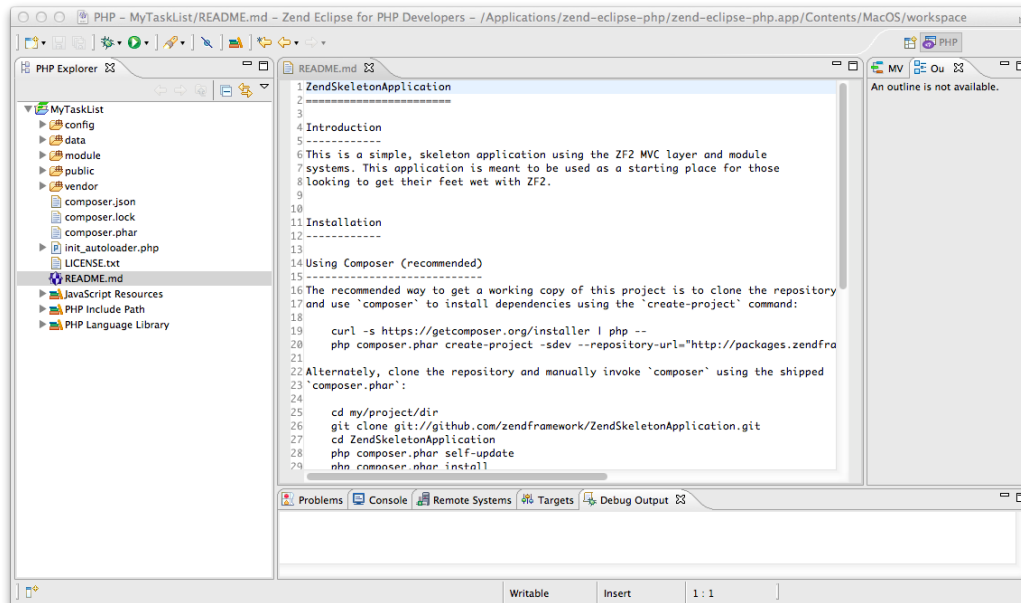
Enter *MyTaskList* as the Project Name and set the location to the Zend Server document root. Due to the integration between Zend Server and Zend Studio, you should find the correct directory as an option in the drop down list. Select Zend Framework as the Content and you can then select which version of Zend Framework to use. Select the latest Zend Framework 2 version and press Next.

The next step is the Launch Settings tab. Choose Launch URL and set the host to <http://localhost> (or <http://localhost:10088> on OS X) and the Base Path to `/MyTaskList/`:



Press Finish to create your new project in Zend Studio.

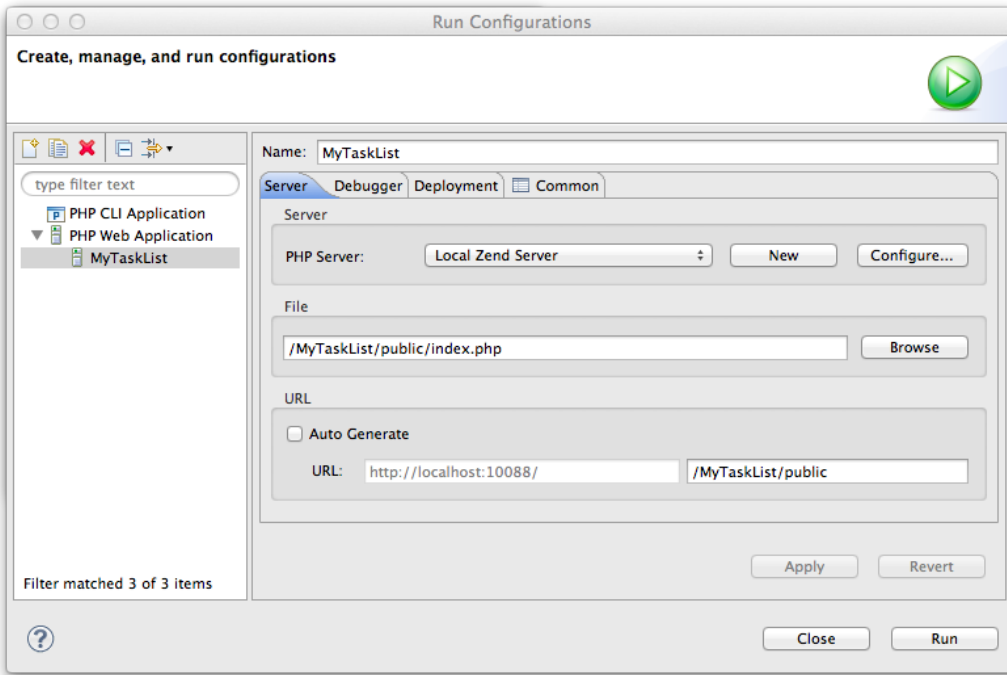
Zend Studio has now created a default Zend Framework project for us:



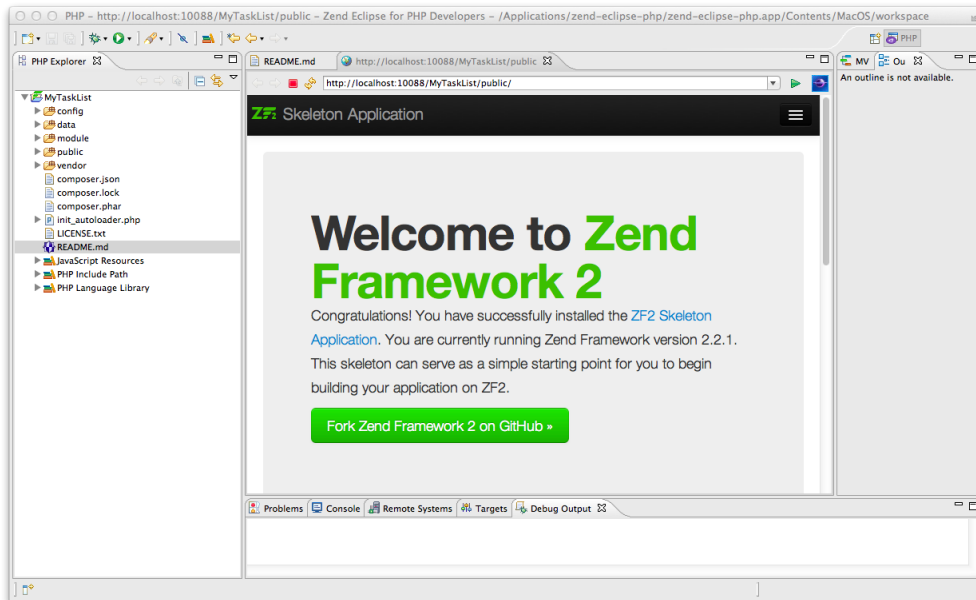
This is a standard Zend Framework 2 Skeleton Application and is a great starting point for a new ZF2 application.

To set up Zend Studio to run this project, select Run -> Run Configurations... and double click on PHP Web Application in the left hand list. Enter MyTaskList as the name, Local Zend Server as the PHP Server and then click the

Browse button and select `index.php` within the `public` folder of the `MyTaskList` project. Uncheck `Auto Generate` in the `URL` section and then set the path to `/MyTaskList/public` and press `Apply` and then `Close`:



To test that all is working, press the run button in the toolbar (white arrow in a green circle). The ZF2 Skeleton Application home page will display in a new tab within Zend Studio:



You can also navigate to the same URL (`http://localhost:10088/MyTaskList/public/` on a Mac) in any browser.

We have successfully installed both Zend Server and Zend Studio, created a project and tested it. Let's start by looking at what we have so far in our Zend Framework project.

A quick tour of the skeleton application

The skeleton application provides a lot of files, so it's worth having a quick high-level look at what has been generated for us. There are a number of high level directories created for us (along with [Composer](#) and other support files):

Folder	Information stored
<code>config</code>	Application-level configuration files.
<code>data</code>	Data files generated by the application, such as caches.
<code>module</code>	The source files that make up this application are stored within separate modules within this folder.
<code>public</code>	The web server's document root. All files served directly by the web server are in here.
<code>vendor</code>	Third party libraries.

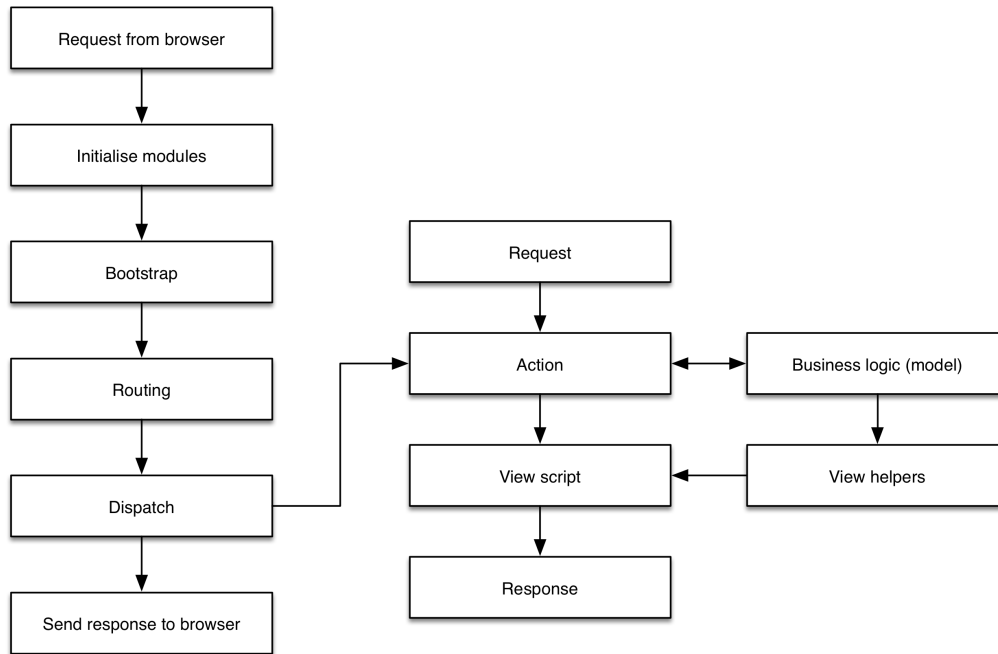
One of the key features of Zend Framework 2 is its module system. This provides organisation within your application; all application code lives within a module. The skeleton provides the Application module for bootstrapping, error and routing configuration. It also provides the application-level controllers for the home page and error display. The Application module contains these key folders:

Folder	Information stored
<code>config</code>	Module-specific configuration files.
<code>language</code>	Translation files.
<code>src/Application</code>	PHP files for this module, including controller and model files. The controller for the home page, <code>IndexController.php</code> , is provided.
<code>view/application</code>	View scripts for each controller action.
<code>view/error</code>	Error view scripts for 404 and generic errors.
<code>view/layout</code>	Layout view scripts. These contain the common HTML shared by a number of pages within the website. An initial default file, <code>layout.phtml</code> , is provided.

Modules are simply namespaces containing a top level `Module` class. They are intended to be reusable and no additional constraints are placed on how they are organised. An application consists of multiple modules, both third party and application specific, with the list of modules to load stored in `config/application.config.php`.

The dispatch cycle

Zend Framework 2 applications use the [Front Controller](#) design pattern. This means that all requests are directed to a single entry point, the `public/index.php` file. This is done using a `.htaccess` file containing rewrite rules that serves all static files (such as CSS & Javascript) and directs all other requests to the `index.php`. The `index.php` file initialises the autoloader and then bootstraps `Zend\Mvc\Application` before finally running the application. The process looks like this:



Starting up

To set up the application for running, a number of things happen. Firstly an instance of `Zend\ServiceManager` is created as the master locator for all class instances used by the application. The Module Manager is then used to load all the application's modules. It does this by reading its configuration file, `application.config.php`, which is solely for use by the Module Manager and does not contain the configuration used by the application itself.

The modules are loaded in the order listed in the configuration file and for each module a number of steps takes place:

- Configuration of autoloading.
- Loading of module configuration.
- Registration of event listeners.
- Configuration of the Service Manager.

The configuration information from all modules is merged together into one configuration array. This means that configuration information in subsequent modules can override information already set. Finally, the global configuration files stored in the `config/autoload` directory are merged (the `*.global.php` and then the `*.local.php` files). This means that any module's configuration can be overridden at the application level and is a key feature that helps to ensure that the code within a third-party module does not need to be changed.

The Service Manager and Event Manager are two other key features of a Zend Framework 2 application. [*Zend\ServiceManager*](#) allows for decoupling the instantiation and configuration of a class and its dependencies from

where that class is used. This is known as Dependency Injection and is used extensively in Zend Framework 2. *ZendEventManager* is an implementation of the Observer design pattern which allows decoupling of code. In Zend Framework 2, every key process in the dispatch cycle is implemented as an event. This means that you can write listeners for these events which can then change the flow of operation or perform additional processes when something else has happened.

Dispatching

Once all modules have been loaded, the application is run. This is done as a series of events, with the first event, *route*, used to determine the controller action that should be run based on the URL requested. Once this is determined, the *dispatch* event is triggered which causes the action method within the controller class to be executed. The view rendering event, *render*, is then triggered if an HTML view is required. Finally the *finish* event is triggered which sends the response back to the user's web browser.

While this is a typical dispatch cycle, Zend Framework 2's dispatch system is very flexible and can be configured in a variety of ways depending on the specific application. Now that we've looked at how Zend Framework works, let's move on and write the *MyTaskList* application.

CHAPTER 22

The MyTaskList application

The application we are going to create is a to-do list manager. The application will allow us to create to-do items and check them off. We'll also need the ability to edit and delete an item. As we are building a simple application, we need just four pages:

Page	Notes
Checklist homepage	This will display the list of to-do items.
Add new item	This page will provide a form for adding a new item.
Edit item	This page will provide a form for editing an item.
Delete item	This page will confirm that we want to delete an item and then delete it.

Each page of the application is known as an action, and actions are grouped into controllers within modules. Generally, related actions are placed into a single controller; for instance, a news controller might have actions of `current`, `archived` and `view`.

We will store information about our to-do items in a database. A single table will suffice with the following fields:

Field name	Type	Null?	Notes
id	integer	No	Primary key, auto-increment
title	varchar(100)	No	Name of the file on disk
completed	tinyint	No	Zero if not done, one if done
created	datetime	No	Date that the to-do item was created

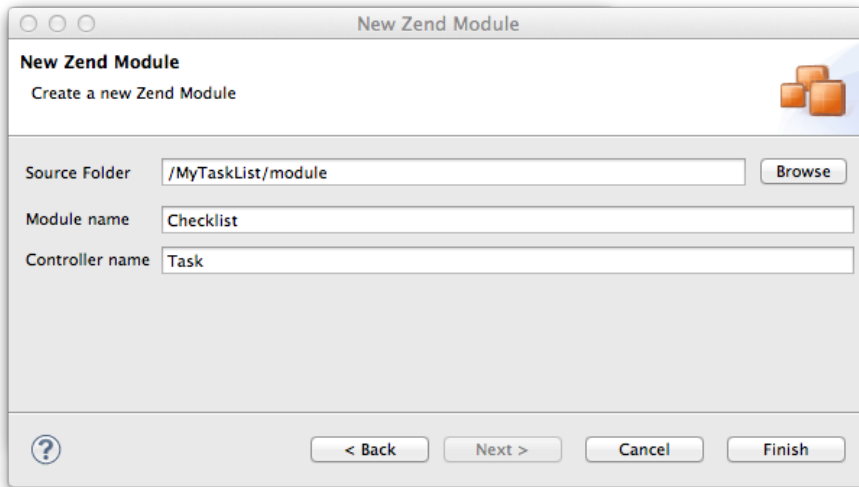
We are going to use MySQL, via PHP's PDO driver, so create a database called `mytasklist` using your preferred MySQL client, and run these SQL statements to create the `task_item` table and some sample data:

Note that if you have Zend Studio, you can use the built-in Database Connectivity features. This is found in the Database Development perspective (Window | Open Perspective | Other | Database Development menu item) and further details are in the [Zend Studio manual](#).

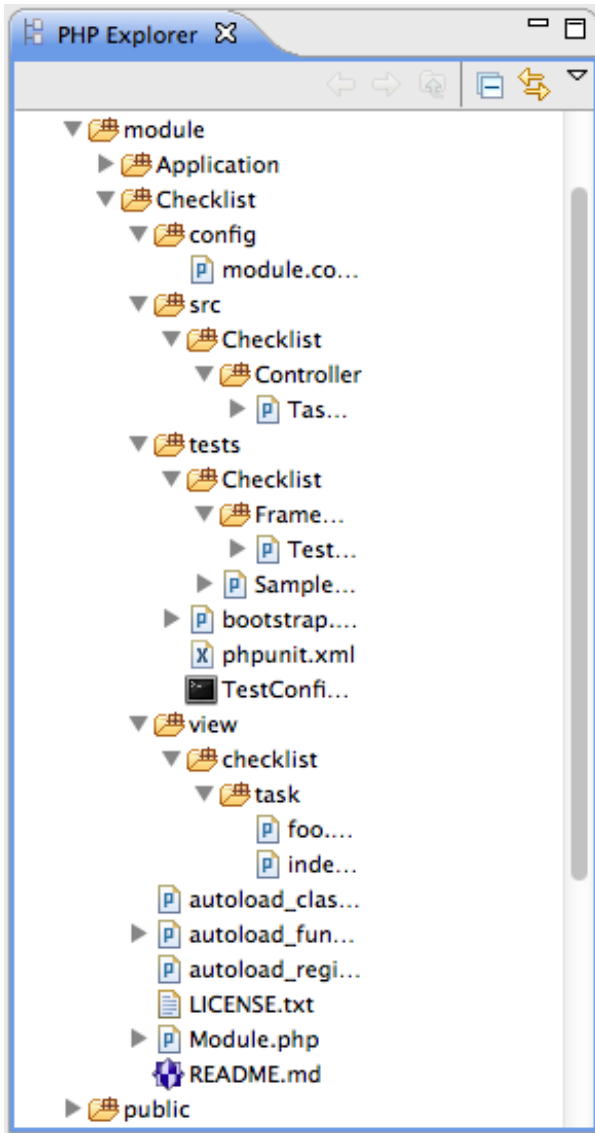
The Checklist module

We will create all our code within a module called *Checklist*. The `Checklist` module will, therefore, contain our controllers, models, forms and views, along with specific configuration files.

We create our new `Checklist` module in Zend Studio. In the PHP Explorer on the left, right click on the `MyTaskList` project folder and choose `New -> Zend Framework Item`. Click on `Zend Module` and press `Next`. The `Source Folder` should already be set to `/MyTaskList/module`. Enter `Checklist` as the `Module name` and `Task` as the `Controller name` and then press `Finish`:



The wizard will now go ahead and create a blank module for us and register it with the Module Manager's `application.config.php`. You can see what it has done in the PHP Explorer view under the `module` folder:



As you can see the Checklist module has separate directories for the different types of files we will have. The `config` folder contains configuration files, and the PHP files that contain classes within the `Checklist` namespace live in the `src/Checklist` directory. The `view` directory also has a sub- folder called `checklist` for our module's view scripts, and the `tests` folder contains PHPUnit test files.

The Module class

As mentioned earlier, a module's `Module` class contains methods that are called during the start-up process and is also used to register listeners that will be triggered during the dispatch process. The `Module` class created for us contains three methods: `getAutoloaderConfig()`, `getConfig()` and `onBootstrap()` which are called by the Module Manager during start-up.

Autoloading files

Our `getAutoloaderConfig()` method returns an array that is compatible with ZF2's `AutoloaderFactory`. It is configured for us with both a classmap file (`autoload_classmap.php`) and a standard autoloader to load any files in `src/Checklist` according to the [PSR-0](#) rules.

Classmap autoloading is faster, but requires adding each new class you create to the array within the `autoload_classmap.php` file, which slows down development. The standard autoloader, however, doesn't have this requirement and will always load a class if its file is named correctly. This allows us to develop quickly by creating new classes when we need them and then gain a performance boost by using the classmap autoloader in production. Zend Framework 2 provides `bin/classmap_generator.php` to create and update the file.

Configuration

The `getConfig()` method in `Checklist\Module` is called by the Module Manager to retrieve the configuration information for this module. By tradition, this method simply loads the `config/module.config.php` file which is an associative array. In practice, the Module Manager requires that the returned value from `getConfig()` be a `Traversable`, which means that you can use any configuration format that `Zend\Config` supports. You will find, though, that most examples use arrays as they are easy to understand and fast.

The actual configuration information is placed in `config/module.config.php`. This nested array provides the key configuration for our module. The `controllers` sub-array is used to register this module's controller classes with the Controller Service Manager which is used by the dispatcher to instantiate a controller. The one controller that we need, `TaskController`, is already registered for us.

The `router` sub-array provides the configuration of the routes that are used by this module. A route is the way that a URL is mapped to a particular action method within a controller class. Zend Studio's default configuration is set up so that a URL of `/checklist/foo/bar` maps to the `barAction()` method of the `FooController` within the `Checklist` module. We will modify this later.

Finally, the `view_manager` sub-array within the `module.config.php` file is used to register the directory where our view files are with the View sub-system. This means that within the `view/checklist` sub-folder, there is a folder for each controller. We have one controller, `TaskController`, so there is a single sub-folder in `view/checklist` called `task`. Within this folder, there are separate `.phtml` files which contain the specific HTML for each action of our module.

Registering events

The `onBootstrap()` method in the `Module` class is the easiest place to register listeners for the MVC events that are triggered by the Event Manager. Note that the default method body provided by Zend Studio is not needed as the `ModuleRouteListener` is already registered by the `Application` module. We do not have to register any events for this tutorial, so go ahead and delete the entire `onBootstrap()` method.

The application's pages

As we have four pages that all apply to tasks, we will group them in a single controller called `TaskController` within our `Checklist` module as four actions. Each action has a related URL which will result in that action being dispatched. The four actions and URLs are:

Page	URL	Action
Homepage	/task	index
Add new task	/task/add	add
Edit task	/task/edit	edit
Delete task	/task/delete	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. As noted earlier, the configuration file, `module.config.php` created by Zend Studio has a route called `checklist` set up for us.

Routing

The default route provided for us isn't quite what we need. The `checklist` route is defined like this:

module/Checklist/src/config/module.config.php:

```
1  'router' => array(  
2      'routes' => array(  
3          'checklist' => array(  
4              'type' => 'Literal',  
5              'options' => array(  
6                  'route' => '/task',  
7                  'defaults' => array(  
8                      '__NAMESPACE__' => 'Checklist\Controller',  
9                      'controller' => 'Task',  
10                     'action' => 'index',  
11                 ),  
12             ),  
13             'may_terminate' => true,
```

```
14         'child_routes' => array(  
15             'default' => array(  
16                 'type' => 'Segment',  
17                 'options' => array(  
18                     'route' => '[:controller[:action]]',  
19                 ),  
20             ),  
21         ),  
22     ),
```

This defines a main route called `checklist`, which maps the URL `/task` to the `index` action of the `Task` controller and then there is a child route called `default` which maps `/task/{controller name}/{action name}` to the `{action name}` action of the `{controller name}` controller. This means that, by default, the URL to call the `add` action of the `Task` controller would be `/task/task/add`. This doesn't look very nice and we would like to shorten it to `/task/add`.

To fix this, we will rename the route from `checklist` to `task` because this route will be solely for the `Task` controller. We will then redefine it to be a single `Segment` type route that can handle actions as well as just route to the `index` action

Open `module/Checklist/config/module.config.php` in Zend Studio and change the entire router section of the array to be:

module/Checklist/src/config/module.config.php:

```
1  'router' => array(  
2      'routes' => array(  
3          'task' => array(  
4              'type' => 'Segment',  
5              'options' => array(  
6                  'route' => '/task[:action[:id]]',  
7                  'defaults' => array(  
8                      '__NAMESPACE__' => 'Checklist\Controller',  
9                      'controller' => 'Task',  
10                     'action' => 'index',  
11                 ),  
12                 'constraints' => array(  
13                     'action' => '(add|edit|delete)',  
14                     'id' => '[0-9]+',  
15                 ),  
16             ),  
17         ),  
18     ),  
19 ),
```

We have now renamed the route to `task` and have set it up as a `Segment` route with two optional parameters in the URL: `action` and `id`. We have set a default of `index` for the `action`, so that if the URL is simply `/task`, then we shall use the *index* action in our controller.

The optional `constraints` section allow us to specify regular expression patterns that match the characters that we expect for a given parameter. For this route, we have specified that the `action` parameter must be either `add`, `edit` or `delete` and that the `id` parameter must only contain numbers.

The routing for our `Checklist` module is now set up, so we can now turn our attention to the controller.

The TaskController

In Zend Framework 2, the controller is a class that is generally called `{Controller name}Controller`. Note that `{Controller name}` starts 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's the `module/Checklist/src/Checklist/Controller` directory. Each action is a public function within the controller class that is named `{action name}Action`. In this case `{action name}` should start with a lower case letter.

Note that this is merely a convention. Zend Framework 2's only restrictions on a controller is that it must implement the `Zend\Stdlib\Dispatchable` interface. The framework provides two abstract classes that do this for us: `Zend\Mvc\Controller\ActionController` and `Zend\Mvc\Controller\RestfulController`. We'll be using the `AbstractActionController`, but if you're intending to write a RESTful web service, `AbstractRestfulController` may be useful.

Zend Studio's module creation wizard has already created `TaskController` for us with two action methods in it: `indexAction()` and `fooAction()`. Remove the `fooAction()` method and the default "Copyright Zend" DocBlock comment at the top of the file. Your controller should now look like this:

module/Checklist/src/Checklist/Controller/TaskController.php:

```

1  namespace Checklist\Controller;
2
3  use Zend\Mvc\Controller\AbstractActionController;
4
5  class TaskController extends AbstractActionController
6  {
7      public function indexAction()
8      {
9          return array();
10     }
11
12 }
```

This controller now contains the action for the home page which will display our list of to-do items. We now need to create a model-layer that can retrieve the tasks from the database for display.

The model

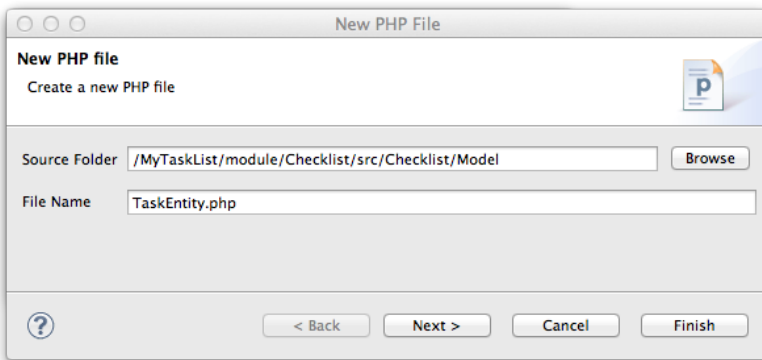
It is time to look at the model section of our application. Remember that the model is the part that deals with the application's core purpose (the so-called "business rules") and, in our case, deals with the database. Zend Framework does not provide a `Zend\Model` component because the model is your 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 fairly simple model layer using an entity and a mapper that uses the `Zend\Db` component. In a larger, more complex, application, you would probably also have a service class that interfaces between the controller and the mapper.

We already have created the database table and added some sample data, so let's start by creating an entity object. An entity object is a simple PHP object that represents a thing in the application. In our case, it represents a task to be completed, so we will call it `TaskEntity`.

Create a new folder in `module/Checklist/src/Checklist` called `Model` and then right click on the new `Model` folder and choose `New -> PHP File`. In the `New PHP File` dialog, set the `File Name` to `TaskEntity.php`

as shown and then press Finish.



This will create a blank PHP file. Update it so that it looks like this:

module/Checklist/src/Checklist/Model/TaskEntity.php:

```
1 <?php
2 namespace Checklist\Model;
3
4 class TaskEntity
5 {
6     protected $id;
7     protected $title;
8     protected $completed = 0;
9     protected $created;
10
11     public function __construct()
12     {
13         $this->created = date('Y-m-d H:i:s');
14     }
15
16     public function getId()
17     {
18         return $this->id;
19     }
20
21     public function setId($Value)
22     {
23         $this->id = $Value;
24     }
25
26     public function getTitle()
27     {
28         return $this->title;
29     }
30
31     public function setTitle($Value)
32     {
33         $this->title = $Value;
34     }
35
36     public function getCompleted()
37     {
38         return $this->completed;
39     }
40 }
```

```

40
41     public function setCompleted($Value)
42     {
43         $this->completed = $Value;
44     }
45
46     public function getCreated()
47     {
48         return $this->created;
49     }
50
51     public function setCreated($Value)
52     {
53         $this->created = $Value;
54     }
55 }

```

The Task entity is a simple PHP class with four properties with getter and setter methods for each property. We also have a constructor to fill in the created property. If you are using Zend Studio rather than Eclipse PDT, then you can generate the getter and setter methods by right clicking in the file and choosing [Source -> Generate Getters and Setters](#).

We now need a mapper class which is responsible for persisting task entities to the database and populating them with new data. Again, right click on the Model folder and choose New -> PHP File and create a PHP file called TaskMapper.php. Update it so that it looks like this:

module/Checklist/src/Checklist/Model/TaskMapper.php:

```

1  <?php
2  namespace Checklist\Model;
3
4  use Zend\Db\Adapter\Adapter;
5  use Checklist\Model\TaskEntity;
6  use Zend\Stdlib\Hydrator\ClassMethods;
7  use Zend\Db\Sql\Sql;
8  use Zend\Db\Sql>Select;
9  use Zend\Db\ResultSet\HydratingResultSet;
10
11  class TaskMapper
12  {
13      protected $tableName = 'task_item';
14      protected $dbAdapter;
15      protected $sql;
16
17      public function __construct(Adapter $dbAdapter)
18      {
19          $this->dbAdapter = $dbAdapter;
20          $this->sql = new Sql($dbAdapter);
21          $this->sql->setTable($this->tableName);
22      }
23
24      public function fetchAll()
25      {
26          $select = $this->sql->select();
27          $select->order(array('completed ASC', 'created ASC'));
28
29          $statement = $this->sql->prepareStatementForSqlObject($select);
30          $results = $statement->execute();

```

```

31         $entityPrototype = new TaskEntity();
32         $hydrator = new ClassMethods();
33         $resultset = new HydratingResultSet($hydrator, $entityPrototype);
34         $resultset->initialize($results);
35         return $resultset;
36     }
37 }
38

```

Within this mapper class we have implemented the `fetchAll()` method and a constructor. There's quite a lot going on here as we're dealing with the `Zend\Db` component, so let's break it down. Firstly we have the constructor which takes a `Zend\Db\Adapter\Adapter` parameter as we can't do anything without a database adapter. `Zend\Db\Sql` is an object that abstracts SQL statements that are compatible with the underlying database adapter in use. We are going to use this object for all of our interaction with the database, so we create it in the constructor.

The `fetchAll()` method retrieves data from the database and places it into a `HydratingResultSet` which is able to return populated `TaskEntity` objects when iterating. To do this, we have three distinct things happening. Firstly we retrieve a `Select` object from the `Sql` object and use the `order()` method to place completed items last. We then create a `Statement` object and execute it to retrieve the data from the database. The `$results` object can be iterated over, but will return an array for each row retrieved but we want a “`TaskEntity`” object. To get this, we create a `HydratingResultSet` which requires a hydrator and an entity prototype to work.

The hydrator is an object that knows how to populate an entity. As there are many ways to create an entity object, there are multiple hydrator objects provided with ZF2 and you can create your own. For our `TaskEntity`, we use the `ClassMethods` hydrator which expects a getter and a setter method for each column in the resultset. Another useful hydrator is `ArraySerializable` which will call `getArrayCopy()` and `populate()` on the entity object when transferring data. The `HydratingResultSet` uses the prototype design pattern when creating the entities when iterating. This means that instead of instantiating a new instance of the entity class on each iteration, it clones the provided instantiated object. See <http://ralphschindler.com/2012/03/09/php-constructor-best-practices-and-the-prototype-pattern> for more details.

Finally, `fetchAll()` returns the result set object with the correct data in it.

Using Service Manager to configure the database credentials and inject into the controller

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

To configure the Service Manager we can either supply the name of the class to be instantiated or create a factory (closure or callback) method that instantiates the object when the Service Manager needs it. We start by implementing `getServiceConfig()` and write a closure that creates a `TaskMapper` instance. Add this method to the `Module` class:

module/Checklist/Module.php:

```

1  class Module
2  {
3      public function getServiceConfig()
4      {
5          return array(
6              'factories' => array(
7                  'TaskMapper' => function ($sm) {
8                      $dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');

```



```

9         $mapper = new TaskMapper($dbAdapter);
10        return $mapper;
11    }
12    },
13    );
14 }
15 // ...

```

Don't forget to add `use Checklist\Model\TaskMapper;` to the list of use statements at the top of the file.

The `getServiceConfig()` method returns an array of class creation definitions that are all merged together by the Module Manager before passing to the Service Manager. To create a service within the Service Manager we use a unique key name, `TaskMapper`. As this has to be unique, it's common (but not a requirement) to use the fully qualified class name as the Service Manager key name. We then define a closure that the Service Manager will call when it is asked for an instance of `TaskMapper`. We can do anything we like in this closure, as long as we return an instance of the required class. In this case, we retrieve an instance of the database adapter from the Service Manager and then instantiate a `TaskMapper` object and return it. This is an example of the [Dependency Injection pattern](#) at work as we have injected the database adapter into the mapper. This also means that Service Manager can be used as a Dependency Injection Container in addition to a Service Locator.

As we have requested an instance of `Zend\Db\Adapter\Adapter` from the Service Manager, we also need to configure the Service Manager so that it knows how to instantiate a `Zend\Db\Adapter\Adapter`. This is done using a class provided by Zend Framework called `Zend\Db\Adapter\AdapterServiceFactory` which we can configure within the merged configuration system. As we noted earlier, the Module Manager merges all the configuration from each module and then merges in the files in the `config/autoload` directory (`*.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 then use `local.php` (outside of the VCS) to store the credentials for your database.

Open `config/autoload/global.php` and replace the empty array with:

config/autoload/global.php:

```

1  return array(
2      'service_manager' => array(
3          'factories' => array(
4              'Zend\Db\Adapter\Adapter' =>
5                  'Zend\Db\Adapter\AdapterServiceFactory',
6          ),
7      ),
8      'db' => array(
9          'driver' => 'Pdo',
10         'dsn' => 'mysql:dbname=mytasklist;hostname=localhost',
11         'driver_options' => array(
12             PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
13         ),
14     ),
15 );

```

Firstly, we provide additional Service Manager configuration in the `service_manager` section. This array works exactly the same as the one in `getServiceConfig()`, except that you should not use closures in a config file as if you do Module Manager will not be able to cache the merged configuration information. As we already have an implementation for creating a `Zend\Db\Adapter\Adapter`, we use the `factories` sub-array to map the key name of `Zend\Db\Adapter\Adapter` to the string name of the factory class (`Zend\Db\Adapter\AdapterServiceFactory`) and the Service Manager will then use `Zend\Db\Adapter\AdapterServiceFactory` to instantiate a database adapter for us.

The `Zend\Db\Adapter\AdapterServiceFactory` object looks for a key called `db` in the configuration array

and uses this to configure the database adapter. Therefore, we create the `db` key in our `global.php` file with the relevant configuration data. The only data that is missing is the username and password required to connect to the database. We do not want to store this in the version control system, so we store this in the `local.php` configuration file, which, by default, is ignored by git.

Open `config/autoload/local.php` and replace the empty array with:

config/autoload/local.php:

```
1  return array(  
2      'db' => array(  
3          'username' => 'YOUR_USERNAME',  
4          'password' => 'YOUR_PASSWORD',  
5      ),  
6  );
```

Obviously you should replace `YOUR_USERNAME` and `YOUR_PASSWORD` with the correct credentials.

Now that the Service Manager can create a `TaskMapper` instance for us, we can add a method to the controller to retrieve it. Add `getTaskMapper()` to the `TaskController` class:

module/Checklist/src/Checklist/Controller/TaskController.php:

```
1  public function getTaskMapper()  
2  {  
3      $sm = $this->getServiceLocator();  
4      return $sm->get('TaskMapper');  
5  }
```

We can now call `getTaskMapper()` from within our controller whenever we need to interact with our model layer. Let's start with a list of tasks when the index action is called.

CHAPTER 24

Listing tasks

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

module/Checklist/src/Checklist/Controller/TaskController.php:

```
1 public function indexAction()
2 {
3     $mapper = $this->getTaskMapper();
4     return new ViewModel(array('tasks' => $mapper->fetchAll()));
5 }
```

You'll also need to add `use Zend\View\Model\ViewModel;` to list of use statements at the top of the file.

To provide variables to the view layer, 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}`. You can also return an array from a controller as Zend Framework will construct a `ViewModel` behind the scenes for you.

We can now fill in the `task/index.phtml` view script. Replace the contents with this new code:

module/Checklist/view/checklist/task/index.phtml:

```
1 <?php
2 $title = 'My task list';
3 $this->headTitle($title);
4 ?>
5 <h1><?php echo $this->escapeHtml($title); ?></h1>
6 <p><a href="<?php echo $this->url('task', array(
7     'action'=>'add')) ?>">Add new item</a></p>
8
9
10 <table class="table">
11 <tr>
12     <th>Task</th>
13     <th>Created</th>
```

```
14     <th>Completed?</th>
15     <th>&nbsp;</th>
16 </tr>
17 <?php foreach ($tasks as $task): ?>
18 <tr>
19     <td>
20         <a href="<?php echo $this->url('task',
21             array('action'=>'edit', 'id' => $task->getId())); ?>">
22             <?php echo $this->escapeHtml($task->getTitle()); ?></a>
23     </td>
24     <td><?php echo $this->escapeHtml($task->getCreated()); ?></td>
25     <td><?php echo $task->getCompleted() ? 'Yes' : 'No'; ?></td>
26     <td>
27         <a href="<?php echo $this->url('task',
28             array('action'=>'delete', 'id' => $task->getId())); ?>">Delete</a>
29     </td>
30 </tr>
31 <?php endforeach; ?>
32 </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 item using the `url()` view helper.

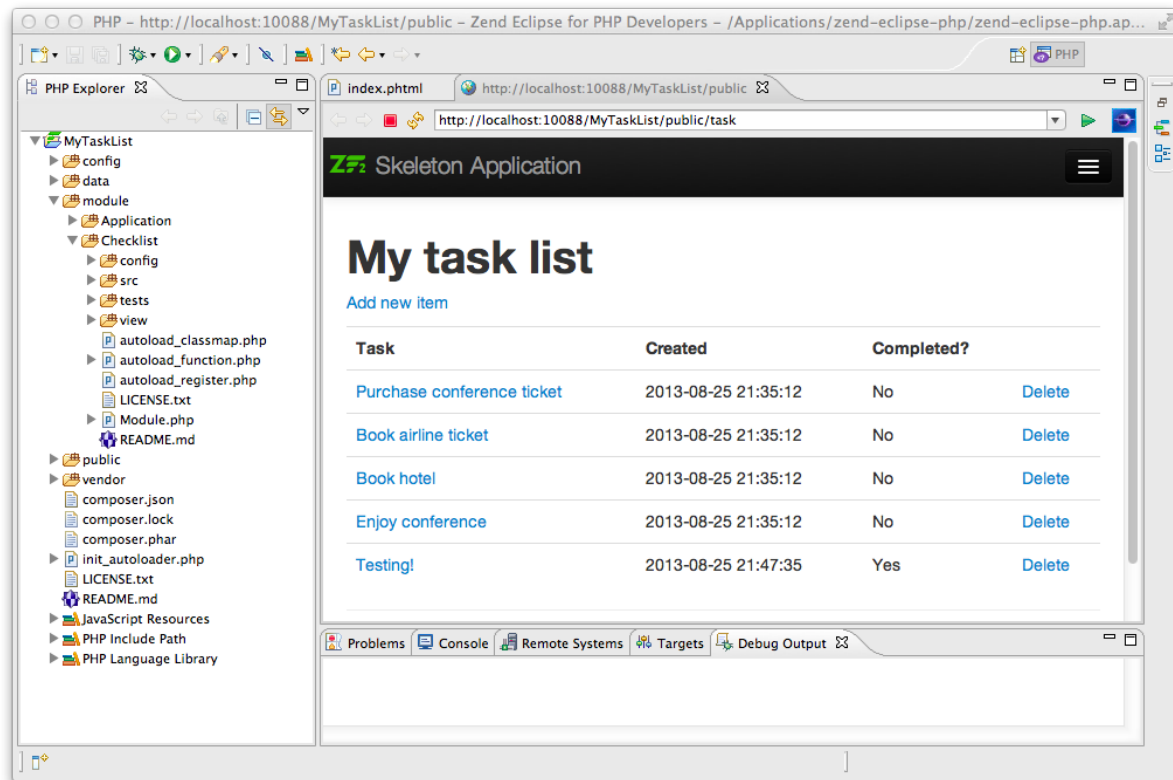
The `url()` view helper is provided by Zend Framework and is used to create the links we need. The first parameter to `url()` is the route name that we wish to use for construction of the URL and then the second parameter is an array of all the variables to fit into the place-holders to use. In this case we use our task route which is set up to accept two place-holder variables: *action* and *id*.

We iterate over the `$tasks` that we assigned from the controller action within an HTML table. The Zend Framework view system automatically ensures that these variables are extracted into the scope of the view script. Alternatively, you can also prefix with `$this->` if you would like.

For each row, we display each task's title, creation date, completion date and provide links to allow for editing and deleting the record. A standard `foreach:` loop is used to iterate over the list of tasks, 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 that we always use the `escapeHtml()` view helper to help protect ourselves from [XSS vulnerabilities](#).

If you now run the application from within Zend Studio and navigate to <http://localhost:10088/MyTaskList/public/task> you should see this:



Redirect the home page

When you first pressed the Run button, you saw the application's home page which is the skeleton's welcome page. It would be helpful if we could redirect immediately to `/tasks` to save us having to edit the URL each time.

To do this, go to Navigate -> Open Type... in Zend Studio and type `IndexController` in the search box of the Open PHP Type dialog and press return. This will open `module/Application/src/Application/Controller/IndexController.php` for you. Change the `indexAction()` method so that it reads:

module/Application/src/Application/Controller/IndexController.php:

```

1 public function indexAction()
2 {
3     return $this->redirect()->toRoute('task');
4 }

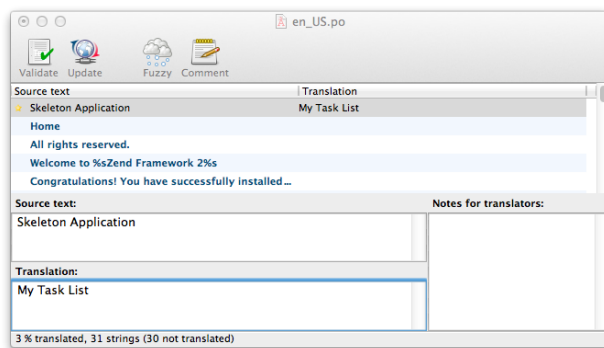
```

We use the `redirect` controller plugin to redirect the request for the home page to the URL defined by the route name `task` which we set up earlier. Now, when you press the green "Run" button, you will be taken directly to the list of tasks.

We've picked up the skeleton application's layout which is fine for this tutorial, but we need to change the title and remove the copyright message.

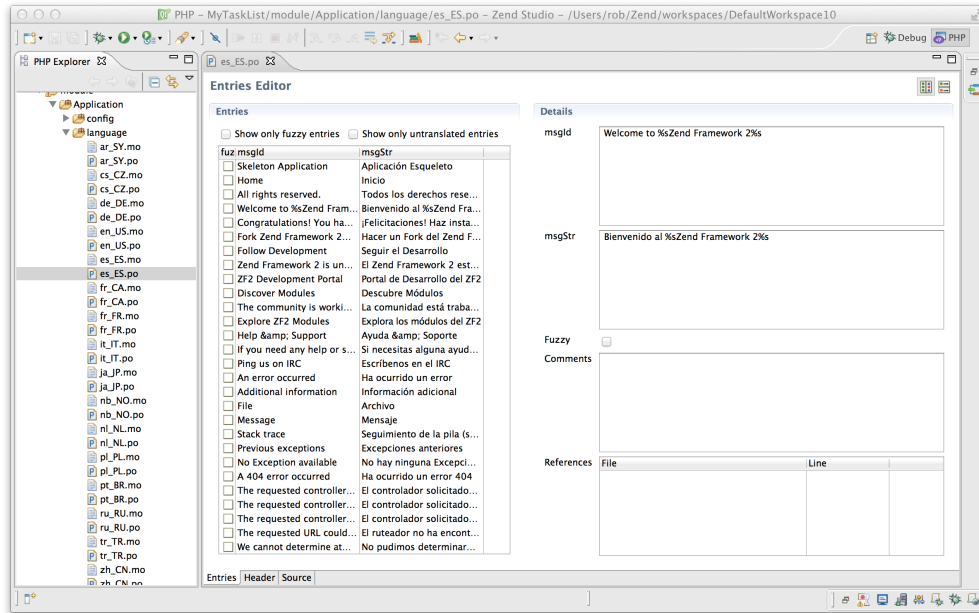
The Zend Skeleton Application is set up to use Zend\I18n's [translation functionality](#) for all the text. This allows you to translate all the text strings in the application into a different language if you need to.

The translation data is stored in separate files in the [gettext](#) format which have the extension `.po` and are stored in the `application/language` folder. The title of the application is "Skeleton Application" and to change this, you need to use the *poedit* application (<http://www.poedit.net/download.php/>). Start *poedit* and open `application/language/en_US.po`. Click on "Skeleton Application" in the list of original strings and then type in "My Task List" as the translation.



Press Save in the toolbar and *poedit* will create an updated `en_US.mo` file.

Alternatively, the *gtd* Eclipse plugin allows for editing PO files directly in Zend Studio or PDT. To install *gtd*, select the Help > Install New Software menu, and press the "Add..." button. Enter the *gtd* for the Name, <http://gtd.sourceforge.net/update> as the Location and then press the "OK" button. You will see the *gtd* name appear in the list. Click on the checkbox next to *gtd* and work through the install wizard by pressing "Next" button as required. At the end of the installation you will be able to create or edit the PO files using the *gtd* plugin:



It follows that as Zend Studio and PDT are based on Eclipse you can install any other Eclipse plugins that are listed on <http://marketplace.eclipse.org/> using the same process.

The next thing to do is 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 - <?php echo date('Y') ?> by Zend Technologies Ltd. <?php echo $this->
    translate('All rights reserved.') ?></p>
```

The page looks a little better now!

CHAPTER 26

Adding new tasks

We can now write the functionality to add new tasks. There are two things we need to do:

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

We use `Zend\Form` to do this. The `Zend\Form` component manages the form and works in tandem with the `Zend\InputFilter` component which will provide validation.

Create a new folder in `module/Checklist/src/Checklist` called `Form` and then within the `Form` folder, create a new PHP file called `TaskForm.php` with these contents:

module/Checklist/src/Checklist/Form/TaskForm.php:

```
1  <?php
2  namespace Checklist\Form;
3
4  use Zend\Form\Form;
5  use Zend\Stdlib\Hydrator\ClassMethods;
6
7  class TaskForm extends Form
8  {
9      public function __construct($name = null, $options = array())
10     {
11         parent::__construct('task');
12
13         $this->setAttribute('method', 'post');
14         $this->setInputFilter(new TaskFilter());
15         $this->setHydrator(new ClassMethods());
16
17         $this->add(array(
18             'name' => 'id',
19             'type' => 'hidden',
20         ));
21
22         $this->add(array(
23             'name' => 'title',
```

```

24         'type' => 'text',
25         'options' => array(
26             'label' => 'Title',
27         ),
28         'attributes' => array(
29             'id' => 'title',
30             'maxlength' => 100,
31         )
32     ));
33
34     $this->add(array(
35         'name' => 'completed',
36         'type' => 'checkbox',
37         'options' => array(
38             'label' => 'Completed?',
39             'label_attributes' => array('class' => 'checkbox'),
40         ),
41     ));
42
43     $this->add(array(
44         'name' => 'submit',
45         'attributes' => array(
46             'type' => 'submit',
47             'value' => 'Go',
48             'class' => 'btn btn-primary',
49         ),
50     ));
51 }
52

```

Within the constructor of `TaskForm`, we set the name when we call the parent's constructor and then set the method and the input filter that we want to use. We also set the form's hydrator to be `ClassMethods`, as a form object uses hydration to transfer data to and from an entity object in exactly the same way as the `Zend\Db` components do. Finally, we create the form elements for the id, title, whether the task is complete and the 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 is this done using an input filter which can either be standalone or within any class that implements `InputFilterAwareInterface`, such as a model entity. For this application we are going to create a separate class for our input filter.

Create a new PHP file called `TaskFilter.php` in the `module/Checklist/src/Checklist/Form` folder with these contents:

module/Checklist/src/Checklist/Form/TaskFilter.php:

```

1  <?php
2  namespace Checklist\Form;
3
4  use Zend\InputFilter\InputFilter;
5
6  class TaskFilter extends InputFilter
7  {
8      public function __construct()
9      {
10         $this->add(array(
11             'name' => 'id',
12             'required' => true,
13             'filters' => array(

```

```

14         array('name' => 'Int'),
15     ),
16 );
17
18 $this->add(array(
19     'name' => 'title',
20     'required' => true,
21     'filters' => array(
22         array('name' => 'StripTags'),
23         array('name' => 'StringTrim'),
24     ),
25     'validators' => array(
26         array(
27             'name' => 'StringLength',
28             'options' => array(
29                 'encoding' => 'UTF-8',
30                 'max' => 100
31             ),
32         ),
33     ),
34 );
35
36 $this->add(array(
37     'name' => 'completed',
38     'required' => false,
39 );
40 }
41 }

```

In the constructor for the `TaskFilter`, we create inputs for each property that we want to filter. Each input can have a name, a required property a list of filters and a list of validators. All are optional other than the name property. The difference between filters and validators is that a filter changes the data passed through it and a validator tests if the data matches some specific criteria. For the title, we filter the string with `StripTags` and `StringTrim` and finally ensure that the string is no longer than 100 characters with the `StringLength` validator. For the *completed* element, we simply set required to false.

We now need to display the form and process it on submission. This is done within the `TaskController`'s `addAction()`. Open `TaskController.php` (Navigate -> Open Resource... is a convenient way to do this) and add a new method called `addAction()` to the class that looks like this:

module/Checklist/src/Checklist/Controller/TaskController.php:

```

1  public function addAction()
2  {
3      $form = new TaskForm();
4      $task = new TaskEntity();
5      $form->bind($task);
6
7      $request = $this->getRequest();
8      if ($request->isPost()) {
9          $form->setData($request->getPost());
10         if ($form->isValid()) {
11             $this->getTaskMapper()->saveTask($task);
12
13             // Redirect to list of tasks
14             return $this->redirect()->toRoute('task');
15         }
16     }

```

```

17     return array('form' => $form);
18 }
19

```

Add use `Checklist\Model\TaskEntity`; and use `Checklist\Form\TaskForm`; to the list of use statements at the top of the file.

Let's look at what the `addAction()` does in detail.

```

1  $form = new TaskForm();
2  $task = new TaskEntity();
3  $form->bind($task);

```

We instantiate a new `TaskForm` object and an empty `TaskEntity` which we bind to the form for use by the form later. The form's `bind()` method attaches the model to the form. This is used in two ways:

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

When adding a new task, we only need to worry about point 2, however for editing an item, we need data transfer in both directions.

```

1  $request = $this->getRequest();
2  if ($request->isPost()) {
3      $form->setData($request->getPost());
4      if ($form->isValid()) {

```

For a submitted form, we set the posted data to the form and check to see if it is valid using the `isValid()` member function of the form. The `isValid()` method uses the form's input filter to test for validity and if it returns true, it will then transfer the filtered data values to the entity object that is bound to the form using the registered hydrator. This means that after `isValid()` is called, `$task` now contains the submitted form data.

```

1  $this->getTaskMapper()->saveTask($task);

```

As the form is valid, we can save `$task` to the database using the mapper's `saveTask()` method.

```

1  // Redirect to list of tasks
2  return $this->redirect()->toRoute('task');

```

After we have saved the new task, we redirect back to the list of tasks using the `Redirect` controller plugin.

```

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

```

Finally, if this request is not a POST, we return the variables that we want assigned to the view. In this case, just the form object.

We also need to add the `saveTask()` method to the `TaskMapper` class. Open `module/Checklist/src/Checklist/Model/TaskMapper.php` and add this method to the end of the class:

module/Checklist/src/Checklist/Model/TaskMapper.php:

```

1  public function saveTask(TaskEntity $task)
2  {
3      $hydrator = new ClassMethods();
4      $data = $hydrator->extract($task);
5
6      if ($task->getId()) {
7          // update action

```

```

8         $action = $this->sql->update();
9         $action->set($data);
10        $action->where(array('id' => $task->getId()));
11    } else {
12        // insert action
13        $action = $this->sql->insert();
14        unset($data['id']);
15        $action->values($data);
16    }
17    $statement = $this->sql->prepareStatementForSqlObject($action);
18    $result = $statement->execute();
19
20    if (!$task->getId()) {
21        $task->setId($result->getGeneratedValue());
22    }
23    return $result;
24
25 }

```

The `saveTask()` method handles both inserting a new record if `$task` doesn't have an `id` or updating it if it does. In either case, we need the data from the entity as an array, so we can use the hydrator to do this. If we are updating, then we use the `Sql` object's `update()` method to create an `Update` object where we can set the data and a where clause. For inserting, we need an `Insert` object to which we set the values. Obviously, when inserting, the database will auto-increment the `id`, so we do not need the `id` property in the values list. In either case, we create a statement object and then execute it. Finally, if we are inserting, we populate the task entity's `id` with the value of the auto-generated `id`.

We now need to render the form in the `add.phtml` view script. Create a new PHP file called `add.phtml` in the `module/Checklist/view/checklist/task` folder and add this code:

module/Checklist/view/checklist/task/add.phtml:

```

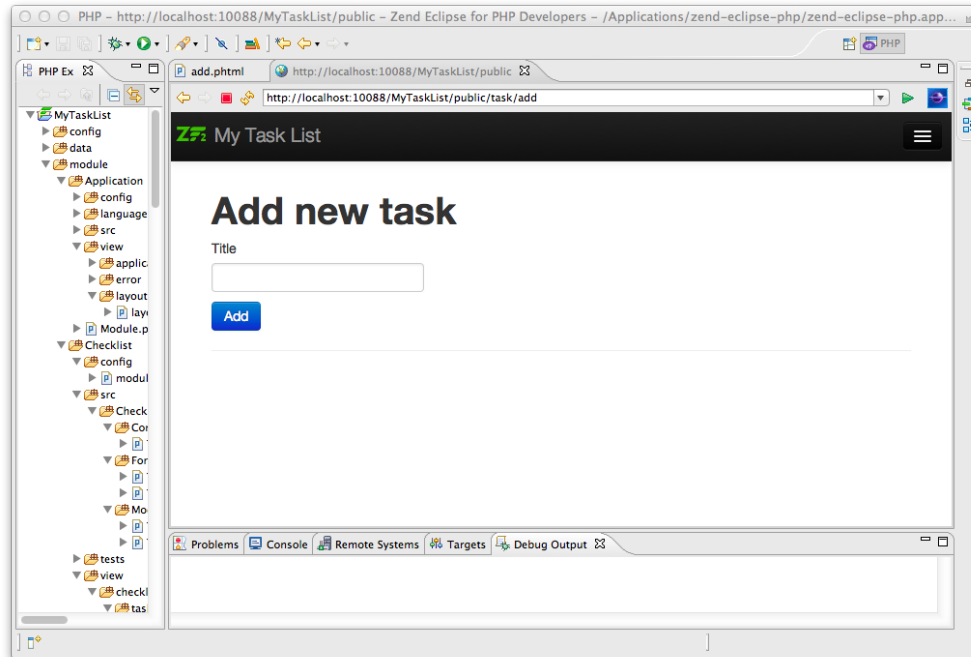
1  <?php
2  $title = 'Add new task';
3  $this->headTitle($title);
4  ?>
5  <h1><?php echo $this->escapeHtml($title); ?></h1>
6
7  <?php
8  $form = $this->form;
9  $form->setAttribute('action', $this->url('task', array('action' => 'add')));
10 $form->get('submit')->setAttribute('value', 'Add');
11 $form->prepare();
12
13 echo $this->form()->openTag($form);
14 echo $this->formHidden($form->get('id'));
15 echo $this->formRow($form->get('title'));
16 ?>
17 <div>
18 <?php echo $this->formInput($form->get('submit')); ?>
19 </div>
20 <?php
21 echo $this->form()->closeTag($form);

```

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 the title element, which has a label, we can use `formRow()` view helper which will render the HTML for the label, the element and any validator messages that may exist. For the `id` and `submit` elements,

we use `formHidden()` and `formInput()` respectively as we only need to render the element itself. We also want the submit button on its own line, so we put it within a `div`. Note that the `formRow` view helper is just a convenience - we could have used `formInput()`, `formLabel()` and `formElementErrors()` separately had we wanted to.

If you now run the application from within Zend Studio and click the “Add new item” link from the task list page, you should see:



You can now add a new task item and see it in the list of tasks.

CHAPTER 27

Editing a task

Editing a task is almost identical to adding one, so the code is very similar. This time we use `editAction()` in the `TaskController`. Open `TaskController.php` and add this method to it:

module/Checklist/src/Checklist/Controller/TaskController.php:

```
1 public function editAction()
2 {
3     $id = (int)$this->params('id');
4     if (!$id) {
5         return $this->redirect()->toRoute('task', array('action'=>'add'));
6     }
7     $task = $this->getTaskMapper()->getTask($id);
8
9     $form = new TaskForm();
10    $form->bind($task);
11
12    $request = $this->getRequest();
13    if ($request->isPost()) {
14        $form->setData($request->getPost());
15        if ($form->isValid()) {
16            $this->getTaskMapper()->saveTask($task);
17
18            return $this->redirect()->toRoute('task');
19        }
20    }
21
22    return array(
23        'id' => $id,
24        'form' => $form,
25    );
26 }
```

This code should look familiar. Let's look at the only difference from adding a task: We look for the id that is in the matched route and use it to load the task to be edited:

```
1 $id = (int)$this->params('id');
2 if (!$id) {
3     return $this->redirect()->toRoute('task', array('action'=>'add'));
4 }
5 $task = $this->getTaskMapper()->getTask($id);
```

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

As we use the form's `bind()` method with its hydrator, we do not need to populate the `$task`'s data into the form manually as it will automatically be transferred for us.

We also need to write a `getTask()` method in the `TaskMapper` to get a single record from the database, so let's do that now. Open `TaskMapper.php` and add this method:

module/Checklist/src/Checklist/Model/TaskMapper.php:

```
1 public function getTask($id)
2 {
3     $select = $this->sql->select();
4     $select->where(array('id' => $id));
5
6     $statement = $this->sql->prepareStatementForSqlObject($select);
7     $result = $statement->execute()->current();
8     if (!$result) {
9         return null;
10    }
11
12    $hydrator = new ClassMethods();
13    $task = new TaskEntity();
14    $hydrator->hydrate($result, $task);
15
16    return $task;
17 }
```

This method simply sets a where clause on the `Sql`'s `Select` object and then executes it. Calling `current()` on the result from `execute()` will return either the array of data for the row or `false`. If we retrieved data, then we use the hydrator to populate a new `TaskEntity` (`$task`) with `$data`.

In the same way as with the action methods, the view template, `edit.phtml`, looks very similar to the one for adding an task. Create a new PHP file called `edit.phtml` in in the `module/Checklist/view/checklist/task` folder and add this code:

module/Checklist/view/checklist/task/edit.phtml:

```
1 <?php
2 $title = 'Edit task';
3 $this->headTitle($title);
4 ?>
5 <h1><?php echo $this->escapeHtml($title); ?></h1>
6
7 <?php
8 $form = $this->form;
9 $url = $this->url('task', array('action' => 'edit', 'id' => $id));
10 $form->setAttribute('action', $url);
11 $form->get('submit')->setAttribute('value', 'Edit');
12 $form->prepare();
```



```
13 echo $this->form()->openTag($form);
14 echo $this->formHidden($form->get('id'));
15 echo $this->formRow($form->get('title'));
16 echo $this->formRow($form->get('completed'));
17 ?>
18 <div>
19 <?php echo $this->formInput($form->get('submit')); ?>
20 </div>
21 <?php
22 echo $this->form()->closeTag($form);
23
```

Compared to the add view script, we set the title to , 'Edit Task', and update the action URL to the edit action with the correct id. We also change the label of the button to , 'edit' and render the completed form element.

You should now be able to edit tasks.

CHAPTER 28

Deleting a task

To round out the core functionality of our application, we need to be able to delete a task. We have a *Delete* link next to each task on our list page and the naïve approach would be to run the delete action when it's clicked. This would be wrong. Remembering the HTTP specification, we recall that you shouldn't do an irreversible action using GET and should use POST instead.

We shall therefore 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 by adding the `deleteAction()` method to the `TaskController`. Open `TaskController.php` and add this method to it:

module/Checklist/src/Checklist/Controller/TaskController.php:

```
1 public function deleteAction()
2 {
3     $id = $this->params('id');
4     $task = $this->getTaskMapper()->getTask($id);
5     if (!$task) {
6         return $this->redirect()->toRoute('task');
7     }
8
9     $request = $this->getRequest();
10    if ($request->isPost()) {
11        if ($request->getPost()->get('del') == 'Yes') {
12            $this->getTaskMapper()->deleteTask($id);
13        }
14
15        return $this->redirect()->toRoute('task');
16    }
17
18    return array(
19        'id' => $id,
20        'task' => $task
21    );
22 }
```

As before, we get the id from the matched route and retrieve the task object. We then check the `Request` object's `isPost()` to determine whether to show the confirmation page or to delete the task. We use the `TaskMapper`'s `deleteTask()` method to delete the row and then redirect back to the list of tasks. If the request is not a POST, then we assign the task to the view, along with the id.

We also need to write `deleteTask()`, so open `TaskMapper.php` and add this method:

module/Checklist/src/Checklist/Model/TaskMapper.php:

```
1 public function deleteTask($id)
2 {
3     $delete = $this->sql->delete();
4     $delete->where(array('id' => $id));
5
6     $statement = $this->sql->prepareStatementForSqlObject($delete);
7     return $statement->execute();
8 }
```

This code should look fairly familiar as we again use a `Delete` object from `Zend\Db\Sql` and execute the statement from it. As we are using a `Delete` object, we set the where clause to avoid deleting every row in the table.

The view script is a simple HTML form. Create a new PHP file, `delete.phtml` in the `module/Checklist/view/checklist/task` folder with this content:

module/Checklist/view/checklist/task/delete.phtml:

```
1 <?php
2 $title = 'Delete task';
3 $this->headTitle($title);
4 ?>
5 <h1><?php echo $this->escapeHtml($title); ?></h1>
6
7 <p>Are you sure that you want to delete the
8     '<?php echo $this->escapeHtml($task->getTitle()); ?>' task?
9 </p>
10 <?php
11 $url = $this->url('task', array('action' => 'delete', 'id'=>$id)); ?>
12 <form action="<?php echo $url; ?>" method="post">
13 <div>
14     <input type="submit" name="del" value="Yes" />
15     <input type="submit" name="del" value="No" />
16 </div>
17 </form>
```

In this view script, we display a confirmation message and then a form with just Yes and No buttons. In the action, we checked specifically for the “Yes” value when doing the deletion.

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

Application Diagnostics

One really useful feature of Zend Server is the code trace feature that can show you the method-by-method execution of any given PHP request. This is especially useful in a Zend Framework 2 application as the use of Events and Service Manager means that our code base isn't necessarily linear.

Let's consider a contrived example and introduce a delay into our codebase. One of the more common causes of slow down is related to database calls taking too long due to a complicated query, incorrect indexing or by retrieving too much data. We have a very simple database table with just 5 rows, so we can simulate this by adding a `sleep()` call to our TaskMapper's `fetchAll()` method.

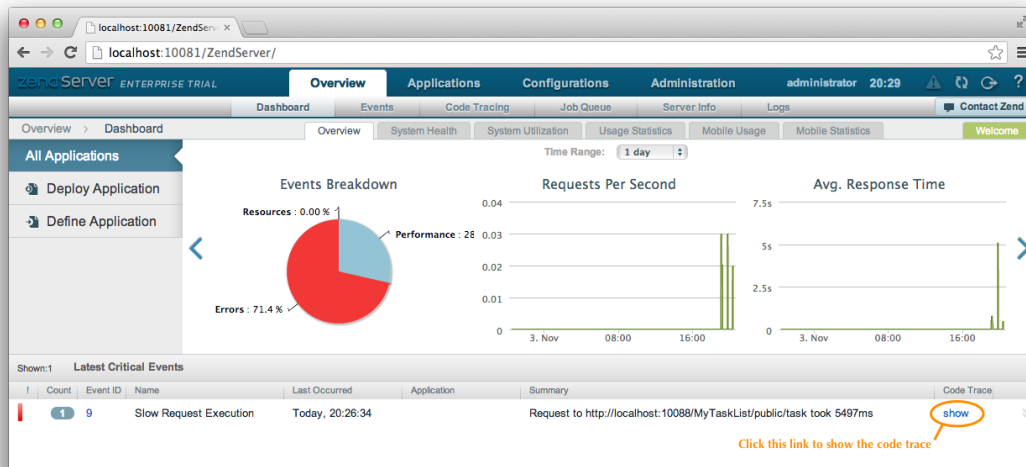
Open `Checklist/src/Checklist/Model/TaskMapper.php` and add `sleep(5);` just before the end of the `fetchAll()` method:

Checklist/src/Checklist/Model/TaskMapper.php:

```
1 public function fetchAll()
2 {
3     $select = $this->sql->select();
4     $select->order(array('completed ASC', 'created ASC'));
5
6     $statement = $this->sql->prepareStatementForSqlObject($select);
7     $results = $statement->execute();
8
9     $entityPrototype = new TaskEntity();
10    $hydrator = new ClassMethods();
11    $resultset = new HydratingResultSet($hydrator, $entityPrototype);
12    $resultset->initialize($results);
13
14    sleep(5);
15    return $resultset;
16 }
```

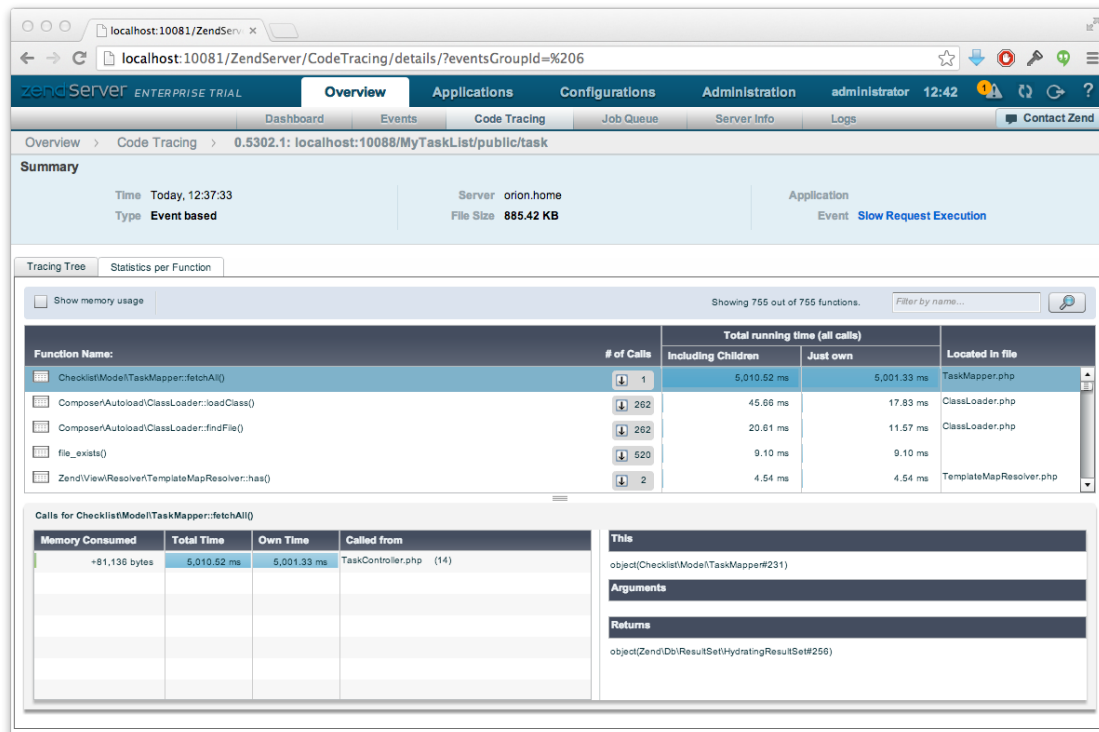
It will now take 5 seconds (and a little bit) to display the list of tasks.

If you now look at the home page of Zend Server's console, you'll see a "Slow Request Execution" critical event listed. Click on the "show" link in the "Code Trace" column as shown:



You will then see much more detail about this critical event. The easiest way to use the profile view is to click on the “Statistics per Function” tab and then order by “Just own” total running time.

This will result in the display of the slowest method at the top as shown in the screenshot.



As you can see, Zend Server has correctly determined that `fetchAll()` is the cause of the slowdown and so we can immediately go to the problem source in Zend Studio and fix the problem.

In addition to helping debugging while developing, this is obviously also extremely powerful when Zend Server is running on the production servers as this profile information is then available for those situations when a given issue only seems to happen on the live web site.

Step-by-step debugging

Another useful feature of Zend Studio and Eclipse/PDT is the step-by-step debugger. With the debugger you can set breakpoints in your code and then run the page in a browser. When the breakpoint is reached, Zend Studio pauses the page and you can then inspect variables and move forward through your code one line at a time.

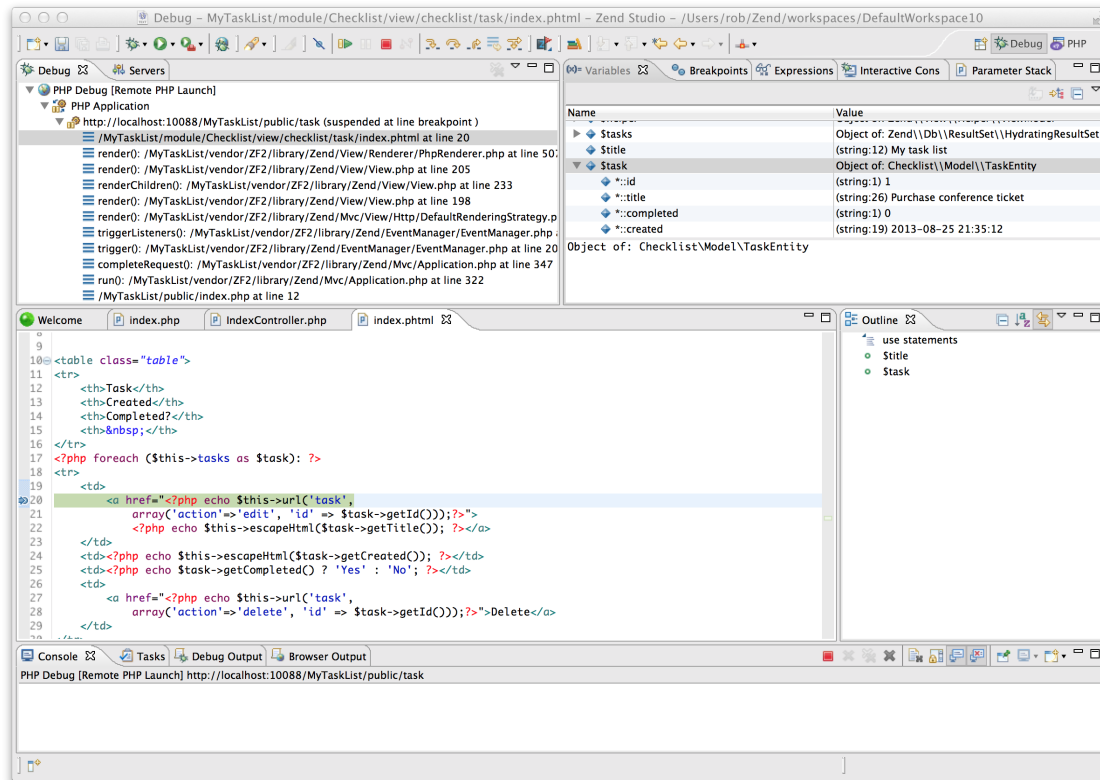
To see this in action, let's inspect the value of `$task` in the checklist module's `index.phtml` file. Open the module/Checklist/view/checklist/task/index.phtml file and double click in the gutter next to the opening `<a` tag to set a blue breakpoint marker:

```

16 </tr>
17 <?php foreach ($this->tasks as $task): ?>
18 <tr>
19     <td>
20         <a href="<?php echo $this->url('task',
21             array('action'=>'edit', 'id' => $task->getId());?>">
22             <?php echo $this->escapeHtml($task->getTitle()); ?></a>
23         </td>
24         <td><?php echo $this->escapeHtml($task->getCreated()); ?></td>
25         <td><?php echo $task->getCompleted() ? 'Yes' : 'No'; ?></td>
26         <td>
27         <a href="<?php echo $this->url('task',

```

The break point is now set. The easiest way to run to this point is to use the [Zend Studio Firefox tool bar](#). Once installed, you can navigate to <http://localhost:10088/MyTaskList/public/task> in Firefox and then press the *Debug* button in the toolbar. Zend Studio will then come to the foreground and ask you if you want to use the Debug perspective. Answer yes, as this view is designed to provide useful information while debugging. Zend Studio will pause the application on the first line of `index.php`, so press F8 to continue to the breakpoint that you set.



You will now see the code we are interested in. The centre pane shows our code with the line that the debugger is stopped on highlighted. The top left pane shows the stack trace which tells us which methods were used to get to this line of code. The top right pane shows a list of variables in scope. You can click the arrow next to `$task` to expand it and see the properties of the object. Pressing F8 will resume running the application until the next breakpoint. As our breakpoint is in a loop, it iterates once around the loop and stops again. The data in `$task` is now the second database record. Once you have finished inspecting the state of your code, you can press the square red stop button to stop the debugging mode. Clicking the PHP button in the top right hand corner of Zend Studio takes you back to the code editing view.

CHAPTER 31

Conclusion

This concludes our brief look at building a simple, but fully functional, Zend Framework 2 application using Zend Studio 10 with the code running on Zend Server 6. It barely scratches the surface of the power and flexibility of Zend Framework and we recommend reading the [manual](#) for more information. Similarly, the combination of Zend Studio and Zend Server makes for a very powerful system for writing, debugging and deploying PHP applications. The [Zend Studio manual](#) is very helpful for getting the most out of this tool.

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.

Installation using Composer

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

zf.php (Zend Tool) will be installed in the *vendor/bin* folder. You may run it with *php vendor/bin/zf.php*.

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.

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.

Usage

From Composer or Manual install:

The `zf.php` should be installed into the `vendor/ZFTool` directory (relative to your project root) - however, the command needs to be run from your project root in order for it to work correctly. You can symlink `vendor/ZFTool/zf.php` to your project root, or alternatively substitute `zf.php` for `vendor/ZFTool/zf.php` in the examples below.

Using the PHAR:

Simply substitute `zftool.phar` for `zf.php` in the below examples.

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.

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.

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).

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 supplied, defaults to autoload_classmap.php inside <directory>.

--append | -a Append to classmap file **if** it exists

--overwrite | -w Whether or not to overwrite existing classmap file

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).

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.

Learning Dependency Injection

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:

```
1 $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](#).

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.

```
1 namespace My {  
2  
3     class A  
4     {  
5         /* Some useful functionality */  
6     }  
7  
8     class B  
9     {  
10        protected $a = null;  
11        public function __construct(A $a)  
12        {  
13            $this->a = $a;  
14        }  
15    }  
16 }
```

```

15     }
16 }
    
```

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

```

1 $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:

```

1 $di = new Zend\Di\Di;
2 $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:

```

1 $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):

```

1 namespace My {
2
3     class A
4     {
5         protected $username = null;
6         protected $password = null;
7         public function __construct($username, $password)
8         {
9             $this->username = $username;
10            $this->password = $password;
11        }
12    }
13
14    class B
15    {
16        protected $a = null;
17        public function __construct(A $a)
18        {
19            $this->a = $a;
20        }
21    }
22
23    class C
24    {
25        protected $b = null;
26        public function __construct(B $b)
27        {
28            $this->b = $b;
29        }
30    }
31
32 }
    
```


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

```
1 $di = new Zend\Di\Di;
2 $di->getInstanceManager()->setProperty('A', 'username', 'MyUsernameValue');
3 $di->getInstanceManager()->setProperty('A', 'password', 'MyHardToGuessPassword%$#');
```

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:

```
1 $c = $di->get('My\C');
2 // or
3 $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:

```
1 $parameters = array(
2     'username' => 'MyUsernameValue',
3     'password' => 'MyHardToGuessPassword%$#',
4 );
5
6 $c = $di->get('My\C', $parameters);
7 // or
8 $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:

```
1 namespace My {
2     class B
3     {
4         protected $a;
5         public function setA(A $a)
6         {
7             $this->a = $a;
8         }
9     }
10 }
```

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.

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:

```
1 namespace My {
2     class B
3     {
4         protected $a;
5         public function setA($a)
6         {
7             $this->a = $a;
8         }
9     }
10 }
```

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

```
1 use Zend\Di\Di;
2 use Zend\Di\Definition;
3 use Zend\Di\Definition\Builder;
4
5 // Describe this class:
6 $builder = new Definition\BuilderDefinition;
7 $builder->addClass(($class = new Builder\PhpClass));
8
9 $class->setName('My\B');
10 $class->addInjectableMethod(($im = new Builder\InjectableMethod));
11
12 $im->setName('setA');
13 $im->addParameter('a', 'My\A');
14
15 // Use both our Builder Definition as well as the default
16 // RuntimeDefinition, builder first
17 $aDef = new Definition\AggregateDefinition;
18 $aDef->addDefinition($builder);
19 $aDef->addDefinition(new Definition\RuntimeDefinition);
20
21 // Now make sure the Di understands it
22 $di = new Di;
23 $di->setDefinition($aDef);
24
25 // and finally, create C
26 $parameters = array(
27     'username' => 'MyUsernameValue',
28     'password' => 'MyHardToGuessPassword%$#',
29 );
30
31 $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.

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`:

```
1 $compiler = new Zend\Di\Definition\Compiler();
2 $compiler->addCodeScannerDirectory(
3     new Zend\Code\Scanner\ScannerDirectory('path/to/library/My/')
4 );
5 $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):

```
1 $di = new Zend\Di\Di;
2 $di->setDefinition($definition);
3 $di->getInstanceManager()->setProperty('My\A', 'username', 'foo');
4 $di->getInstanceManager()->setProperty('My\A', 'password', 'bar');
5 $c = $di->get('My\C');
```

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

```
1 if (!file_exists(__DIR__ . '/di-definition.php') && $isProduction) {
2     $compiler = new Zend\Di\Definition\Compiler();
3     $compiler->addCodeScannerDirectory(
```

```
4         new Zend\Code\Scanner\ScannerDirectory('path/to/library/My/')
5     );
6     $definition = $compiler->compile();
7     file_put_contents(
8         __DIR__ . '/di-definition.php',
9         '<?php return ' . var_export($definition->toArray(), true) . ';'
10    );
11 } else {
12     $definition = new Zend\Di\Definition\ArrayDefinition(
13         include __DIR__ . '/di-definition.php'
14    );
15 }
16
17 // $definition can now be used; in a production system it will be written
18 // 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.

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`:

```
1 // First, compile the information
2 $compiler = new Zend\Di\Definition\CompilerDefinition();
3 $compiler->addDirectoryScanner(
4     new Zend\Code\Scanner\DirectoryScanner(__DIR__ . '/My/')
5 );
6 $compiler->compile();
7 $definition = $compiler->toArrayDefinition();
8
9 // Now, create a Definition class for this information
10 $codeGenerator = new Zend\Code\Generator\FileGenerator();
11 $codeGenerator->setClass(($class = new Zend\Code\Generator\ClassGenerator()));
12 $class->setNamespaceName('My');
13 $class->setName('DiDefinition');
14 $class->setExtendedClass('\Zend\Di\Definition\ArrayDefinition');
15 $class->addMethod(
16     '__construct',
17     array(),
18     \Zend\Code\Generator\MethodGenerator::FLAG_PUBLIC,
19     'parent::__construct(' . var_export($definition->toArray(), true) . ');'
20 );
21 file_put_contents(__DIR__ . '/My/DiDefinition.php', $codeGenerator->generate());
```

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:

```

1  use Zend\Di\Di;
2  use Zend\Di\Definition;
3  use Zend\Di\Definition\Builder;
4
5  $di = new Di;
6  $diDefAggregate = new Definition\Aggregate();
7
8  // first add in provided Definitions, for example
9  $diDefAggregate->addDefinition(new ThirdParty\Dbal\DiDefinition());
10 $diDefAggregate->addDefinition(new Zend\Controller\DiDefinition());
11
12 // for code that does not have TypeHints
13 $builder = new Definition\BuilderDefinition();
14 $builder->addClass($class = Builder\PhpClass());
15 $class->addInjectionMethod(
16     ($injectMethod = new Builder\InjectionMethod())
17 );
18 $injectMethod->setName('injectImplementation');
19 $injectMethod->addParameter(
20     'implementation', 'Class\For\Specific\Implementation'
21 );
22
23 // now, your application code
24 $compiler = new Definition\Compiler();
25 $compiler->addCodeScannerDirectory(
26     new Zend\Code\Scanner\DirectoryScanner(__DIR__ . '/App/')
27 );
28 $appDefinition = $compiler->compile();
29 $diDefAggregate->addDefinition($appDefinition);
30
31 // now, pass in properties
32 $im = $di->getInstanceManager();
33
34 // this could come from Zend\Config\Config::toArray
35 $propertiesFromConfig = array(
36     'ThirdParty\Dbal\DbAdapter' => array(
37         'username' => 'someUsername',
38         'password' => 'somePassword'
39     ),
40     'Zend\Controller\Helper\ContentType' => array(
41         'default' => 'xhtml5'
42     ),
43 );
44 $im->setProperties($propertiesFromConfig);

```

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:

```
1 use Zend\Di\ServiceLocator\Generator;
2
3 // $di is a fully configured DI instance
4 $generator = new Generator($di);
5
6 $generator->setNamespace('Application')
7     ->setContainerClass('Context');
8 $file = $generator->getCodeGenerator();
9 $file->setFilename(__DIR__ . '/../Application/Context.php');
10 $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:

```
1 <?php
2
3 namespace Application;
4
5 use Zend\Di\ServiceLocator;
6
7 class Context extends ServiceLocator
8 {
9
10     public function get($name, array $params = array())
11     {
12         switch ($name) {
13             case 'composed':
14             case 'My\ComposedClass':
15                 return $this->getMyComposedClass();
16
17             case 'struct':
18             case 'My\Struct':
19                 return $this->getMyStruct();
20
21             default:
22                 return parent::get($name, $params);
23         }
24     }
25
26     public function getComposedClass()
27     {
28         if (isset($this->services['My\ComposedClass'])) {
29             return $this->services['My\ComposedClass'];
30         }
31
32         $object = new \My\ComposedClass();
33         $this->services['My\ComposedClass'] = $object;
34         return $object;
35     }
36
37     public function getMyStruct()
38     {
39         if (isset($this->services['My\Struct'])) {
```

```
40     }
41
42     $object = new \My\Struct();
43     $this->services['My\Struct'] = $object;
44     return $object;
45 }
46
47 public function getComposed()
48 {
49     return $this->get('My\ComposedClass');
50 }
51
52 public function getStruct()
53 {
54     return $this->get('My\Struct');
55 }
56 }
```

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

```
1 $container = new Application\Context;
2
3 $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.

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.*

Setting up phpunit to use composer's autoload.php

If you used composer to generate an `autoload.php` file for you, as seen in the note on *using composer to autoload module files*, then you need to use a phpunit binary installed by composer. You can add this as a development dependency using composer itself:

```
$ php composer.phar require --dev phpunit/phpunit
```

The above command will update your `composer.json` file and perform an update for you, which will also setup autoloading rules.

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.

Bootstrapping your tests

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

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2  
3 <phpunit bootstrap="Bootstrap.php" colors="true">  
4   <testsuites>  
5     <testsuite name="zf2tutorial">  
6       <directory>./AlbumTest</directory>  
7     </testsuite>  
8   </testsuites>  
9 </phpunit>
```

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

```
1 <?php  
2  
3 namespace AlbumTest;  
4  
5 use Zend\Loader\AutoloaderFactory;  
6 use Zend\Mvc\Service\ServiceManagerConfig;  
7 use Zend\ServiceManager\ServiceManager;  
8 use RuntimeException;  
9  
10 error_reporting(E_ALL | E_STRICT);  
11 chdir(__DIR__);  
12  
13 /**  
14  * Test bootstrap, for setting up autoloading  
15  */  
16 class Bootstrap  
17 {  
18     protected static $serviceManager;  
19  
20     public static function init()  
21     {  
22         $zf2ModulePaths = array(dirname(dirname(__DIR__)));  
23         if (($path = static::findParentPath('vendor'))) {  
24             $zf2ModulePaths[] = $path;  
25         }  
26         if (($path = static::findParentPath('module')) !== $zf2ModulePaths[0]) {
```

```

27     $zf2ModulePaths[] = $path;
28 }
29
30 static::initAutoloader();
31
32 // use ModuleManager to load this module and it's dependencies
33 $config = array(
34     'module_listener_options' => array(
35         'module_paths' => $zf2ModulePaths,
36     ),
37     'modules' => array(
38         'Album'
39     )
40 );
41
42 $serviceManager = new ServiceManager(new ServiceManagerConfig());
43 $serviceManager->setService('ApplicationConfig', $config);
44 $serviceManager->get('ModuleManager')->loadModules();
45 static::$serviceManager = $serviceManager;
46 }
47
48 public static function chroot()
49 {
50     $rootPath = dirname(static::findParentPath('module'));
51     chdir($rootPath);
52 }
53
54 public static function getServiceManager()
55 {
56     return static::$serviceManager;
57 }
58
59 protected static function initAutoloader()
60 {
61     $vendorPath = static::findParentPath('vendor');
62
63     if (file_exists($vendorPath.'/autoload.php')) {
64         include $vendorPath.'/autoload.php';
65     }
66
67     if (! class_exists('Zend\Loader\AutoloaderFactory')) {
68         throw new RuntimeException(
69             'Unable to load ZF2. Run `php composer.phar install`'
70         );
71     }
72
73     AutoloaderFactory::factory(array(
74         'Zend\Loader\StandardAutoloader' => array(
75             'autoregister_zf' => true,
76             'namespaces' => array(
77                 __NAMESPACE__ => __DIR__ . '/' . __NAMESPACE__,
78             ),
79         ),
80     ));
81 }
82
83 protected static function findParentPath($path)
84 {

```

```
85     $dir = __DIR__;
86     $previousDir = '.';
87     while (!is_dir($dir . '/' . $path)) {
88         $dir = dirname($dir);
89         if ($previousDir === $dir) {
90             return false;
91         }
92         $previousDir = $dir;
93     }
94     return $dir . '/' . $path;
95 }
96 }
97
98 Bootstrap::init();
99 Bootstrap::chroot();
```

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.

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 `AlbumControllerTest.php` under `zf2-tutorial/module/Album/test/AlbumTest/Controller` with the following contents:

```
1 <?php
2
3 namespace AlbumTest\Controller;
4
5 use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;
6
7 class AlbumControllerTest extends AbstractHttpControllerTestCase
8 {
9     public function setUp()
10     {
11         $this->setApplicationConfig(
12             include '/var/www/zf2-tutorial/config/application.config.php'
13         );
14         parent::setUp();
```

```

15     }
16 }

```

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 [ZendTest](#) 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:

```

1 public function testIndexActionCanBeAccessed()
2 {
3     $this->dispatch('/album');
4     $this->assertResponseStatusCode(200);
5
6     $this->assertModuleName('Album');
7     $this->assertControllerName('Album\Controller\Album');
8     $this->assertControllerClass('AlbumController');
9     $this->assertMatchedRouteName('album');
10 }

```

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.

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:373
/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:

```
1 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.

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. The problem with this approach is that we would end up with our test cases actually doing 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 `fetchAll` retrieves 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)

```

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:

```

1  public function testAddActionRedirectsAfterValidPost()
2  {
3      $albumTableMock = $this->getMockBuilder('Album\Model\AlbumTable')
4          ->disableOriginalConstructor()
5          ->getMock();
6
7      $albumTableMock->expects($this->once())
8          ->method('saveAlbum')
9          ->will($this->returnValue(null));
10
11     $serviceManager = $this->getApplicationServiceLocator();
12     $serviceManager->setAllowOverride(true);
13     $serviceManager->setService('Album\Model\AlbumTable', $albumTableMock);
14
15     $postData = array(
16         'title' => 'Led Zeppelin III',
17         'artist' => 'Led Zeppelin',
18         'id'     => '',
19     );
20     $this->dispatch('/album/add', 'POST', $postData);
21     $this->assertResponseStatusCode(302);
22
23     $this->assertRedirectTo('/album/');
24 }

```

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`.

When testing the `editAction` you will also need to mock out the `getAlbum` method:

```
1 $albumTableMock->expects($this->once())
2     ->method('getAlbum')
3     ->will($this->returnValue(new \Album\Model\Album()));
```

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:

```
1 <?php
2 namespace AlbumTest\Model;
3
4 use Album\Model\Album;
5 use PHPUnit_Framework_TestCase;
6
7 class AlbumTest extends PHPUnit_Framework_TestCase
8 {
9     public function testAlbumInitialState()
10     {
11         $album = new Album();
12
13         $this->assertNull(
14             $album->artist,
15             '"artist" should initially be null'
16         );
17         $this->assertNull(
18             $album->id,
19             '"id" should initially be null'
20         );
21         $this->assertNull(
22             $album->title,
```



```

23         '"title" should initially be null'
24     );
25 }
26
27 public function testExchangeArraySetsPropertiesCorrectly()
28 {
29     $album = new Album();
30     $data = array('artist' => 'some artist',
31                  'id'      => 123,
32                  'title'   => 'some title');
33
34     $album->exchangeArray($data);
35
36     $this->assertSame(
37         $data['artist'],
38         $album->artist,
39         '"artist" was not set correctly'
40     );
41     $this->assertSame(
42         $data['id'],
43         $album->id,
44         '"id" was not set correctly'
45     );
46     $this->assertSame(
47         $data['title'],
48         $album->title,
49         '"title" was not set correctly'
50     );
51 }
52
53 public function testExchangeArraySetsPropertiesToNullIfKeysAreNotPresent()
54 {
55     $album = new Album();
56
57     $album->exchangeArray(array('artist' => 'some artist',
58                                'id'      => 123,
59                                'title'   => 'some title'));
60     $album->exchangeArray(array());
61
62     $this->assertNull(
63         $album->artist, '"artist" should have defaulted to null'
64     );
65     $this->assertNull(
66         $album->id, '"id" should have defaulted to null'
67     );
68     $this->assertNull(
69         $album->title, '"title" should have defaulted to null'
70     );
71 }
72
73 public function testGetArrayCopyReturnsAnArrayWithPropertyValues()
74 {
75     $album = new Album();
76     $data = array('artist' => 'some artist',
77                  'id'      => 123,
78                  'title'   => 'some title');
79
80     $album->exchangeArray($data);

```

```
81         $copyArray = $album->getArrayCopy();
82
83         $this->assertSame(
84             $data['artist'],
85             $copyArray['artist'],
86             '"artist" was not set correctly'
87         );
88         $this->assertSame(
89             $data['id'],
90             $copyArray['id'],
91             '"id" was not set correctly'
92         );
93         $this->assertSame(
94             $data['title'],
95             $copyArray['title'],
96             '"title" was not set correctly'
97         );
98     }
99
100     public function testInputFiltersAreSetCorrectly()
101     {
102         $album = new Album();
103
104         $inputFilter = $album->getInputFilter();
105
106         $this->assertSame(3, $inputFilter->count());
107         $this->assertTrue($inputFilter->has('artist'));
108         $this->assertTrue($inputFilter->has('id'));
109         $this->assertTrue($inputFilter->has('title'));
110     }
111 }
```

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)
```

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:

```

1 <?php
2 namespace AlbumTest\Model;
3
4 use Album\Model\AlbumTable;
5 use Album\Model\Album;
6 use Zend\Db\ResultSet\ResultSet;
7 use PHPUnit\Framework\TestCase;
8
9 class AlbumTableTest extends PHPUnit\Framework\TestCase
10 {
11     public function testFetchAllReturnsAllAlbums()
12     {
13         $resultSet = new ResultSet();
14         $mockTableGateway = $this->getMock(
15             'Zend\Db\TableGateway\TableGateway',
16             array('select'),
17             array(),
18             '',
19             false
20         );
21         $mockTableGateway->expects($this->once())
22             ->method('select')
23             ->with()
24             ->will($this->returnValue($resultSet));
25
26         $albumTable = new AlbumTable($mockTableGateway);
27
28         $this->assertSame($resultSet, $albumTable->fetchAll());
29     }
30 }
```

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:

```

1 public function testCanRetrieveAnAlbumByItsId()
2 {
3     $album = new Album();
4     $album->exchangeArray(array('id' => 123,
5                                'artist' => 'The Military Wives',
6                                'title' => 'In My Dreams'));
7
8     $resultSet = new ResultSet();
```

```

9     $resultSet->setArrayObjectPrototype(new Album());
10    $resultSet->initialize(array($album));
11
12    $mockTableGateway = $this->getMock(
13        'Zend\Db\TableGateway\TableGateway',
14        array('select'),
15        array(),
16        '',
17        false
18    );
19    $mockTableGateway->expects($this->once())
20        ->method('select')
21        ->with(array('id' => 123))
22        ->will($this->returnValue($resultSet));
23
24    $albumTable = new AlbumTable($mockTableGateway);
25
26    $this->assertSame($album, $albumTable->getAlbum(123));
27 }
28
29 public function testCanDeleteAnAlbumByItsId()
30 {
31     $mockTableGateway = $this->getMock(
32         'Zend\Db\TableGateway\TableGateway',
33         array('delete'),
34         array(),
35         '',
36         false
37     );
38     $mockTableGateway->expects($this->once())
39         ->method('delete')
40         ->with(array('id' => 123));
41
42     $albumTable = new AlbumTable($mockTableGateway);
43     $albumTable->deleteAlbum(123);
44 }
45
46 public function testSaveAlbumWillInsertNewAlbumsIfTheyDontAlreadyHaveAnId()
47 {
48     $albumData = array(
49         'artist' => 'The Military Wives',
50         'title'  => 'In My Dreams'
51     );
52     $album      = new Album();
53     $album->exchangeArray($albumData);
54
55     $mockTableGateway = $this->getMock(
56         'Zend\Db\TableGateway\TableGateway',
57         array('insert'),
58         array(),
59         '',
60         false
61     );
62     $mockTableGateway->expects($this->once())
63         ->method('insert')
64         ->with($albumData);
65
66     $albumTable = new AlbumTable($mockTableGateway);

```

```

67     $albumTable->saveAlbum($album);
68 }
69
70 public function testSaveAlbumWillUpdateExistingAlbumsIfTheyAlreadyHaveAnId()
71 {
72     $albumData = array(
73         'id'      => 123,
74         'artist' => 'The Military Wives',
75         'title'  => 'In My Dreams',
76     );
77     $album      = new Album();
78     $album->exchangeArray($albumData);
79
80     $resultSet = new ResultSet();
81     $resultSet->setArrayObjectPrototype(new Album());
82     $resultSet->initialize(array($album));
83
84     $mockTableGateway = $this->getMock(
85         'Zend\Db\TableGateway\TableGateway',
86         array('select', 'update'),
87         array(),
88         '',
89         false
90     );
91     $mockTableGateway->expects($this->once())
92         ->method('select')
93         ->with(array('id' => 123))
94         ->will($this->returnValue($resultSet));
95     $mockTableGateway->expects($this->once())
96         ->method('update')
97         ->with(
98             array(
99                 'artist' => 'The Military Wives',
100                 'title'  => 'In My Dreams'
101             ),
102             array('id' => 123)
103         );
104
105     $albumTable = new AlbumTable($mockTableGateway);
106     $albumTable->saveAlbum($album);
107 }
108
109 public function testExceptionIsThrownWhenGettingNonExistentAlbum()
110 {
111     $resultSet = new ResultSet();
112     $resultSet->setArrayObjectPrototype(new Album());
113     $resultSet->initialize(array());
114
115     $mockTableGateway = $this->getMock(
116         'Zend\Db\TableGateway\TableGateway',
117         array('select'),
118         array(),
119         '',
120         false
121     );
122     $mockTableGateway->expects($this->once())
123         ->method('select')
124         ->with(array('id' => 123))

```

```
125         ->will($this->returnValue($resultSet));
126
127     $albumTable = new AlbumTable($mockTableGateway);
128
129     try {
130         $albumTable->getAlbum(123);
131     }
132     catch (\Exception $e) {
133         $this->assertSame('Could not find row 123', $e->getMessage());
134         return;
135     }
136
137     $this->fail('Expected exception was not thrown');
138 }
```

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)
```

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.

Using the EventManager

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

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.

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:

```
1 use Zend\EventManager\EventManager;
2
3 $events = new EventManager();
4 $events->attach('do', function ($e) {
5     $event = $e->getName();
6     $params = $e->getParams();
7     printf(
```

```
8         'Handled event "%s", with parameters %s',
9         $event,
10        json_encode($params)
11    );
12 });
13
14 $params = array('foo' => 'bar', 'baz' => 'bat');
15 $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.

```
1 use Zend\EventManager\EventManager;
2 use Zend\EventManager\EventManagerAwareInterface;
3 use Zend\EventManager\EventManagerInterface;
4
5 class Example implements EventManagerAwareInterface
6 {
7     protected $events;
8
9     public function setEventManager(EventManagerInterface $events)
10    {
11        $events->setIdentifiers(array(
12            __CLASS__,
13            get_class($this)
14        ));
15        $this->events = $events;
16    }
17
18     public function getEventManager()
19    {
20        if (!$this->events) {
21            $this->setEventManager(new EventManager());
22        }
23        return $this->events;
24    }
25
26     public function doIt($foo, $baz)
27    {
28        $params = compact('foo', 'baz');
29        $this->getEventManager()->trigger(__FUNCTION__, $this, $params);
30    }
31 }
32
33
```



```

34 $example = new Example();
35
36 $example->getEventManager()->attach('doIt', function($e) {
37     $event = $e->getName();
38     $target = get_class($e->getTarget()); // "Example"
39     $params = $e->getParams();
40     printf(
41         'Handled event "%s" on target "%s", with parameters %s',
42         $event,
43         $target,
44         json_encode($params)
45     );
46 });
47
48 $example->doIt('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?

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:

```

1  use Zend\EventManager\SharedEventManager;
2
3  $sharedEvents = new SharedEventManager();
4  $sharedEvents->attach('Example', 'do', function ($e) {
5      $event = $e->getName();
6      $target = get_class($e->getTarget()); // "Example"
7      $params = $e->getParams();
8      printf(
9          'Handled event "%s" on target "%s", with parameters %s',
10         $event,
11         $target,
12         json_encode($params)
13     );
14 });

```

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:

```
1 $example = new Example();
2 $example->getEventManager()->setSharedManager($sharedEvents);
3 $example->do('bar', 'bat');
```

and expect the following to be echoed:

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

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

```
1 class SubExample extends Example
2 {
3 }
```

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()`:

```
$events->setSharedManager(null);
```

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

```
$events->setSharedManager($sharedEvents);
```

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:

```
1 // Multiple named events:
2 $events->attach(
3     array('foo', 'bar', 'baz'), // events
4     $listener
5 );
6
7 // All events via wildcard:
8 $events->attach(
9     '*', // all events
10    $listener
11 );
12
13 // Multiple named targets:
14 $sharedEvents->attach(
15     array('Foo', 'Bar', 'Baz'), // targets
16     'doSomething', // named event
17     $listener
18 );
19
20 // All targets via wildcard
21 $sharedEvents->attach(
22     '*', // all targets
23     'doSomething', // named event
24     $listener
25 );
26
27 // Mix and match: multiple named events on multiple named targets:
```

```
28 $sharedEvents->attach(  
29     array('Foo', 'Bar', 'Baz'), // targets  
30     array('foo', 'bar', 'baz'), // events  
31     $listener  
32 );  
33  
34 // Mix and match: all events on multiple named targets:  
35 $sharedEvents->attach(  
36     array('Foo', 'Bar', 'Baz'), // targets  
37     '*', // events  
38     $listener  
39 );  
40  
41 // Mix and match: multiple named events on all targets:  
42 $sharedEvents->attach(  
43     '*', // targets  
44     array('foo', 'bar', 'baz'), // events  
45     $listener  
46 );  
47  
48 // Mix and match: all events on all targets:  
49 $sharedEvents->attach(  
50     '*', // targets  
51     '*', // events  
52     $listener  
53 );
```

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

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:

```
1 use Zend\EventManager\EventInterface;
2 use Zend\EventManager\EventManagerInterface;
3 use Zend\EventManager\ListenerAggregateInterface;
4 use Zend\Log\Logger;
5
6 class LogEvents implements ListenerAggregateInterface
7 {
8     protected $listeners = array();
9     protected $log;
10
11     public function __construct(Logger $log)
12     {
13         $this->log = $log;
14     }
15
16     public function attach(EventManagerInterface $events)
17     {
18         $this->listeners[] = $events->attach('do', array($this, 'log'));
19         $this->listeners[] = $events->attach('doSomethingElse', array($this, 'log'));
20     }
21
22     public function detach(EventCollection $events)
23     {
24         foreach ($this->listeners as $index => $listener) {
25             if ($events->detach($listener)) {
```

```
26         unset($this->listeners[$index]);
27     }
28 }
29 }
30
31 public function log(EventInterface $e)
32 {
33     $event = $e->getName();
34     $params = $e->getParams();
35     $this->log->info(sprintf('%s: %s', $event, json_encode($params)));
36 }
37 }
```

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

```
$logListener = new LogEvents($logger);

$events->attachAggregate($logListener); // OR
$events->attach($logListener);
```

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 somewhere. Aggregates typically do this for you – as you can see in the example above.

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.

Short-circuiting listener execution

You may want to short-circuit 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:

```

1 public function someExpensiveCall($criterial, $criteria2)
2 {
3     $params = compact('criterial', 'criteria2');
4     $results = $this->getEventManager()->trigger(
5         __FUNCTION__,
6         $this,
7         $params,
8         function ($r) {
9             return ($r instanceof SomeResultClass);
10        }
11    );
12    if ($results->stopped()) {
13        return $results->last();
14    }
15
16    // ... do some work ...
17 }

```

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.

```

1 $events->attach('do', function ($e) {
2     $e->stopPropagation();
3     return new SomeResultClass();
4 });

```

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.

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 `SharedEventManager::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:

```

1 $priority = 100;
2 $events->attach('Example', 'do', function ($e) {

```

```

3     $event = $e->getName();
4     $target = get_class($e->getTarget()); // "Example"
5     $params = $e->getParams();
6     printf(
7         'Handled event "%s" on target "%s", with parameters %s',
8         $event,
9         $target,
10        json_encode($params)
11    );
12 }, $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.

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:

```

1     $routeMatch = $e->getParam('route-match', false);
2     if (!$routeMatch) {
3         // Oh noes! we cannot do our work! whatever shall we do?!?!?!
4     }

```

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:

```

1     // in the method:
2     $params['__RESULT'] = $computedResult;
3     $events->trigger(__FUNCTION__ . '.post', $this, $params);
4
5     // in the listener:
6     $result = $e->getParam('__RESULT__');
7     if (!$result) {
8         // Oh noes! we cannot do our work! whatever shall we do?!?!?!
9     }

```

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:

```

1     $response = $e->getResponse();
2     $result = $e->getResult();

```



```

3 if (is_string($result)) {
4     $content = $view->render('layout.phtml', array('content' => $result));
5     $response->setContent($content);
6 }

```

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.

```

1 $event = new CustomEvent();
2 $event->setSomeKey($value);
3
4 // Injected with event name and target:
5 $events->trigger('foo', $this, $event);
6
7 // Injected with event name:
8 $event->setTarget($this);
9 $events->trigger('foo', $event);
10
11 // Fully encapsulates all necessary properties:
12 $event->setName('foo');
13 $event->setTarget($this);
14 $events->trigger($event);
15
16 // Passing a callback following the event object works for
17 // short-circuiting, too.
18 $results = $events->trigger('foo', $this, $event, $callback);

```

This is a really powerful technique for domain-specific event systems, and definitely worth experimenting with.

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 system.

Here's what the method will look like:

```

1 public function someExpensiveCall($criterial, $criteria2)
2 {
3     $params = compact('criterial', 'criteria2');
4     $results = $this->getEventManager()->trigger(
5         __FUNCTION__ . '.pre',
6         $this,
7         $params,
8         function ($r) {

```

```
9         return ($r instanceof SomeResultClass);
10     }
11 );
12 if ($results->stopped()) {
13     return $results->last();
14 }
15
16 // ... do some work ...
17
18 $params['__RESULT__'] = $calculatedResult;
19 $this->events()->trigger(__FUNCTION__ . '.post', $this, $params);
20 return $calculatedResult;
21 }
```

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.

```
1 $events->attach('someExpensiveCall.pre', function($e) use ($cache) {
2     $params = $e->getParams();
3     $key     = md5(json_encode($params));
4     $hit     = $cache->load($key);
5     return $hit;
6 }, 100);
7
8 $events->attach('someExpensiveCall.post', function($e) use ($cache) {
9     $params = $e->getParams();
10    $result  = $params['__RESULT__'];
11    unset($params['__RESULT__']);
12    $key     = md5(json_encode($params));
13    $cache->save($result, $key);
14 }, -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:

```
1 public function setEventManager(EventManagerInterface $events)
2 {
3     $this->events = $events;
4     $events->setIdentifiers(array(__CLASS__, get_class($this)));
5     $events->attach('someExpensiveCall', array($this, 'doSomeExpensiveCall'));
6 }
7
8 public function someExpensiveCall($criterial, $criteria2)
9 {
10     $params = compact('criterial', 'criteria2');
11     $results = $this->getEventManager()->trigger(
12         __FUNCTION__,
```

```
13         $this,  
14         $params,  
15         function ($r) {  
16             return ($r instanceof SomeResultClass);  
17         }  
18     );  
19     return $results->last();  
20 }  
21  
22 public function doSomeExpensiveCall($e)  
23 {  
24     // ... do some work ...  
25     $e->setParam('__RESULT__', $calculatedResult);  
26     return $calculatedResult;  
27 }
```

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.

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.

Advanced Configuration Tricks

Configuration of Zend Framework 2 applications happens in several steps:

- Initial configuration is passed to the `Application` instance and used to seed the `ModuleManager` and `ServiceManager`. In this tutorial, we will call this configuration **system configuration**.
- The `ModuleManager`'s `ConfigListener` aggregates configuration and merges it while modules are being loaded. In this tutorial, we will call this configuration **application configuration**.
- Once configuration is aggregated from all modules, the `ConfigListener` will also merge application configuration globbed in specified directories (typically `config/autoload/`).
- Finally, immediately prior to the merged application configuration being passed to the `ServiceManager`, it is passed to a special `EVENT_MERGE_CONFIG` event to allow further modification.

In this tutorial, we'll look at the exact sequence, and how you can tie into it.

System configuration

To begin module loading, we have to tell the `Application` instance about the available modules and where they live, optionally provide some information to the default module listeners (e.g., where application configuration lives, and what files to load; whether to cache merged configuration, and where; etc.), and optionally seed the `ServiceManager`. For purposes of this tutorial we will call this the **system configuration**.

When using the skeleton application, the **system configuration** is by default in `config/application.config.php`. The defaults look like this:

```
1 <?php
2 return array(
3     // This should be an array of module namespaces used in the application.
4     'modules' => array(
5         'Application',
6     ),
7
8     // These are various options for the listeners attached to the ModuleManager
```

```
9      'module_listener_options' => array(  
10          // This should be an array of paths in which modules reside.  
11          // If a string key is provided, the listener will consider that a module  
12          // namespace, the value of that key the specific path to that module's  
13          // Module class.  
14          'module_paths' => array(  
15              './module',  
16              './vendor',  
17          ),  
18  
19          // An array of paths from which to glob configuration files after  
20          // modules are loaded. These effectively override configuration  
21          // provided by modules themselves. Paths may use GLOB_BRACE notation.  
22          'config_glob_paths' => array(  
23              'config/autoload/{,*.}global,{,*.}local}.php',  
24          ),  
25  
26          // Whether or not to enable a configuration cache.  
27          // If enabled, the merged configuration will be cached and used in  
28          // subsequent requests.  
29          'config_cache_enabled' => $booleanValue,  
30  
31          // The key used to create the configuration cache file name.  
32          'config_cache_key' => $stringKey,  
33  
34          // Whether or not to enable a module class map cache.  
35          // If enabled, creates a module class map cache which will be used  
36          // by in future requests, to reduce the autoloading process.  
37          'module_map_cache_enabled' => $booleanValue,  
38  
39          // The key used to create the class map cache file name.  
40          'module_map_cache_key' => $stringKey,  
41  
42          // The path in which to cache merged configuration.  
43          'cache_dir' => $stringPath,  
44  
45          // Whether or not to enable modules dependency checking.  
46          // Enabled by default, prevents usage of modules that depend on other modules  
47          // that weren't loaded.  
48          'check_dependencies' => true,  
49      ),  
50  
51      // Used to create an own service manager. May contain one or more child arrays.  
52      'service_listener_options' => array(  
53          // array(  
54          //     'service_manager' => $stringServiceManagerName,  
55          //     'config_key'      => $stringConfigKey,  
56          //     'interface'       => $stringOptionalInterface,  
57          //     'method'          => $stringRequiredMethodName,  
58          // ),  
59      )  
60  
61      // Initial configuration with which to seed the ServiceManager.  
62      // Should be compatible with Zend\ServiceManager\Config.  
63      // 'service_manager' => array(),  
64  );
```

The system configuration is for the bits and pieces related to the MVC that run before your application is ready. The

configuration is usually brief, and quite minimal.

Also, system configuration is used *immediately*, and is not merged with any other configuration – which means, with the exception of the values under the ‘service_manager’ key, it cannot be overridden by a module.

This leads us to our first trick: how do you provide environment-specific system configuration?

Environment-specific system configuration

What happens when you want to change the set of modules you use based on the environment? Or if the configuration caching should be enabled based on environment?

It is for this reason that the default system configuration we provide in the skeleton application is in PHP; providing it in PHP means you can programmatically manipulate it.

As an example, let’s make the following requirements:

- We want to use the `ZendDeveloperTools` module in development only.
- We want to have configuration caching on in production only.

To make this happen, we’ll set an environment variable in our web server configuration, `APP_ENV`. In Apache, you’d put a directive like the following in either your system-wide `apache.conf` or `httpd.conf`, or in the definition for your virtual host; alternately, it can be placed in an `.htaccess` file.

```
SetEnv "APP_ENV" "development"
```

For other web servers, consult the web server documentation to determine how to set environment variables.

To simplify matters, we’ll assume the environment is “production” if no environment variable is present.

We’ll modify the `config/application.config.php` file to read as follows:

```
1 <?php
2 $env = getenv('APP_ENV') ?: 'production';
3
4 // Use the $env value to determine which modules to load
5 $modules = array(
6     'Application',
7 );
8 if ($env == 'development') {
9     $modules[] = 'ZendDeveloperTools';
10 }
11
12 return array(
13     'modules' => $modules,
14
15     'module_listener_options' => array(
16         'module_paths' => array(
17             './module',
18             './vendor',
19         ),
20
21         'config_glob_paths' => array(
22             'config/autoload/{,*.}global,{,*.}local.php',
23         ),
24
25         // Use the $env value to determine the state of the flag
26         'config_cache_enabled' => ($env == 'production'),
27
```

```
28     'config_cache_key' => 'app_config',
29
30     // Use the $env value to determine the state of the flag
31     'module_map_cache_enabled' => ($env == 'production'),
32
33     'module_map_cache_key' => 'module_map',
34
35     'cache_dir' => 'data/config/',
36
37     // Use the $env value to determine the state of the flag
38     'check_dependencies' => ($env != 'production'),
39 ),
40 );
```

This approach gives you flexibility to alter system-level settings.

However, how about altering *application specific* settings (not system configuration) based on the environment?

Environment-specific application configuration

Sometimes you want to change application configuration to load things such as database adapters, log writers, cache adapters, and more based on the environment. These are typically managed in the service manager, and may be defined by modules. You can override them at the application level via `Zend\ModuleManager\Listener\ConfigListener`, by specifying a glob path in the **system configuration** – the `module_listener_options.config_glob_paths` key from the previous examples.

The default value for this is `config/autoload/{,*.}global,{,*.}local}.php`. What this means is that it will look for **application configuration** files in the `config/autoload` directory, in the following order:

- `global.php`
- `*.global.php`
- `local.php`
- `*.local.php`

This allows you to define application-level defaults in “global” configuration files, which you would then commit to your version control system, and environment-specific overrides in your “local” configuration files, which you would *omit* from version control.

This is a great solution for development, as it allows you to specify alternate configuration that’s specific to your development environment without worrying about accidentally deploying it. However, what if you have more environments – such as a “testing” or “staging” environment – and they each have their own specific overrides?

Again, the application environment variable comes to play. We can alter the glob path in the system configuration slightly:

```
'config_glob_paths' => array(
    sprintf('config/autoload/{,*.}{global,%s,local}.php', $env)
),
```

The above will allow you to define an additional set of application configuration files per environment; furthermore, these will be loaded *only* if that environment is detected!

As an example, consider the following tree of configuration files:

```
config/
  autoload/
```



```
global.php
local.php
users.development.php
users.testing.php
users.local.php
```

If `$env` evaluates to `testing`, then the following files will be merged, in the following order:

```
global.php
users.testing.php
local.php
users.local.php
```

Note that `users.development.php` is not loaded – this is because it will not match the glob pattern!

Also, because of the order in which they are loaded, you can predict which values will overwrite the others, allowing you to both selectively overwrite as well as debug later.

Note: The files under `config/autoload/` are merged *after* your module configuration, detailed in next section. We have detailed it here, however, as setting up the **application configuration** glob path happens within the **system configuration** (`config/application.config.php`).

Module Configuration

One responsibility of modules is to provide their own configuration to the application. Modules have two general mechanisms for doing this.

First, modules that either implement `Zend\ModuleManager\Feature\ConfigProviderInterface` and/or a `getConfig()` method can return their configuration. The default, recommended implementation of the `getConfig()` method is:

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

where `module.config.php` returns a PHP array. From that PHP array you can provide general configuration as well as configuration for all the available Manager classes provided by the ServiceManager. Please refer to the [Configuration mapping table](#) to see which configuration key is used for each specific Manager.

Second, modules can implement a number of interfaces and/or methods related to specific service manager or plugin manager configuration. You will find an overview of all interfaces and their matching Module Configuration functions inside the [Configuration mapping table](#).

All interfaces are in the `Zend\ModuleManager\Feature` namespace, and each is expected to return an array of configuration for a service manager, as denoted in the [section on default service configuration](#).

Configuration mapping table

Manager name	Interface name	Module Method name	Config key name
ControllerPluginManager	ControllerPluginProviderInterface	getControllerPluginConfig()	controller_plugins
ControllerManager	ControllerProviderInterface	getControllerConfig()	controllers
FilterManager	FilterProviderInterface	getFilterConfig()	filters
FormElementManager	FormElementProviderInterface	getFormElementConfig()	form_elements
HydratorManager	HydratorProviderInterface	getHydratorConfig()	hydrators
InputFilterManager	InputFilterProviderInterface	getInputFilterConfig()	input_filters
RoutePluginManager	RouteProviderInterface	getRouteConfig()	route_manager
SerializerAdapterManager	SerializerProviderInterface	getSerializerConfig()	serializers
ServiceLocator	ServiceProviderInterface	getServiceConfig()	service_manager
ValidatorManager	ValidatorProviderInterface	getValidatorConfig()	validators
ViewHelperManager	ViewHelperProviderInterface	getViewHelperConfig()	view_helpers
LogProcessorManager	LogProcessorProviderInterface	getLogProcessorConfig()	log_processors
LogWriterManager	LogWriterProviderInterface	getLogWriterConfig()	log_writers

Configuration Priority

Considering that you may have service configuration in your module configuration file, what has precedence?

The order in which they are merged is:

- configuration returned by the various service configuration methods in a module class
- configuration returned by `getConfig()`

In other words, your `getConfig()` win over the various service configuration methods. Additionally, and of particular note: the configuration returned from those methods will *not* be cached.

Note: Use the various service configuration methods when you need to define closures or instance callbacks for factories, abstract factories, and initializers. This prevents caching problems, and also allows you to write your configuration files in other markup formats.

Manipulating merged configuration

Occasionally you will want to not just override an application configuration key, but actually remove it. Since merging will not remove keys, how can you handle this?

`Zend\ModuleManager\Listener\ConfigListener` triggers a special event, `Zend\ModuleManager\ModuleEvent::EVENT_MERGE_CONFIG`, after merging all configuration, but prior to it being passed to the `ServiceManager`. By listening to this event, you can inspect the merged configuration and manipulate it.

The `ConfigListener` itself listens to the event at priority 1000 (i.e., very high), which is when the configuration is merged. You can tie into this to modify the merged configuration from your module, via the `init()` method.

```
1 namespace Foo;
2
3 use Zend\ModuleManager\ModuleEvent;
```

```

4 use Zend\ModuleManager\ModuleManager;
5
6 class Module
7 {
8     public function init(ModuleManager $moduleManager)
9     {
10         $sevents = $moduleManager->getEventManager();
11
12         // Registering a listener at default priority, 1, which will trigger
13         // after the ConfigListener merges config.
14         $sevents->attach(ModuleEvent::EVENT_MERGE_CONFIG, array($this, 'onMergeConfig
15         ↪'));
16     }
17
18     public function onMergeConfig(ModuleEvent $e)
19     {
20         $configListener = $e->getConfigListener();
21         $config          = $configListener->getMergedConfig(false);
22
23         // Modify the configuration; here, we'll remove a specific key:
24         if (isset($config['some_key'])) {
25             unset($config['some_key']);
26         }
27
28         // Pass the changed configuration back to the listener:
29         $configListener->setMergedConfig($config);
30     }
31 }

```

At this point, the merged application configuration will no longer contain the key `some_key`.

Note: If a cached config is used by the `ModuleManager`, the `EVENT_MERGE_CONFIG` event will not be triggered. However, typically that means that what is cached will be what was originally manipulated by your listener.

Configuration merging workflow

To cap off the tutorial, let's review how and when configuration is defined and merged.

- **System configuration**

- Defined in `config/application.config.php`
- No merging occurs
- Allows manipulation programmatically, which allows the ability to:
 - * Alter flags based on computed values
 - * Alter the configuration glob path based on computed values
- Configuration is passed to the `Application` instance, and then the `ModuleManager` in order to initialize the system.

- **Application configuration**

- The `ModuleManager` loops through each module class in the order defined in the **system configuration**

- * Service configuration defined in `Module` class methods is aggregated
- * Configuration returned by `Module::getConfig()` is aggregated
- Files detected from the **service configuration** `config_glob_paths` setting are merged, based on the order they resolve in the glob path.
- `ConfigListener` triggers `EVENT_MERGE_CONFIG`: - `ConfigListener` merges configuration - Any other event listeners manipulate the configuration
- Merged configuration is finally passed to the `ServiceManager`

Using Zend\Navigation in your Album Module

In this tutorial we will use the *Zend\Navigation component* to add a navigation menu to the black bar at the top of the screen, and add breadcrumbs above the main site content.

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, let's 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:

```
1 'home' => array(  
2     'type' => 'Zend\Mvc\Router\Http\Literal',  
3     'options' => array(  
4         'route' => '/',  
5         'defaults' => array(  
6             'controller' => 'Application\Controller\Index', // <-- change back here  
7             'action' => 'index',  
8         ),  
9     ),  
10 ),
```

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.

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:

```
1 'service_manager' => array(  
2     'factories' => array(  
3         'navigation' => 'Zend\Navigation\Service\DefaultNavigationFactory', // <--  
4         ↪add this  
5     ),  
6 ),
```

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 automatically 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:

```
1 return array(  
2     ...  
3     'navigation' => array(  
4         'default' => array(  
5             array(  
6                 'label' => 'Home',  
7                 'route' => 'home',  
8             ),  
9             array(  
10                'label' => 'Album',  
11                'route' => 'album',  
12                'pages' => array(  
13                    array(  
14                        'label' => 'Add',  
15                        'route' => 'album',  
16                        'action' => 'add',  
17                    ),  
18                    array(  
19                        'label' => 'Edit',  
20                        'route' => 'album',  
21                        'action' => 'edit',  
22                    ),  
23                    array(  
24                        'label' => 'Delete',  
25                        'route' => 'album',  
26                        'action' => 'delete',  
27                    ),  
28                ),  
29            ),  
30        ),  
31    ),
```

```
32  ...
33  );
```

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](#).

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:

```
1  ...
2  <div class="collapse navbar-collapse">
3      <?php // <-- Add this !!
4      echo $this->navigation('navigation')->menu();
5      ?>
6  </div>
7  ...
```

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 menu, with just a few tweaks however, we can make it look awesome:

module/Application/view/layout/layout.phtml:

```
1  <div class="collapse navbar-collapse">
2      <?php // <-- Update this !!
3      echo $this->navigation('navigation')
4          ->menu()
5          ->setMinDepth(0)
6          ->setMaxDepth(0)
7          ->setUlClass('nav navbar-nav');
8      ?>
9  </div>
```

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.

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 by using the *breadcrumbs view helper*:

module/Application/view/layout/layout.phtml:

```

1  ...
2  <div class="container">
3      <?php echo $this->navigation('navigation')->breadcrumbs()->setMinDepth(0); // <--
    ↪Add this!! ?>
4      <?php echo $this->content; ?>
5  </div>
6  ...
    
```

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 its 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:

```

1  <ul class="breadcrumb">
2      <?php
3      // iterate through the pages
4      foreach ($this->pages as $key => $page) :
5          ?>
6          <li>
7              <?php
8              // if this isn't the last page, add a link and the separator
9              if ($key < count($this->pages) - 1) :
10                 ?>
11                 <a href="<?php echo $page->getHref(); ?>"><?php echo $page->
    ↪getLabel(); ?></a>
12                 <?php
13                 // otherwise, just output the name
14                 else:
15                     ?>
16                     <?php echo $page->getLabel(); ?>
17                     <?php endif; ?>
18             </li>
19         <?php endforeach; ?>
20 </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:

```

1  ...
2  <div class="container">
3      <?php
4      echo $this->navigation('navigation') // <-- Update this!!
5          ->breadcrumbs()
6          ->setMinDepth(0)
7          ->setPartial('partial/breadcrumb.phtml');
8      ?>
9      <?php echo $this->content; ?>
10 </div>
11 ...
    
```

Refreshing the page now gives us a lovely styled set of breadcrumbs on each page.

Using Zend\Paginator in your Album Module

In this tutorial we will use the *Zend\Paginator component* 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:



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!):

```

1 INSERT INTO `album` (`artist`, `title`)
2 VALUES
3     ('David Bowie', 'The Next Day (Deluxe Version)'),
4     ('Bastille', 'Bad Blood'),
5     ('Bruno Mars', 'Unorthodox Jukebox'),
6     ('Emeli Sandé', 'Our Version of Events (Special Edition)'),
7     ('Bon Jovi', 'What About Now (Deluxe Version)'),
8     ('Justin Timberlake', 'The 20/20 Experience (Deluxe Version)'),
9     ('Bastille', 'Bad Blood (The Extended Cut)'),
10    ('P!nk', 'The Truth About Love'),
11    ('Sound City - Real to Reel', 'Sound City - Real to Reel'),

```

```

12 ('Jake Bugg', 'Jake Bugg'),
13 ('Various Artists', 'The Trevor Nelson Collection'),
14 ('David Bowie', 'The Next Day'),
15 ('Mumford & Sons', 'Babel'),
16 ('The Lumineers', 'The Lumineers'),
17 ('Various Artists', 'Get Ur Freak On - R&B Anthems'),
18 ('The 1975', 'Music For Cars EP'),
19 ('Various Artists', 'Saturday Night Club Classics - Ministry of Sound'),
20 ('Hurts', 'Exile (Deluxe)'),
21 ('Various Artists', 'Mixmag - The Greatest Dance Tracks of All Time'),
22 ('Ben Howard', 'Every Kingdom'),
23 ('Stereophonics', 'Graffiti On the Train'),
24 ('The Script', '#3'),
25 ('Stornoway', 'Tales from Terra Firma'),
26 ('David Bowie', 'Hunky Dory (Remastered)'),
27 ('Worship Central', 'Let It Be Known (Live)'),
28 ('Ellie Goulding', 'Halcyon'),
29 ('Various Artists', 'Dermot O'Leary Presents the Saturday Sessions 2013'),
30 ('Stereophonics', 'Graffiti On the Train (Deluxe Version)'),
31 ('Dido', 'Girl Who Got Away (Deluxe)'),
32 ('Hurts', 'Exile'),
33 ('Bruno Mars', 'Doo-Wops & Hooligans'),
34 ('Calvin Harris', '18 Months'),
35 ('Olly Murs', 'Right Place Right Time'),
36 ('Alt-J (?)', 'An Awesome Wave'),
37 ('One Direction', 'Take Me Home'),
38 ('Various Artists', 'Pop Stars'),
39 ('Various Artists', 'Now That's What I Call Music! 83'),
40 ('John Grant', 'Pale Green Ghosts'),
41 ('Paloma Faith', 'Fall to Grace'),
42 ('Laura Mvula', 'Sing To the Moon (Deluxe)'),
43 ('Duke Dumont', 'Need U (100%) [feat. A*M*E] - EP'),
44 ('Watsky', 'Cardboard Castles'),
45 ('Blondie', 'Blondie: Greatest Hits'),
46 ('Foals', 'Holy Fire'),
47 ('Maroon 5', 'Overexposed'),
48 ('Bastille', 'Pompeii (Remixes) - EP'),
49 ('Imagine Dragons', 'Hear Me - EP'),
50 ('Various Artists', '100 Hits: 80s Classics'),
51 ('Various Artists', 'Les Misérables (Highlights From the Motion Picture
↳ Soundtrack)'),
52 ('Mumford & Sons', 'Sigh No More'),
53 ('Frank Ocean', 'Channel ORANGE'),
54 ('Bon Jovi', 'What About Now'),
55 ('Various Artists', 'BRIT Awards 2013'),
56 ('Taylor Swift', 'Red'),
57 ('Fleetwood Mac', 'Fleetwood Mac: Greatest Hits'),
58 ('David Guetta', 'Nothing But the Beat Ultimate'),
59 ('Various Artists', 'Clubbers Guide 2013 (Mixed By Danny Howard) - Ministry of
↳ Sound'),
60 ('David Bowie', 'Best of Bowie'),
61 ('Laura Mvula', 'Sing To the Moon'),
62 ('ADELE', '21'),
63 ('Of Monsters and Men', 'My Head Is an Animal'),
64 ('Rihanna', 'Unapologetic'),
65 ('Various Artists', 'BBC Radio 1's Live Lounge - 2012'),
66 ('Avicii & Nicky Romero', 'I Could Be the One (Avicii vs. Nicky Romero)'),
67 ('The Streets', 'A Grand Don't Come for Free'),

```

```

68 ('Tim McGraw', 'Two Lanes of Freedom'),
69 ('Foo Fighters', 'Foo Fighters: Greatest Hits'),
70 ('Various Artists', 'Now That\'s What I Call Running!'),
71 ('Swedish House Mafia', 'Until Now'),
72 ('The xx', 'Coexist'),
73 ('Five', 'Five: Greatest Hits'),
74 ('Jimi Hendrix', 'People, Hell & Angels'),
75 ('Biffy Clyro', 'Opposites (Deluxe)'),
76 ('The Smiths', 'The Sound of the Smiths'),
77 ('The Saturdays', 'What About Us - EP'),
78 ('Fleetwood Mac', 'Rumours'),
79 ('Various Artists', 'The Big Reunion'),
80 ('Various Artists', 'Anthems 90s - Ministry of Sound'),
81 ('The Vaccines', 'Come of Age'),
82 ('Nicole Scherzinger', 'Boomerang (Remixes) - EP'),
83 ('Bob Marley', 'Legend (Bonus Track Version)'),
84 ('Josh Groban', 'All That Echoes'),
85 ('Blue', 'Best of Blue'),
86 ('Ed Sheeran', '+'),
87 ('Olly Murs', 'In Case You Didn\'t Know (Deluxe Edition)'),
88 ('Macklemore & Ryan Lewis', 'The Heist (Deluxe Edition)'),
89 ('Various Artists', 'Defected Presents Most Rated Miami 2013'),
90 ('Gorgon City', 'Real EP'),
91 ('Mumford & Sons', 'Babel (Deluxe Version)'),
92 ('Various Artists', 'The Music of Nashville: Season 1, Vol. 1 (Original_
↳Soundtrack)'),
93 ('Various Artists', 'The Twilight Saga: Breaking Dawn, Pt. 2 (Original Motion_
↳Picture Soundtrack)'),
94 ('Various Artists', 'Mum - The Ultimate Mothers Day Collection'),
95 ('One Direction', 'Up All Night'),
96 ('Bon Jovi', 'Bon Jovi Greatest Hits'),
97 ('Agnetha Fältskog', 'A'),
98 ('Fun.', 'Some Nights'),
99 ('Justin Bieber', 'Believe Acoustic'),
100 ('Atoms for Peace', 'Amok'),
101 ('Justin Timberlake', 'Justified'),
102 ('Passenger', 'All the Little Lights'),
103 ('Kodaline', 'The High Hopes EP'),
104 ('Lana Del Rey', 'Born to Die'),
105 ('JAY Z & Kanye West', 'Watch the Throne (Deluxe Version)'),
106 ('Biffy Clyro', 'Opposites'),
107 ('Various Artists', 'Return of the 90s'),
108 ('Gabrielle Aplin', 'Please Don\'t Say You Love Me - EP'),
109 ('Various Artists', '100 Hits - Driving Rock'),
110 ('Jimi Hendrix', 'Experience Hendrix - The Best of Jimi Hendrix'),
111 ('Various Artists', 'The Workout Mix 2013'),
112 ('The 1975', 'Sex'),
113 ('Chase & Status', 'No More Idols'),
114 ('Rihanna', 'Unapologetic (Deluxe Version)'),
115 ('The Killers', 'Battle Born'),
116 ('Olly Murs', 'Right Place Right Time (Deluxe Edition)'),
117 ('A$AP Rocky', 'LONG.LIVE.A$AP (Deluxe Version)'),
118 ('Various Artists', 'Cooking Songs'),
119 ('Haim', 'Forever - EP'),
120 ('Lianne La Havas', 'Is Your Love Big Enough?'),
121 ('Michael Bublé', 'To Be Loved'),
122 ('Daughter', 'If You Leave'),
123 ('The xx', 'xx'),

```

```
124 ('Eminem', 'Curtain Call'),
125 ('Kendrick Lamar', 'good kid, m.A.A.d city (Deluxe)'),
126 ('Disclosure', 'The Face - EP'),
127 ('Palma Violets', '180'),
128 ('Cody Simpson', 'Paradise'),
129 ('Ed Sheeran', '+ (Deluxe Version)'),
130 ('Michael Bublé', 'Crazy Love (Hollywood Edition)'),
131 ('Bon Jovi', 'Bon Jovi Greatest Hits - The Ultimate Collection'),
132 ('Rita Ora', 'Ora'),
133 ('g33k', 'Spabby'),
134 ('Various Artists', 'Annie Mac Presents 2012'),
135 ('David Bowie', 'The Platinum Collection'),
136 ('Bridgit Mendler', 'Ready or Not (Remixes) - EP'),
137 ('Dido', 'Girl Who Got Away'),
138 ('Various Artists', 'Now That's What I Call Disney'),
139 ('The 1975', 'Facedown - EP'),
140 ('Kodaline', 'The Kodaline - EP'),
141 ('Various Artists', '100 Hits: Super 70s'),
142 ('Fred V & Grafix', 'Goggles - EP'),
143 ('Biffy Clyro', 'Only Revolutions (Deluxe Version)'),
144 ('Train', 'California 37'),
145 ('Ben Howard', 'Every Kingdom (Deluxe Edition)'),
146 ('Various Artists', 'Motown Anthems'),
147 ('Courteeners', 'ANNA'),
148 ('Johnny Marr', 'The Messenger'),
149 ('Rodriguez', 'Searching for Sugar Man'),
150 ('Jessie Ware', 'Devotion'),
151 ('Bruno Mars', 'Unorthodox Jukebox'),
152 ('Various Artists', 'Call the Midwife (Music From the TV Series)'
153 );
```

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.

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

```
1 <?php
2 namespace Album\Model;
3
4 use Zend\Db\ResultSet\ResultSet;
5 use Zend\Db\TableGateway\TableGateway;
6 use Zend\Db\Sql\Select;
7 use Zend\Paginator\Adapter\DbSelect;
8 use Zend\Paginator\Paginator;
9
10 class AlbumTable
11 {
12     ...
```

```

13 public function fetchAll($paginated=false)
14 {
15     if ($paginated) {
16         // create a new Select object for the table album
17         $select = new Select('album');
18         // create a new result set based on the Album entity
19         $resultSetPrototype = new ResultSet();
20         $resultSetPrototype->setArrayObjectPrototype(new Album());
21         // create a new pagination adapter object
22         $paginatorAdapter = new DbSelect(
23             // our configured select object
24             $select,
25             // the adapter to run it against
26             $this->tableGateway->getAdapter(),
27             // the result set to hydrate
28             $resultSetPrototype
29         );
30         $paginator = new Paginator($paginatorAdapter);
31         return $paginator;
32     }
33     $resultSet = $this->tableGateway->select();
34     return $resultSet;
35 }
36 ...

```

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.

Modifying the AlbumController

Next, we need to tell the album controller to return a `Pagination` object instead of a `ResultSet`. Both these objects can be iterated over to return hydrated `Album` objects, so we won't need to make many changes to the view script:

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

```

1  ...
2  public function indexAction()
3  {
4      // grab the paginator from the AlbumTable
5      $paginator = $this->getAlbumTable()->fetchAll(true);
6      // set the current page to what has been passed in query string, or to 1 if none
7      ↪set $paginator->setCurrentPageNumber((int) $this->params()->fromQuery('page', 1));
8      // set the number of items per page to 10
9      $paginator->setItemCountPerPage(10);
10
11     return new ViewModel(array(
12         'paginator' => $paginator
13     ));
14 }
15 ...

```

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.

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`

```
1 <table class="table">
2     <tr>
3         <th>Title</th>
4         <th>Artist</th>
5         <th>&nbsp;</th>
6     </tr>
7     <?php foreach ($this->paginator as $album) : // <-- change here! ?>
8         <tr>
9             <td><?php echo $this->escapeHtml($album->title); ?></td>
10            <td><?php echo $this->escapeHtml($album->artist); ?></td>
11            <td>
12                <a href="<?php echo $this->url('album',
13                    array('action' => 'edit', 'id' => $album->id)); ?>">Edit</a>
14                <a href="<?php echo $this->url('album',
15                    array('action' => 'delete', 'id' => $album->id)); ?>">Delete</a>
16            </td>
17        </tr>
18    <?php endforeach; ?>
19 </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...

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 partial in the `module/Application/view/partial/` folder, so that we can use the control in all our modules:

`module/Application/view/partial/paginator.phtml`

```
1 <?php if ($this->pageCount): ?>
2     <div>
3         <ul class="pagination">
4             <!-- Previous page link -->
5             <?php if (isset($this->previous)): ?>
6                 <li>
7                     <a href="<?php echo $this->url($this->route); ?>?page=<?php echo
8                     <?php echo $this->previous; ?>"><<
9                     </a>
10                </li>
11            <?php else: ?>
```

```

12         <li class="disabled">
13             <a href="#">
14                 <<
15             </a>
16         </li>
17     <?php endif; ?>
18
19     <!-- Numbered page links -->
20     <?php foreach ($this->pagesInRange as $page): ?>
21         <?php if ($page != $this->current): ?>
22             <li>
23                 <a href="<?php echo $this->url($this->route); ?>?page=<?php _
24                 <?php echo $page; ?>
25                 </a>
26             </li>
27         <?php else: ?>
28             <li class="active">
29                 <a href="#"><?php echo $page; ?></a>
30             </li>
31         <?php endif; ?>
32     <?php endforeach; ?>
33
34     <!-- Next page link -->
35     <?php if (isset($this->next)): ?>
36         <li>
37             <a href="<?php echo $this->url($this->route); ?>?page=<?php echo
38             >>
39             </a>
40         </li>
41     <?php else: ?>
42         <li class="disabled">
43             <a href="#">
44                 >>
45             </a>
46         </li>
47     <?php endif; ?>
48 </ul>
49 </div>
50 <?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.

Using the PaginationControl View Helper

The only thing left for us to do so that we can page through the albums 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

```
1  ...
2  <?php
3  // add at the end of the file after the table
4  echo $this->paginationControl(
5      // the paginator object
6      $this->paginator,
7      // the scrolling style
8      'sliding',
9      // the partial to use to render the control
10     'partial/paginator.phtml',
11     // the route to link to when a user clicks a control link
12     array(
13         'route' => 'album'
14     )
15 );
16 ?>
```

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!

Setting up a database adapter

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.

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:

```
1 // config/autoload/global.php
2
3 return array(
4     'db' => array(
5         'driver'      => 'Pdo',
6         'dsn'         => 'mysql:dbname=zf2tutorial;host=localhost',
7     ),
8     'service_manager' => array(
9         'factories' => array(
10             'Zend\Db\Adapter\Adapter' => 'Zend\Db\Adapter\AdapterServiceFactory',
11         ),
12     ),
13 );
```

The adapter can then be accessed in any `ServiceLocatorAware` classes.

```
1 public function getAdapter()
2 {
3     if (!$this->adapter) {
```

```

4     $sm = $this->getServiceLocator();
5     $this->adapter = $sm->get('Zend\Db\Adapter\Adapter');
6 }
7 return $this->adapter;
8 }

```

More information on adapter options can be found in the docs for [Zend\Db\Adapter](#).

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:

```

1 // config/autoload/global.php
2
3 return array(
4     'db' => array(
5         'driver'      => 'Pdo',
6         'dsn'         => 'mysql:dbname=zf2tutorial;host=localhost',
7     ),
8     'service_manager' => array(
9         'factories' => array(
10             'Zend\Db\Adapter\Adapter' => function ($serviceManager) {
11                 $adapterFactory = new Zend\Db\Adapter\AdapterServiceFactory();
12                 $adapter = $adapterFactory->createService($serviceManager);
13
14                 \Zend\Db\TableGateway\Feature\GlobalAdapterFeature::setStaticAdapter(
15                     ↪$adapter);
16
17                 return $adapter;
18             }
19         ),
20     );

```

The adapter can then later be fetched using `Zend\Db\TableGateway\Feature\GlobalAdapterFeature::getStaticAdapter()` for use in e.g. `Zend\Validator\DbRecordExists`:

```

1 $validator = new Zend\Validator\Db\RecordExists(
2     array(
3         'table'      => 'users',
4         'field'      => 'emailaddress',
5         'adapter' => ↪
6         ↪\Zend\Db\TableGateway\Feature\GlobalAdapterFeature::getStaticAdapter()
7     )
8 );

```

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.

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 make more sense to your application.

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.

- 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 have 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.

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:

- Create a map of all the old files, new files, old classes and new classes.
- 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:

```
1 <?php return array (
2     array (
3         'root_directory' => '/realpath/to/project/application/models',
4         'original_class' => 'Application_Model_Invoice',
5         'original_file' => '/realpath/to/project/application/models/Invoice.php',
6         'new_namespace' => 'Application\\Model',
7         'new_class' => 'Invoice',
8         'new_file' => '/realpath/to/project/application/models/Application/Model/
↳ Invoice.php',
9     ),
10     ...
11 );
```

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/DirectPayment.php
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:

```
1 namespace Application\Model;  
2  
3 use Application\Model\PurchaseOrder;  
4 use Application\Model\Transaction;  
5 use Zend_Filter_Alnum;  
6  
7 class Invoice  
8 {  
9  
10     protected $tickets;  
11     protected $transaction;  
12  
13     ...  
14 }
```

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.

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.

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`:

```
1 define('ZF2_PATH', '/path/to/zf2/library');
2 require_once ZF2_PATH . '/Zend/Loader/StandardAutoloader.php';
3 $loader = new Zend\Loader\StandardAutoloader(array(
4     'autoregister_zf' => true,
5 ));
6 $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')`).

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:

```
1 "require": {  
2     "php": ">=5.3.23",  
3     "zendframework/zendframework1": "1.12",  
4     ...  
5 }
```

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](#).

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):

```
1 SetEnv APPLICATION_ENV development  
2 RewriteEngine On  
3 RewriteCond %{REQUEST_FILENAME} -s [OR]  
4 RewriteCond %{REQUEST_FILENAME} -l [OR]  
5 RewriteCond %{REQUEST_FILENAME} -d  
6 RewriteRule ^ - [NC,L]  
7 RewriteRule ^album(/.*)?$ index_zf2.php [NC,L]  
8 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`:

```
1 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](#).

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.

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:

```
1 use Zend\Authentication\Adapter\AdapterInterface;
2
3 class My\Auth\Adapter implements AdapterInterface
4 {
5     /**
6      * Sets username and password for authentication
7      *
8      * @return void
9      */
10    public function __construct($username, $password)
11    {
12        // ...
13    }
14
15    /**
16     * Performs an authentication attempt
17     *
18     * @return \Zend\Authentication\Result
19     * @throws \Zend\Authentication\Adapter\Exception\ExceptionInterface
20     *         If authentication cannot be performed
21     */
22    public function authenticate()
23    {
24        // ...
25    }
26 }
```

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`.

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:


```

1 use Zend\Authentication\Result;
2
3 Result::SUCCESS
4 Result::FAILURE
5 Result::FAILURE_IDENTITY_NOT_FOUND
6 Result::FAILURE_IDENTITY_AMBIGUOUS
7 Result::FAILURE_CREDENTIAL_INVALID
8 Result::FAILURE_UNCATEGORIZED

```

The following example illustrates how a developer may branch on the result code:

```

1 // inside of AuthController / loginAction
2 $result = $this->auth->authenticate($adapter);
3
4 switch ($result->getCode()) {
5
6     case Result::FAILURE_IDENTITY_NOT_FOUND:
7         /** do stuff for nonexistent identity */
8         break;
9
10    case Result::FAILURE_CREDENTIAL_INVALID:
11        /** do stuff for invalid credential */
12        break;
13
14    case Result::SUCCESS:
15        /** do stuff for successful authentication */
16        break;
17
18    default:
19        /** do stuff for other failure */
20        break;
21 }

```

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.

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 *ZendSession*. 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.

```
1 use Zend\Authentication\AuthenticationService;
2 use Zend\Authentication\Storage\Session as SessionStorage;
3
4 $auth = new AuthenticationService();
5
6 // Use 'someNamespace' instead of 'Zend_Auth'
7 $auth->setStorage(new SessionStorage('someNamespace'));
8
9 /**
10  * @todo Set up the auth adapter, $authAdapter
11  */
12
13 // Authenticate, saving the result, and persisting the identity on
14 // success
15 $result = $auth->authenticate($authAdapter);
```

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:

```
1 $storage = new Chain;
2 $storage->add(new Session);
3 $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 Session 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.

```
1 $chain->add(new A, 2);
2 $chain->add(new B, 10); // First use B
```

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 implement `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`:

```

1  use Zend\Authentication\Storage\StorageInterface;
2
3  class My\Storage implements StorageInterface
4  {
5      /**
6       * Returns true if and only if storage is empty
7       *
8       * @throws \Zend\Authentication\Exception\ExceptionInterface
9       *         If it is impossible to
10      *         determine whether storage is empty
11      * @return boolean
12      */
13     public function isEmpty()
14     {
15         /**
16          * @todo implementation
17          */
18     }
19
20     /**
21      * Returns the contents of storage
22      *
23      * Behavior is undefined when storage is empty.
24      *
25      * @throws \Zend\Authentication\Exception\ExceptionInterface
26      *         If reading contents from storage is impossible
27      * @return mixed
28      */
29
30     public function read()
31     {
32         /**
33          * @todo implementation
34          */
35     }
36
37     /**
38      * Writes $contents to storage
39      *
40      * @param mixed $contents
41      * @throws \Zend\Authentication\Exception\ExceptionInterface
42      *         If writing $contents to storage is impossible
43      * @return void
44      */
45

```

```
46     public function write($contents)
47     {
48         /**
49          * @todo implementation
50          */
51     }
52
53     /**
54      * Clears contents from storage
55      *
56      * @throws \Zend\Authentication\Exception\ExceptionInterface
57      *          If clearing contents from storage is impossible
58      * @return void
59      */
60
61     public function clear()
62     {
63         /**
64          * @todo implementation
65          */
66     }
67 }
```

In order to use this custom storage class, `Zend\Authentication\AuthenticationService::setStorage()` is invoked before an authentication query is attempted:

```
1  use Zend\Authentication\AuthenticationService;
2
3  // Instruct AuthenticationService to use the custom storage class
4  $auth = new AuthenticationService();
5
6  $auth->setStorage(new My\Storage());
7
8  /**
9   * @todo Set up the auth adapter, $authAdapter
10  */
11
12  // Authenticate, saving the result, and persisting the identity on
13  // success
14  $result = $auth->authenticate($authAdapter);
```

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:

```
1  use Zend\Authentication\AuthenticationService;
2
3  // instantiate the authentication service
4  $auth = new AuthenticationService();
```

```

5
6 // Set up the authentication adapter
7 $authAdapter = new My\Auth\Adapter($username, $password);
8
9 // Attempt authentication, saving the result
10 $result = $auth->authenticate($authAdapter);
11
12 if (!$result->isValid()) {
13     // Authentication failed; print the reasons why
14     foreach ($result->getMessages() as $message) {
15         echo "$message\n";
16     }
17 } else {
18     // Authentication succeeded; the identity ($username) is stored
19     // in the session
20     // $result->getIdentity() === $auth->getIdentity()
21     // $result->getIdentity() === $username
22 }

```

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:

```

1 use Zend\Authentication\AuthenticationService;
2
3 $auth = new AuthenticationService();
4
5 /**
6  * @todo Set up the auth adapter, $authAdapter
7  */
8
9 if ($auth->hasIdentity()) {
10     // Identity exists; get it
11     $identity = $auth->getIdentity();
12 }

```

To remove an identity from persistent storage, simply use the `clearIdentity()` method. This typically would be used for implementing an application “logout” operation:

```

1 $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`:

```

1 // Set up the authentication adapter
2 $authAdapter = new My\Auth\Adapter($username, $password);
3
4 // Attempt authentication, saving the result
5 $result = $authAdapter->authenticate();
6
7 if (!$result->isValid()) {
8     // Authentication failed; print the reasons why
9     foreach ($result->getMessages() as $message) {
10         echo "$message\n";
11     }

```

```
12 } else {  
13     // Authentication succeeded  
14     // $result->getIdentity() === $username  
15 }
```

Database Table Authentication

Note: `Zend\Authentication\Adapter\DbTable` has been deprecated, as its responsibilities have been splitted off into `Zend\Authentication\Adapter\DbTable\CallbackCheck` and `Zend\Authentication\Adapter\DbTable\CredentialTreatmentAdapter`. Use `Zend\Authentication\Adapter\DbTable\CredentialTreatmentAdapter` instead of `Zend\Authentication\Adapter\DbTable`.

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.

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:

```
1 use Zend\Db\Adapter\Adapter as DbAdapter;
2
3 // Create a SQLite database connection
4 $dbAdapter = new DbAdapter(array(
5     'driver' => 'Pdo_Sqlite',
6     'database' => 'path/to/sqlite.db'
7 ));
8
9 // Build a simple table creation query
10 $sqlCreate = 'CREATE TABLE [users] ('
11     . '[id] INTEGER NOT NULL PRIMARY KEY, '
12     . '[username] VARCHAR(50) UNIQUE NOT NULL, '
13     . '[password] VARCHAR(32) NULL, '
14     . '[real_name] VARCHAR(150) NULL)';
15
16 // Create the authentication credentials table
17 $dbAdapter->query($sqlCreate);
18
19 // Build a query to insert a row for which authentication may succeed
20 $sqlInsert = "INSERT INTO users (username, password, real_name) "
21     . "VALUES ('my_username', 'my_password', 'My Real Name')";
22
23 // Insert the data
24 $dbAdapter->query($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:

```
1 use Zend\Authentication\Adapter\DbTable as AuthAdapter;
2
3 // Configure the instance with constructor parameters...
4 $authAdapter = new AuthAdapter($dbAdapter,
5     'users',
6     'username',
7     'password'
8 );
9
10 // ...or configure the instance with setter methods
11 $authAdapter = new AuthAdapter($dbAdapter);
12
13 $authAdapter
14     ->setTableName('users')
15     ->setIdentityColumn('username')
16     ->setCredentialColumn('password')
17 ;
```

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:

```

1 // Set the input credential values (e.g., from a login form)
2 $authAdapter
3     ->setIdentity('my_username')
4     ->setCredential('my_password')
5 ;
6
7 // 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:

```

1 // Print the identity
2 echo $result->getIdentity() . "\n\n";
3
4 // Print the result row
5 print_r($authAdapter->getResultRowObject());
6
7 /* Output:
8 my_username
9
10 Array
11 (
12     [id] => 1
13     [username] => my_username
14     [password] => my_password
15     [real_name] => My Real Name
16 )
17 */

```

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:

```

1 $columnsToReturn = array(
2     'id', 'username', 'real_name'
3 );
4 print_r($authAdapter->getResultRowObject($columnsToReturn));
5
6 /* Output:
7
8 Array
9 (
10     [id] => 1
11     [username] => my_username
12     [real_name] => My Real Name
13 )
14 */
15
16 $columnsToOmit = array('password');
17 print_r($authAdapter->getResultRowObject(null, $columnsToOmit));
18
19 /* Output:
20
21 Array
22 (

```

```
23     [id] => 1
24     [username] => my_username
25     [real_name] => My Real Name
26 )
27 */
```

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:

```
1  // authenticate with Zend\Authentication\Adapter\DbTable
2  $result = $this->_auth->authenticate($adapter);
3
4  if ($result->isValid()) {
5      // store the identity as an object where only the username and
6      // real_name have been returned
7      $storage = $this->_auth->getStorage();
8      $storage->write($adapter->getResultRowObject(array(
9          'username',
10         'real_name',
11     )));
12
13     // store the identity as an object where the password column has
14     // been omitted
15     $storage->write($adapter->getResultRowObject(
16         null,
17         'password'
18     ));
19
20     /* ... */
21
22 } else {
23
24     /* ... */
25
26 }
```

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.

```
1  use Zend\Authentication\Adapter\DbTable as AuthAdapter;
2
```

```

3 // The status field value of an account is not equal to "compromised"
4 $adapter = new AuthAdapter($db,
5     'users',
6     'username',
7     'password',
8     'MD5(?) AND status != "compromised"'
9 );
10
11 // The active field value of an account is equal to "TRUE"
12 $adapter = new AuthAdapter($db,
13     'users',
14     'username',
15     'password',
16     'MD5(?) AND active = "TRUE"'
17 );

```

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:

```

1 $sqlAlter = "ALTER TABLE [users] "
2     . "ADD COLUMN [password_salt] "
3     . "AFTER [password]";

```

Here's a simple way to generate a salt string for every user at registration:

```

1 $dynamicSalt = '';
2 for ($i = 0; $i < 50; $i++) {
3     $dynamicSalt .= chr(rand(33, 126));
4 }

```

And now let's build the adapter:

```

1 $adapter = new AuthAdapter($db,
2     'users',
3     'username',
4     'password',
5     "MD5(CONCAT('staticSalt', ?, password_salt))"
6 );

```

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.

```
1 // Continuing with the example from above
2 $adapter = new AuthAdapter($db,
3                             'users',
4                             'username',
5                             'password',
6                             'MD5(?) '
7                             );
8
9 // get select object (by reference)
10 $select = $adapter->getDbSelect();
11 $select->where('active = "TRUE"');
12
13 // authenticate, this ensures that users.active = TRUE
14 $adapter->authenticate();
```

Digest Authentication

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
```

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()`.

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.

```
1 use Zend\Authentication\Adapter\Digest as AuthAdapter;
2
3 $adapter = new AuthAdapter($filename,
4                             $realm,
5                             $username,
6                             $password);
7
8 $result = $adapter->authenticate();
9
10 $identity = $result->getIdentity();
11
12 print_r($identity);
13
14 /*
15 Array
16 (
17     [realm] => Some Realm
18     [username] => someUser
19 )
20 */
```

HTTP Authentication Adapter

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.

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.

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 49.1: Configuration Options

Option Name	Required	Description
<code>accept_schemes</code>	Yes	Determines which authentication schemes the adapter will accept from the client. Must be a space-separated list containing 'basic' and/or 'digest'.
<code>realm</code>	Yes	Sets the authentication realm; usernames should be unique within a given realm.
<code>digest_domains</code>	Yes, when <code>accept_schemes</code> contains <code>digest</code>	Space-separated list of URIs for which the same authentication information is valid. The URIs need not all point to the same server.
<code>nonce_timeout</code>	Yes, when <code>accept_schemes</code> contains <code>digest</code>	Sets the number of seconds for which the nonce is valid. See notes below.
<code>use_opaque</code>	No	Specifies whether to send the opaque value in the header. True by default.
<code>algorithm</code>	No	Specified the algorithm. Defaults to MD5, the only supported option (for now).
<code>proxy_auth</code>	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.

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.

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:

```
1 use Zend\Authentication\Adapter\Http\FileResolver;
2 $path      = 'files/passwd.txt';
3 $resolver = new FileResolver($path);
```

or

```
1 $path      = 'files/passwd.txt';
2 $resolver = new FileResolver();
3 $resolver->setFile($path);
```

If the given path is empty or not readable, an exception is thrown.

Basic Usage

First, set up an array with the required configuration values:

```
1 $config = array(
2     'accept_schemes' => 'basic digest',
3     'realm'          => 'My Web Site',
4     'digest_domains' => '/members_only /my_account',
5     'nonce_timeout'  => 3600,
6 );
```

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:

```
1 $adapter = new Zend\Authentication\Adapter\Http($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:

```
1 use Zend\Authentication\Adapter\Http\FileResolver;
2
3 $basicResolver = new FileResolver();
4 $basicResolver->setFile('files/basicPasswd.txt');
5
6 $digestResolver = new FileResolver();
7 $digestResolver->setFile('files/digestPasswd.txt');
8
9 $adapter->setBasicResolver($basicResolver);
10 $adapter->setDigestResolver($digestResolver);
```

Finally, we perform the authentication. The adapter needs a reference to both the Request and Response objects in order to do its job:

```
1  assert($request instanceof Zend\Http\Request);
2  assert($response instanceof Zend\Http\Response);
3
4  $adapter->setRequest($request);
5  $adapter->setResponse($response);
6
7  $result = $adapter->authenticate();
8  if (!$result->isValid()) {
9      // Bad username/password, or canceled password prompt
10 }
```

LDAP Authentication

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.

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:

```
1 use Zend\Authentication\AuthenticationService;
2 use Zend\Authentication\Adapter\Ldap as AuthAdapter;
3 use Zend\Config\Reader\Ini as ConfigReader;
4 use Zend\Config\Config;
5 use Zend\Log\Logger;
6 use Zend\Log\Writer\Stream as LogWriter;
7 use Zend\Log\Filter\Priority as LogFilter;
8
9 $username = $this->getRequest()->getPost('username');
10 $password = $this->getRequest()->getPost('password');
11
12
13 $auth = new AuthenticationService();
14
15 $configReader = new ConfigReader();
```

```

16 $configData = $configReader->fromFile('./ldap-config.ini');
17 $config = new Config($configData, true);
18
19 $log_path = $config->production->ldap->log_path;
20 $options = $config->production->ldap->toArray();
21 unset($options['log_path']);
22
23 $adapter = new AuthAdapter($options,
24                             $username,
25                             $password);
26
27 $result = $auth->authenticate($adapter);
28
29 if ($log_path) {
30     $messages = $result->getMessages();
31
32     $logger = new Logger;
33     $writer = new LogWriter($log_path);
34
35     $logger->addWriter($writer);
36
37     $filter = new LogFilter(Logger::DEBUG);
38     $writer->addFilter($filter);
39
40     foreach ($messages as $i => $message) {
41         if ($i-- > 1) { // $messages[2] and up are log messages
42             $message = str_replace("\n", "\n ", $message);
43             $logger->debug("Ldap: $i: $message");
44         }
45     }
46 }

```

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 DN's shown below).

```

1  [production]
2
3  ldap.log_path = /tmp/ldap.log
4
5  ; Typical options for OpenLDAP
6  ldap.server1.host = s0.foo.net
7  ldap.server1.accountDomainName = foo.net
8  ldap.server1.accountDomainNameShort = FOO
9  ldap.server1.accountCanonicalForm = 3
10 ldap.server1.username = "CN=user1,DC=foo,DC=net"
11 ldap.server1.password = pass1
12 ldap.server1.baseDn = "OU=Sales,DC=foo,DC=net"
13 ldap.server1.bindRequiresDn = true
14

```

```

15 ; Typical options for Active Directory
16 ldap.server2.host = dc1.w.net
17 ldap.server2.useStartTls = true
18 ldap.server2.accountDomainName = w.net
19 ldap.server2.accountDomainNameShort = W
20 ldap.server2.accountCanonicalForm = 3
21 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).

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):

```

1 Array
2 (
3     [server2] => Array
4         (
5             [host] => dc1.w.net
6             [useStartTls] => 1
7             [accountDomainName] => w.net
8             [accountDomainNameShort] => W
9             [accountCanonicalForm] => 3
10            [baseDn] => CN=Users,DC=w,DC=net
11        )
12
13    [server1] => Array
14        (
15            [host] => s0.foo.net
16            [accountDomainName] => foo.net
17            [accountDomainNameShort] => FOO
18            [accountCanonicalForm] => 3
19            [username] => CN=user1,DC=foo,DC=net
20            [password] => pass1
21            [baseDn] => OU=Sales,DC=foo,DC=net
22            [bindRequiresDn] => 1
23        )
24
25 )

```

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.

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 50.1: 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 <code>alice@foo.net</code> , a baseDn of <code>DC=foo,DC=net</code> should work. A more precise location (e.g., <code>OU=Sales,DC=foo,DC=net</code>) 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., <code>alice</code>), 3 for backslash-style names (e.g., <code>FOO\alice</code>) or 4 for principal style usernames (e.g., <code>alice@foo.net</code>). The default value is 4 (e.g., <code>alice@foo.net</code>). For example, with a value of 3, the identity returned by <code>Zend\Authentication\Result::getIdentity()</code> (and <code>Zend\Authentication\AuthenticationService::getIdentity()</code> , if <code>Zend\Authentication\AuthenticationService</code> was used) will always be <code>FOO\alice</code> , regardless of what form Alice supplied, whether it be <code>alice</code> , <code>alice@foo.net</code> , <code>FOO\alice</code> , <code>FoO\alIE</code> , <code>foo.net\alice</code> , etc. See the Account Name Canonicalization section in the <code>Zend\Ldap\Ldap</code> 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 <code>EXAMPLE\username</code> with an AD server but to <code>username@example.com</code> 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., <code>example.com</code>). 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

50.4. Server Options 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

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.

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 50.2: 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).

Common Options for Specific Servers

Options for Active Directory

For *ADS*, the following options are noteworthy:

Table 50.3: 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.

Options for OpenLDAP

For OpenLDAP or a generic *LDAP* server using a typical *posixAccount* style schema, the following options are noteworthy:

Table 50.4: 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).

Authentication Validator

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`

Basic Usage

```
1 use Zend\Authentication\AuthenticationService;
2 use Zend\Authentication\Validator\Authentication as AuthenticationValidator;
3
4 $service = new AuthenticationService();
5 $adapter = new My\Authentication\Adapter();
6 $validator = new AuthenticationValidator(array(
7     'service' => $service,
8     'adapter' => $adapter,
9 ));
10
11 $validator->setCredential('myCredentialContext');
12 $validator->isValid('myIdentity', array(
13     'myCredentialContext' => 'myCredential',
14 ));
```

Introduction to Zend\Barcode

Overview

`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. Renderers allow you to draw barcodes based on the support required.

Barcode creation using Zend\Barcode\Barcode class

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.

```
1 use Zend\Barcode\Barcode;
2
3 // Only the text to draw is required
4 $barcodeOptions = array('text' => 'ZEND-FRAMEWORK');
5
6 // No required options
7 $rendererOptions = array();
8 $renderer = Barcode::factory(
9     'code39', 'image', $barcodeOptions, $rendererOptions
10 );
```

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.

```
1 use Zend\Config\Config;
2 use Zend\Barcode\Barcode;
3
4 // Using only one Zend\Config\Config object
5 $config = new Config(array(
6     'barcode' => 'code39',
7     'barcodeParams' => array('text' => 'ZEND-FRAMEWORK'),
8     'renderer' => 'image',
9     'rendererParams' => array('imageType' => 'gif'),
10 ));
11
12 $renderer = Barcode::factory($config);
```

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

```
1 use Zend\Barcode\Barcode;
2
3 // Only the text to draw is required
4 $barcodeOptions = array('text' => 'ZEND-FRAMEWORK');
5
6 // No required options
7 $rendererOptions = array();
8
9 // Draw the barcode in a new image,
10 $imageResource = Barcode::factory(
11     'code39', 'image', $barcodeOptions, $rendererOptions
12 )->draw();
```

Drawing a barcode with Zend\Barcode\Barcode::draw()

```
1 use Zend\Barcode\Barcode;
2
3 // Only the text to draw is required
4 $barcodeOptions = array('text' => 'ZEND-FRAMEWORK');
5
6 // No required options
7 $rendererOptions = array();
8
9 // Draw the barcode in a new image,
10 $imageResource = Barcode::draw(
11     'code39', 'image', $barcodeOptions, $rendererOptions
12 );
```


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

```

1  use Zend\Barcode\Barcode;
2
3  // Only the text to draw is required
4  $barcodeOptions = array('text' => 'ZEND-FRAMEWORK');
5
6  // No required options
7  $rendererOptions = array();
8
9  // Draw the barcode in a new image,
10 // send the headers and the image
11 Barcode::factory(
12     'code39', 'image', $barcodeOptions, $rendererOptions
13 )->render();

```

This will generate this barcode:



Rendering a barcode with `Zend\Barcode\Barcode::render()`

```

1  use Zend\Barcode\Barcode;
2
3  // Only the text to draw is required
4  $barcodeOptions = array('text' => 'ZEND-FRAMEWORK');
5
6  // No required options
7  $rendererOptions = array();
8
9  // Draw the barcode in a new image,
10 // send the headers and the image
11 Barcode::render(
12     'code39', 'image', $barcodeOptions, $rendererOptions
13 );

```

This will generate the same barcode as the previous example.

Zend\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

```
1 use Zend\Barcode\Object;  
2  
3 $options = array('text' => 'ZEND-FRAMEWORK', 'barHeight' => 40);  
4  
5 // Case 1: constructor  
6 $barcode = new Object\Code39($options);  
7  
8 // Case 2: setOptions()  
9 $barcode = new Object\Code39();  
10 $barcode->setOptions($options);  
11  
12 // Case 3: individual setters  
13 $barcode = new Object\Code39();  
14 $barcode->setText('ZEND-FRAMEWORK')  
15     ->setBarHeight(40);
```

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 54.1: Common Options

Option	Data Type	Default Value	Description
barcode- Namespace	String	ZendBarcode\Objects	Namespace of the barcode; for example, if you need to extend the embedding objects
barHeight	Integer	50	Height of the bars
barThick- Width	Integer	3	Width of the thick bar
barThin- Width	Integer	1	Width of the thin bar
factor	Integer, Float, String or Boolean	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”)
back- ground- Color	Integer or String	0xFFFFFFFF (white)	Color of the background. Could be provided as an integer or as a HTML value (e.g. “#333333”)
orientation	Integer, Float, String or Boolean	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
withQuiet- Zones	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
withCheck- sum	Boolean	FALSE	Indicate whether or not the checksum is automatically added to the barcode
withCheck- sumInText	Boolean	FALSE	Indicate whether or not the checksum is displayed in the textual representation
text	String	NULL	The text to represent as a barcode

Particular case of static setBarcodeFont()

You can set a common font for all your objects by using the static method `ZendBarcode\Barcode::setBarcodeFont()`. This value can be always be overridden for individual objects by using the `setFont()` method.

```

1 use ZendBarcode\Barcode;
2
3 // In your bootstrap:
4 Barcode::setBarcodeFont('my_font.ttf');
```

```

5
6 // Later in your code:
7 Barcode::render(
8     'code39',
9     'pdf',
10    array('text' => 'ZEND-FRAMEWORK')
11 ); // will use 'my_font.ttf'
12
13 // or:
14 Barcode::render(
15     'code39',
16     'image',
17     array(
18         'text' => 'ZEND-FRAMEWORK',
19         'font' => 3
20     )
21 ); // will use the 3rd GD internal font

```

Common Additional Getters

Table 54.2: 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

orphan

Description of shipped barcodes

You will find below detailed information about all barcode types shipped by default with Zend Framework.

Zend\Barcode\Object\Error

ERROR:
'a' contains invalid characters

This barcode is a special case. It is internally used to automatically render an exception caught by the Zend\Barcode component.

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.

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.

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.

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 54.3: Zend\Barcode\Object\Code25interleaved Options

Option	Data Type	Default Value	Description
<code>withBearerBars</code>	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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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

```
1 use Zend\Barcode\Renderer;
2
3 $options = array('topOffset' => 10);
4
5 // Case 1
6 $renderer = new Renderer\Pdf($options);
7
8 // Case 2
9 $renderer = new Renderer\Pdf();
10 $renderer->setOptions($options);
11
12 // Case 3
13 $renderer = new Renderer\Pdf();
14 $renderer->setTopOffset(10);
```

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 55.1: Common Options

Option	Data Type	Default Value	Description
render-erNames-pace	String	Zend\Barcode\Renderer	Namespace of the renderer; for example, if you need to extend the renderers
horizon-talPosi-tion	String	“left”	Can be “left”, “center” or “right”. Can be useful with PDF or if the <code>setWidth()</code> method is used with an image renderer.
vertical-Position	String	“top”	Can be “top”, “middle” or “bottom”. Can be useful with PDF or if the <code>setHeight()</code> 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.
automati-cRender-Error	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.
module-Size	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”).

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 55.2: 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.
im-ageType	String	“png”	Specify the image format. Can be “png”, “jpeg”, “jpg” or “gif”.

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.

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.

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.

```

1 use Zend\Cache\StorageFactory;
2
3 // Via factory:
4 $cache = StorageFactory::factory(array(
5     'adapter' => array(
6         'name' => 'apc',
7         'options' => array('ttl' => 3600),
8     ),
9     'plugins' => array(
10        'exception_handler' => array('throw_exceptions' => false),
11    ),
12 ));
13
14 // Alternately:
15 $cache = StorageFactory::adapterFactory('apc', array('ttl' => 3600));
16 $plugin = StorageFactory::pluginFactory('exception_handler', array(
17     'throw_exceptions' => false,
18 ));
19 $cache->addPlugin($plugin);
20
21 // Or manually:
22 $cache = new Zend\Cache\Storage\Adapter\Apc();
23 $cache->getOptions()->setTtl(3600);
24
25 $plugin = new Zend\Cache\Storage\Plugin\ExceptionHandler();
26 $plugin->getOptions()->setThrowExceptions(false);
27 $cache->addPlugin($plugin);

```

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

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

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

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

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

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

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

The FlushableInterface

The `Zend\Cache\Storage\FlushableInterface` implements a method to flush the complete storage.

flush()

Flush the whole storage.

Return type boolean

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

The OptimizableInterface

The Zend\Cache\Storage\OptimizableInterface implements a method to run optimization processes on the storage.

optimize()

Optimize the storage.

Return type boolean

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

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 56.1: 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 56.2: Adapter specific options

Name	Data Type	Default Value	Description
namespace_separator	string	”.”	A separator for the namespace and prefix

The Dbal Adapter

The `Zend\Cache\Storage\Adapter\Dbal` 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 56.3: Capabilities

Capability	Value
supported-Datatypes	string, null => string, boolean => string, integer => string, double => string
supportedMeta-data	<none>
minTtl	0
maxKeyLength	0
namespaceIsPrefix	true
namespaceSeparator	<Option value of namespace_separator>

Table 56.4: 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.

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 56.5: Capabilities

Capability	Value
supported-Datatypes	string, null => string, boolean => string, integer => string, double => string
supportedMeta-data	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 56.6: 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 <code>clearstatcache()</code> 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_+\-]*\$/ Di	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 <code>umask</code> to set file and directory permissions

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 56.7: Capabilities

Capability	Value
<code>supportedDatatypes</code>	<code>null, boolean, integer, double, string, array (serialized), object (serialized)</code>
<code>supportedMetadata</code>	<code><none></code>
<code>minTtl</code>	<code>1</code>
<code>maxTtl</code>	<code>0</code>
<code>staticTtl</code>	<code>true</code>
<code>ttlPrecision</code>	<code>1</code>
<code>useRequestTime</code>	<code>false</code>
<code>expiredRead</code>	<code>false</code>
<code>maxKeyLength</code>	<code>255</code>
<code>namespaceIsPrefix</code>	<code>true</code>
<code>namespaceSeparator</code>	<code><none></code>

Table 56.8: Adapter specific options

Name	Data Type	Default Value	Description
<code>servers</code>	<code>array</code>	<code>[]</code>	List of servers in <code>[] = array(string host, integer port)</code>
<code>lib_options</code>	<code>array</code>	<code>[]</code>	Associative array of Lib-memcached 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

The Redis Adapter

The `Zend\Cache\Storage\Adapter\Redis` adapter stores cache items over the redis protocol. It’s using the required PHP extension `redis`.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\FlushableInterface`
- `Zend\Cache\Storage\TotalSpaceCapableInterface`

Table 56.9: Capabilities

Capability	Value
supportedDatatypes	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 56.10: Adapter specific options

Name	Data Type	Default Value	Description
database	integer	0	Set database identifier
lib_option	array	[]	Associative array of redis options where the array key is the option name
namespace_separator	string	”.”	A separator for the namespace and prefix
password	string	“”	Set password
persistent_id	string		Set persistent id (<i>name</i> of the connection, leave blank to not use a persistent connection)
resource_manager	string		Set the redis resource manager to use
server			<p>Server can be described as follows:</p> <ul style="list-style-type: none"> • URI: <code>/path/to/sock.sock</code> • Assoc: <code>array('host' => <host>[, 'port' => <port>[, 'timeout' => <timeout>])</code> • List: <code>array(<host>[, <port>[, <timeout>])</code>

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 56.11: Capabilities

Capability	Value
<code>supportedDatatypes</code>	<code>string, null, boolean, integer, double, array, object, resource</code>
<code>supportedMetadata</code>	<code>mtime</code>
<code>minTtl</code>	<code>1</code>
<code>maxTtl</code>	<code><Value of PHP_INT_MAX></code>
<code>staticTtl</code>	<code>false</code>
<code>ttlPrecision</code>	<code>0.05</code>
<code>useRequestTime</code>	<code>false</code>
<code>expiredRead</code>	<code>true</code>
<code>maxKeyLength</code>	<code>0</code>
<code>namespaceIsPrefix</code>	<code>false</code>

Table 56.12: Adapter specific options

Name	Data Type	Default Value	Description
memory_limit	string integer	<50% of ini value memory_limit>	<p>Limit of how much memory can PHP allocate to allow store items into this adapter</p> <ul style="list-style-type: none"> • 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.

The MongoDB Adapter

The `Zend\Cache\Storage\Adapter\MongoDB` adapter stores cache items into MongoDB, using either the PHP extension `mongo` OR a MongoDB polyfill library, such as `Mongofill`.

This adapter implements the following interfaces:

- `Zend\Cache\Storage\FlushableInterface`

Table 56.13: Capabilities

Capability	Value
supportedDatatypes	null, boolean, integer, double, string, array
supportedMetadata	_id
minTtl	0
maxTtl	0
staticTtl	true
ttlPrecision	1
useRequestTime	false
expiredRead	false
maxKeyLength	255
namespaceIsPrefix	true
namespaceSeparator	<Option value of namespace_separator>

Table 56.14: Adapter specific options

Name	Data Type	Default Value	Description
lib_option	array		<p>Associative array of options where the array keys are the option names and the values are the option values.</p> <ul style="list-style-type: none">• server: The mongodb server connection string (see the Mongo-Client docs); default 'mongodb://localhost:27017'• database: Name of database to use; default 'zend'• collection: Name of collection to use; default 'cache'• connectionOptions: Associative array of connection options (see the Mongo-Client docs); default ['fsync' => false, 'journal' => true]• driverOptions: Associative array of driver options (see the Mongo-Client docs); default []
namespace_separator	string	”.”	A separator for the namespace and prefix

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 56.15: 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 <code>apc.use_request_time</code> >
expiredRead	false
namespaceIsPrefix	true
namespaceSeparator	<Option value of <code>namespace_separator</code> >

Table 56.16: Adapter specific options

Name	Data Type	Default Value	Description
<code>namespace_separator</code>	string	”;	A separator for the namespace and prefix

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 56.17: 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 <code>xcache.var_maxttl</code> >
staticTtl	true
ttlPrecision	1
useRequestTime	true
expiredRead	false
maxKeyLength	5182
namespaceIsPrefix	true
namespaceSeparator	<Option value of <code>namespace_separator</code> >

Table 56.18: 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 <code>admin_user</code> and <code>admin_pass</code> This makes XCache administration functions accessible if <code>xcache.admin.enable_auth</code> is enabled without the need of HTTP-Authentication.
admin_user	string	“”	The username of <code>xcache.admin.user</code>
admin_pass	string	“”	The password of <code>xcache.admin.pass</code> in plain text

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 56.19: 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	::

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 56.20: 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	::

Examples

Basic usage

```
1 $cache = \Zend\Cache\StorageFactory::factory(array(
2     'adapter' => array(
3         'name' => 'filesystem'
4     ),
5     'plugins' => array(
6         // Don't throw exceptions on cache errors
7         'exception_handler' => array(
8             'throw_exceptions' => false
9         ),
10    )
11));
12 $key = 'unique-cache-key';
13 $result = $cache->getItem($key, $success);
14 if (!$success) {
15     $result = doExpensiveStuff();
16     $cache->setItem($key, $result);
17 }
```

Get multiple rows from db

```
1 // Instantiate the cache instance using a namespace for the same type of items
2 $cache = \Zend\Cache\StorageFactory::factory(array(
3     'adapter' => array(
4         'name' => 'filesystem'
5         // With a namespace we can indicate the same type of items
6         // -> So we can simple use the db id as cache key
7         'options' => array(
8             'namespace' => 'dbtable'
9         ),
10    ),
11    'plugins' => array(
12        // Don't throw exceptions on cache errors
13        'exception_handler' => array(
14            'throw_exceptions' => false
15        ),
16        // We store database rows on filesystem so we need to serialize them
17        'Serializer'
18    )
19));
20
21 // Load two rows from cache if possible
22 $ids = array(1, 2);
23 $results = $cache->getItems($ids);
24 if (count($results) < count($ids)) {
25     // Load rows from db if loading from cache failed
26     $missingIds = array_diff($ids, array_keys($results));
27     $missingResults = array();
28     $query = 'SELECT * FROM dbtable WHERE id IN (' . implode(',',
29     ↪ $missingIds) . ')';
30     foreach ($pdo->query($query, PDO::FETCH_ASSOC) as $row) {
31         $missingResults[ $row['id'] ] = $row;
32     }
33     // Update cache items of the loaded rows from db
```



```
34     $cache->setItems($missingResults);
35
36     // merge results from cache and db
37     $results = array_merge($results, $missingResults);
38 }
```

Zend\Cache\Storage\Capabilities

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`.

Available Methods

__construct (*Zend\Cache\Storage\StorageInterface* \$storage, *stdClass* \$marker, array \$capabilities = array(), *Zend\Cache\Storage\Capabilities* \$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

Examples

Get storage capabilities and do specific stuff in base of it

```

1 use Zend\Cache\StorageFactory;
2
3 $cache = StorageFactory::adapterFactory('filesystem');
4 $supportedDatatypes = $cache->getCapabilities()->getSupportedDatatypes();
5
6 // now you can run specific stuff in base of supported feature
7 if ($supportedDatatypes['object']) {
8     $cache->set($key, $object);
9 } else {
10     $cache->set($key, serialize($object));
11 }

```

Listen to change event

```

1 use Zend\Cache\StorageFactory;
2
3 $cache = StorageFactory::adapterFactory('filesystem', array(
4     'no_atime' => false,
5 ));
6
7 // Catching capability changes
8 $cache->getEventManager()->attach('capability', function($event) {
9     echo count($event->getParams()) . ' capabilities changed';
10 });

```

```
10  });  
11  
12  // change option which changes capabilities  
13  $cache->getOptions()->setNoATime(true);
```

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`).

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.

```
1  use Zend\Cache\StorageFactory;
2
3  // Via factory:
4  $cache = StorageFactory::factory(array(
5      'adapter' => 'filesystem',
6      'plugins' => array('serializer'),
7  ));
8
9  // Alternately:
10 $cache = StorageFactory::adapterFactory('filesystem');
11 $plugin = StorageFactory::pluginFactory('serializer');
12 $cache->addPlugin($plugin);
13
14 // Or manually:
15 $cache = new Zend\Cache\Storage\Adapter\Filesystem();
```

```
16 $plugin = new Zend\Cache\Storage\Plugin\Serializer();
17 $cache->addPlugin($plugin);
```

The ClearExpiredByFactor Plugin

The `Zend\Cache\Storage\Plugin\ClearExpiredByFactor` plugin calls the storage method `clearExpired()` randomly (by factor) after every call of `setItem()`, `setItems()`, `addItem()` and `addItems()`.

Table 58.1: Plugin specific options

Name	Data Type	Default Value	Description
<code>clearing_factor</code>	<code>integer</code>	<code>0</code>	The automatic clearing factor

Note: The `ClearExpiredInterface` is required

The storage have to implement the `Zend\Cache\Storage\ClearExpiredInterface` to work with this plugin.

The ExceptionHandler Plugin

The `Zend\Cache\Storage\Plugin\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 caught exception.

Table 58.2: Plugin specific options

Name	Data Type	Default Value	Description
<code>exception_callback</code>	<code>callable</code> <code>null</code>	<code>null</code>	Callback will be called on an exception and get the exception as argument
<code>throw_exceptions</code>	<code>boolean</code>	<code>true</code>	Re-throw caught exceptions

The IgnoreUserAbort Plugin

The `Zend\Cache\Storage\Plugin\IgnoreUserAbort` plugin ignores script terminations by users until write operations to cache finished.

Table 58.3: Plugin specific options

Name	Data Type	Default Value	Description
<code>exit_on_abort</code>	<code>boolean</code>	<code>true</code>	Terminate script execution if user abort the script

The OptimizeByFactor Plugin

The `Zend\Cache\Storage\Plugin\OptimizeByFactor` plugin calls the storage method `optimize()` randomly (by factor) after removing items from cache.

Table 58.4: Plugin specific options

Name	Data Type	Default Value	Description
<code>optimizing_factor</code>	<code>integer</code>	<code>0</code>	The automatic optimization factor

Note: The `OptimizableInterface` is required

The storage have to implement the `Zend\Cache\Storage\OptimizableInterface` to work with this plugin.

The Serializer Plugin

The `Zend\Cache\Storage\Plugin\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 58.5: Plugin specific options

Name	Data Type	Default Value	Description
<code>serializer</code>	<code>null</code> <code>string</code> <code>Zend\Serializer\Adapter\AdapterInterface</code>	<code>null</code>	The serializer to use <ul style="list-style-type: none"> • If <code>null</code> use the default serializer • If <code>string</code> instantiate the serializer with <code>serializer_options</code>
<code>serializer_options</code>	<code>array</code>	<code>[]</code>	Array of serializer options used to instantiate the serializer

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

Examples

Basics of writing an own storage plugin

```
1  use Zend\Cache\Storage\Event;
2  use Zend\Cache\Storage\Plugin\AbstractPlugin;
3  use Zend\EventManager\EventManagerInterface;
4
5  class MyPlugin extends AbstractPlugin
6  {
7
8      protected $handles = array();
9
10     // This method have to attach all events required by this plugin
11     public function attach(EventManagerInterface $events)
12     {
13         $this->handles[] = $events->attach('getItem.pre', array($this, 'onGetItemPre
14         ↪'));
15         $this->handles[] = $events->attach('getItem.post', array($this, 'onGetItemPost
16         ↪'));
17         return $this;
18     }
19
20     // This method have to detach all events required by this plugin
21     public function detach(EventManagerInterface $events)
22     {
23         foreach ($this->handles as $handle) {
24             $events->detach($handle);
25         }
26         $this->handles = array();
27         return $this;
28     }
29
30     public function onGetItemPre(Event $event)
31     {
32         $params = $event->getParams();
33         echo sprintf("Method 'getItem' with key '%s' started\n", $params['key']);
34     }
35
36     public function onGetItemPost(Event $event)
37     {
38         $params = $event->getParams();
39         echo sprintf("Method 'getItem' with key '%s' finished\n", $params['key']);
40     }
41 }
42
43 // After defining this basic plugin we can instantiate and add it to an adapter.
44 ↪instance
45 $plugin = new MyPlugin();
46 $cache->addPlugin($plugin);
```

```
44
45 // Now on calling getItem our basic plugin should print the expected output
46 $cache->getItem('cache-key');
47 // Method 'getItem' with key 'cache-key' started
48 // Method 'getItem' with key 'cache-key' finished
```


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.

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.

```
1 // Via the factory:
2 $callbackCache = Zend\Cache\PatternFactory::factory('callback', array(
3     'storage' => 'apc',
4 ));
```

```
1 // OR, the equivalent manual instantiation:
2 $callbackCache = new Zend\Cache\Pattern\CallbackCache();
3 $callbackCache->setOptions(new Zend\Cache\Pattern\PatternOptions(array(
```

```
4     'storage' => 'apc',  
5  ));
```

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`

Zend\Cache\Pattern\CallbackCache

Overview

The callback cache pattern caches calls of non specific functions and methods given as a callback.

Quick Start

For instantiation you can use the PatternFactory or do it manual:

```
1 use Zend\Cache\PatternFactory;
2 use Zend\Cache\Pattern\PatternOptions;
3
4 // Via the factory:
5 $callbackCache = PatternFactory::factory('callback', array(
6     'storage' => 'apc',
7     'cache_output' => true,
8 ));
9
10 // OR, the equivalent manual instantiation:
11 $callbackCache = new \Zend\Cache\Pattern\CallbackCache();
12 $callbackCache->setOptions(new PatternOptions(array(
13     'storage' => 'apc',
14     'cache_output' => true,
15 )));
```

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

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

Examples

Instantiating the callback cache pattern

```
1 use Zend\Cache\PatternFactory;
2
3 $callbackCache = PatternFactory::factory('callback', array(
4     'storage' => 'apc'
5 ));
```

Zend\Cache\Pattern\ClassCache

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.

Quick Start

Instantiating the class cache pattern

```
1 use Zend\Cache\PatternFactory;
2
3 $classCache = PatternFactory::factory('class', array(
4     'class' => 'MyClass',
5     'storage' => 'apc'
6 ));
```

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)

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*

Examples

Caching of import feeds

```

1  $cachedFeedReader = Zend\Cache\PatternFactory::factory('class', array(
2      'class'    => 'Zend\Feed\Reader\Reader',
3      'storage' => 'apc',
4
5      // The feed reader doesn't output anything
6      // so the output don't need to be caught and cached
7      'cache_output' => false,
8  ));
9
10 $feed = $cachedFeedReader->call("import", array('http://www.planet-php.net/rdf/'));
11 // OR
12 $feed = $cachedFeedReader->import('http://www.planet-php.net/rdf/');
```

Zend\Cache\Pattern\ObjectCache

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.

Quick Start

Instantiating the object cache pattern

```
1 use Zend\Cache\PatternFactory;
2
3 $object      = new stdClass();
4 $objectCache = PatternFactory::factory('object', array(
5     'object' => $object,
6     'storage' => 'apc'
7 ));
```

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 <code>cache_by_default</code> is disabled)
object_non_cache_methods	array	[]	List of methods to no-cache (If <code>cache_by_default</code> is enabled)
object_cache_magic_properties	boolean	false	Cache calls of magic object properties

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

Examples

Caching a filter

```

1  $filter      = new Zend\Filter\RealPath();
2  $cachedFilter = Zend\Cache\PatternFactory::factory('object', array(
3      'object'      => $filter,
4      'object_key' => 'RealpathFilter',
5      'storage'     => 'apc',
6
7      // The realpath filter doesn't output anything
8      // so the output don't need to be caught and cached
9      'cache_output' => false,
10 ));
11
12 $path = $cachedFilter->call("filter", array('/www/var/path/../../mypath'));
13 // OR
14 $path = $cachedFilter->filter('/www/var/path/../../mypath');
```

Zend\Cache\Pattern\OutputCache

Overview

The `OutputCache` pattern caches output between calls to `start()` and `end()`.

Quick Start

Instantiating the output cache pattern

```

1 use Zend\Cache\PatternFactory;
2
3 $outputCache = PatternFactory::factory('output', array(
4     'storage' => 'apc'
5 ));

```

Configuration Options

Option	Data Type	Default Value	Description
storage	string array Zend\Cache\Storage\StorageInterface	<none>	The storage to write/read cached data

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*

Examples

Caching simple view scripts

```
1 $outputCache = Zend\Cache\PatternFactory::factory('output', array(  
2     'storage' => 'apc',  
3 ));  
4  
5 $outputCache->start('mySimpleViewScript');  
6 include '/path/to/view/script.phtml';  
7 $outputCache->end();
```

Zend\Cache\Pattern\CaptureCache

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.

Quick Start

Simplest usage as Apache-404 handler

```
1 # .htdocs
2 ErrorDocument 404 /index.php
```

```
1 // index.php
2 use Zend\Cache\PatternFactory;
3 $capture = Zend\Cache\PatternFactory::factory('capture', array(
4     'public_dir' => __DIR__,
5 ));
6
7 // Start capturing all output excl. headers and write to public directory
8 $capture->start();
9
10 // Don't forget to change HTTP response code
11 header('Status: 200', true, 200);
12
13 // do stuff to dynamically generate output
```

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	0600 (false on win)	Set permissions of generated output files
	boolean		
dir_permission	integer	0700 (false on win)	Set permissions of generated output directories
	boolean		
umask	integer	false	Using umask on generating output files / directories
	boolean		

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*

Examples

Scaling images in base of request

```
1 # .htdocs
2 ErrorDocument 404 /index.php
```

```
1 // index.php
2 $captureCache = Zend\Cache\PatternFactory::factory('capture', array(
3     'public_dir' => __DIR__,
4 ));
5
6 // TODO
```

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.

Overview

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*.

The AdapterInterface

All *CAPTCHA* adapters implement `Zend\Captcha\AdapterInterface`, which looks like the following:

```
1 namespace Zend\Captcha;
2
3 use Zend\Validator\ValidatorInterface;
4
5 interface AdapterInterface extends ValidatorInterface
6 {
7     public function generate();
8
9     public function setName($name);
10
11    public function getName();
12
13    // Get helper name used for rendering this captcha type
14    public function getHelperName();
15 }
```

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*.

Basic Usage

A simple use case might look like the following:

```
1 // Originating request:
2 $captcha = new Zend\Captcha\Figlet(array(
```

```
3     'name' => 'foo',
4     'wordLen' => 6,
5     'timeout' => 300,
6 );
7
8 $id = $captcha->generate();
9
10 //this will output a Figlet string
11 echo $captcha->getFiglet()->render($captcha->getWord());
12
13
14 // On a subsequent request:
15 // Assume a captcha setup as before, with corresponding form fields, the value of $_
16 // ↪POST['foo']
17 // would be key/value array: id => captcha ID, input => captcha value
18 if ($captcha->isValid($_POST['foo'], $_POST)) {
19     // Validated!
20 }
```

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 *ZendForm Quick Start*.

CAPTCHA Adapters

The following adapters are shipped with Zend Framework by default.

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 *CAPTCHAs* 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.

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`.

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.

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.

Zend\Captcha\ReCaptcha

The `Zend\Captcha\ReCaptcha` adapter uses `Zend\Service\ReCaptcha\ReCaptcha` to generate and validate *CAPTCHAs*. 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.

`Zend\Code\Generator` provides facilities to generate arbitrary code using an object-oriented interface, both to create new code as well as to update existing code. While the current implementation is limited to generating *PHP* code, you can easily extend the base class in order to provide code generation for other tasks: JavaScript, configuration files, apache vhosts, etc.

Theory of Operation

In the most typical use case, you will simply instantiate a code generator class and either pass it the appropriate configuration or configure it after instantiation. To generate the code, you will simply echo the object or call its `generate()` method.

```

1 // Passing configuration to the constructor:
2 $file = new Zend\Code\Generator\FileGenerator(array(
3     'classes' => array(
4         new Zend\Code\Generator\ClassGenerator(
5             'World', // name
6             null,    // namespace
7             null,    // flags
8             null,    // extends
9             array(), // interfaces
10            array(), // properties
11            array(
12                new Zend\Code\Generator\MethodGenerator(
13                    'hello', // name
14                    array(), // parameters
15                    'public', // visibility
16                    'echo \'Hello world!\';' // body
17                ),
18            ),
19        ),
20    ),
21 );

```

```
22 // Render the generated file
23 echo $file->generate();
24
25 // or write it to a file:
26 file_put_contents('World.php', $file->generate());
27
28 // OR
29
30 // Configuring after instantiation
31 $method = new Zend\Code\Generator\MethodGenerator();
32 $method->setName('hello')
33     ->setBody('echo \'Hello world!\';');
34
35 $class = new Zend\Code\Generator\ClassGenerator();
36 $class->setName('World')
37     ->addMethodFromGenerator($method);
38
39 $file = new Zend\Code\Generator\FileGenerator();
40 $file->setClass($class);
41
42 // Render the generated file
43 echo $file->generate();
44
45 // or write it to a file:
46 file_put_contents('World.php', $file->generate());
47
```

Both of the above samples will render the same result:

```
1 <?php
2
3 class World
4 {
5
6     public function hello()
7     {
8         echo 'Hello world!';
9     }
10
11 }
```

Another common use case is to update existing code – for instance, to add a method to a class. In such a case, you must first inspect the existing code using reflection, and then add your new method. `Zend\Code\Generator` makes this trivially simple, by leveraging `ZendCodeReflection`.

As an example, let's say we've saved the above to the file `World.php`, and have already included it. We could then do the following:

```
1 $class = Zend\Code\Generator\ClassGenerator::fromReflection(
2     new Zend\Code\Reflection\ClassReflection('World')
3 );
4
5 $method = new Zend\Code\Generator\MethodGenerator();
6 $method->setName('mrMcFeeley')
7     ->setBody('echo \'Hello, Mr. McFeeley!\';');
8 $class->addMethodFromGenerator($method);
9
10 $file = new Zend\Code\Generator\FileGenerator();
```



```
11 $file->setClass($class);
12
13 // Render the generated file
14 echo $file->generate();
15
16 // Or, better yet, write it back to the original file:
17 file_put_contents('World.php', $file->generate());
```

The resulting class file will now look like this:

```
1 <?php
2
3 class World
4 {
5
6     public function hello()
7     {
8         echo 'Hello world!';
9     }
10
11     public function mrMcFeeley()
12     {
13         echo 'Hellow Mr. McFeeley!';
14     }
15
16 }
```


Abstract Classes and Interfaces

Zend\Code\Generator\GeneratorInterface

The base interface from which all CodeGenerator classes implement provides the minimal functionality necessary. It's *API* is as follows:

```
1 interface Zend\Code\Generator\GeneratorInterface
2 {
3     public function generate();
4 }
```

Zend\Code\Generator\AbstractGenerator

Zend\Code\Generator\AbstractGenerator implements Zend\Code\Generator\GeneratorInterface, and adds some properties for tracking whether content has changed as well as the amount of indentation that should appear before generated content. Its *API* is as follows:

```
1 abstract class Zend\Code\Generator\AbstractGenerator
2     implements Zend\Code\Generator\GeneratorInterface
3 {
4     public function __construct(Array|Traversable $options = array())
5     public function setOptions(Array $options)
6     public function setSourceContent($sourceContent)
7     public function getSourceContent()
8     public function setSourceDirty($isSourceDirty = true)
9     public function isSourceDirty()
10    public function setIndentation($indentation)
11    public function getIndentation()
12 }
```

The constructor passes the `$options` parameter to `setOptions()`.

Like most classes in Zend Framework, `setOptions()` compares an option key to existing setters in the class, and passes the value on to that method if found.

`setSourceContent()` and `getSourceContent()` are intended to either set the default content for the code being generated, or to replace said content once all generation tasks are complete.

Zend\Code\Generator\AbstractMemberGenerator

`Zend\Code\Generator\AbstractMemberGenerator` is a base class for generating class members – properties and methods – and provides accessors and mutators for establishing visibility; whether or not the member is abstract, static, or final; and the name of the member. Its *API* is as follows:

```

1  abstract class Zend\Code\Generator\AbstractMemberGenerator
2      extends Zend\Code\Generator\AbstractGenerator
3  {
4      public function setAbstract($isAbstract)
5      public function isAbstract()
6      public function setStatic($isStatic)
7      public function isStatic()
8      public function setVisibility($visibility)
9      public function getVisibility()
10     public function setName($name)
11     public function getName()
12 }
```

Concrete CodeGenerator Classes

Zend\Code\Generator\BodyGenerator

`Zend\Code\Generator\BodyGenerator` is intended for generating arbitrary procedural code to include within a file. As such, you simply set content for the object, and it will return that content when you invoke `generate()`.

The *API* of the class is as follows:

```

1  class Zend\Code\Generator\BodyGenerator extends Zend\Code\Generator\AbstractGenerator
2  {
3      public function setContent($content)
4      public function getContent()
5      public function generate()
6  }
```

Zend\Code\Generator\ClassGenerator

`Zend\Code\Generator\ClassGenerator` is intended for generating *PHP* classes. The basic functionality just generates the *PHP* class itself, as well as optionally the related *PHP* DocBlock. Classes may implement or inherit from other classes, and may be marked as abstract. Utilizing other code generator classes, you can also attach class constants, properties, and methods.

The *API* is as follows:

```

1 class Zend\Code\Generator\ClassGenerator extends Zend\Code\Generator\AbstractGenerator
2 {
3     public static function fromReflection(
4         Zend\Code\Reflection\ClassReflection $reflectionClass
5     )
6     public function addConstants(Array $properties)
7     public function addConstant($property)
8     public function getConstants()
9     public function getConstant($propertyName)
10    public function setDocblock(Zend\Code\Generator\DocBlockGenerator $docblock)
11    public function getDocblock()
12    public function setName($name)
13    public function getName()
14    public function setAbstract($isAbstract)
15    public function isAbstract()
16    public function setExtendedClass($extendedClass)
17    public function getExtendedClass()
18    public function setImplementedInterfaces(Array $implementedInterfaces)
19    public function getImplementedInterfaces()
20    public function addProperties(Array $properties)
21    public function addProperty($property)
22    public function getProperties()
23    public function getProperty($propertyName)
24    public function addMethods(Array $methods)
25    public function addMethod($method)
26    public function getMethods()
27    public function getMethod($methodName)
28    public function hasMethod($methodName)
29    public function isSourceDirty()
30    public function generate()
31 }

```

The `addProperty()` method accepts an array of information that may be used to generate a `Zend\Code\Generator\PropertyGenerator` instance – or simply an instance of `Zend\Code\Generator\PropertyGenerator`. Likewise, `addMethod()` accepts either an array of information for generating a `Zend\Code\Generator\MethodGenerator` instance or a concrete instance of that class.

Note that `setDocBlock()` expects an instance of `Zend\Code\Generator\DocBlockGenerator`.

Zend\Code\Generator\DocBlockGenerator

`Zend\Code\Generator\DocBlockGenerator` can be used to generate arbitrary *PHP* docblocks, including all the standard docblock features: short and long descriptions and annotation tags.

Annotation tags may be set using the `setTag()` and `setTags()` methods; these each take either an array describing the tag that may be passed to the `Zend\Code\Generator\DocBlock\Tag` constructor, or an instance of that class.

The *API* is as follows:

```

1 class Zend\Code\Generator\DocBlockGenerator extends
2     ↪ Zend\Code\Generator\AbstractGenerator
3 {
4     public static function fromReflection(
5         Zend\Code\Reflection\DocblockReflection $reflectionDocblock
6     )
7 }

```

```
6     public function setShortDescription($shortDescription)
7     public function getShortDescription()
8     public function setLongDescription($longDescription)
9     public function getLongDescription()
10    public function setTags(Array $tags)
11    public function setTag($tag)
12    public function getTags()
13    public function generate()
14 }
```

Zend\Code\Generator\DocBlock\Tag

Zend\Code\Generator\DocBlock\Tag is intended for creating arbitrary annotation tags for inclusion in *PHP* docblocks. Tags are expected to contain a name (the portion immediately following the ‘@’ symbol) and a description (everything following the tag name).

The class *API* is as follows:

```
1 class Zend\Code\Generator\DocBlock\Tag
2     extends Zend\Code\Generator\AbstractGenerator
3 {
4     public static function fromReflection(
5         Zend\Code\Reflection\DocBlock\Tag\TagInterface $reflectionTag
6     )
7     public function setName($name)
8     public function getName()
9     public function setDescription($description)
10    public function getDescription()
11    public function generate()
12 }
```

Zend\Code\Generator\DocBlock\Tag\ParamTag

Zend\Code\Generator\DocBlock\Tag\ParamTag is a specialized version of Zend\Code\Generator\DocBlock\Tag, and represents a method parameter. The tag name is therefore known (“param”), but due to the format of this annotation tag, additional information is required in order to generate it: the parameter name and data type it represents.

The class *API* is as follows:

```
1 class Zend\Code\Generator\DocBlock\Tag\ParamTag
2     extends Zend\Code\Generator\DocBlock\Tag
3 {
4     public static function fromReflection(
5         Zend\Code\Reflection\DocBlock\Tag\TagInterface $reflectionTagParam
6     )
7     public function setDatatype($datatype)
8     public function getDatatype()
9     public function setParamName($paramName)
10    public function getParamName()
11    public function generate()
12 }
```

Zend\Code\Generator\DocBlock\Tag\ReturnTag

Like the param docblock tag variant, `Zend\Code\Generator\DocBlock\Tag\ReturnTag` is an annotation tag variant for representing a method return value. In this case, the annotation tag name is known (“return”), but requires a return type.

The class *API* is as follows:

```

1 class Zend\Code\Generator\DocBlock\Tag\ParamTag
2     extends Zend\Code\Generator\DocBlock\Tag
3 {
4     public static function fromReflection(
5         Zend\Code\Reflection\DocBlock\Tag\TagInterface $reflectionTagReturn
6     )
7     public function setDatatype($datatype)
8     public function getDatatype()
9     public function generate()
10 }
```

Zend\Code\Generator\FileGenerator

`Zend\Code\Generator\FileGenerator` is used to generate the full contents of a file that will contain *PHP* code. The file may contain classes or arbitrary *PHP* code, as well as a file-level docblock if desired.

When adding classes to the file, you will need to pass either an array of information to pass to the `Zend\Code\Generator\ClassGenerator` constructor, or an instance of that class. Similarly, with docblocks, you will need to pass information for the `Zend\Code\Generator\DocBlockGenerator` constructor to consume or an instance of the class.

The *API* of the class is as follows:

```

1 class Zend\Code\Generator\FileGenerator extends Zend\Code\Generator\AbstractGenerator
2 {
3     public static function fromReflectedFilePath(
4         $filePath,
5         $usePreviousCodeGeneratorIfItExists = true,
6         $includeIfNotAlreadyIncluded = true)
7     public static function fromReflection(Zend\Code\Reflection\FileReflection
8 ↪ $reflectionFile)
9     public function setDocblock(Zend\Code\Generator\DocBlockGenerator $docblock)
10    public function getDocblock()
11    public function setRequiredFiles($requiredFiles)
12    public function getRequiredFiles()
13    public function setClasses(Array $classes)
14    public function getClass($name = null)
15    public function setClass($class)
16    public function setFilename($filename)
17    public function getFilename()
18    public function getClasses()
19    public function setBody($body)
20    public function getBody()
21    public function isSourceDirty()
22    public function generate()
23 }
```

Zend\Code\Generator\Member\ContainerGenerator

Zend\Code\Generator\Member\ContainerGenerator is used internally by Zend\Code\Generator\ClassGenerator to keep track of class members – properties and methods alike. These are indexed by name, using the concrete instances of the members as values.

The *API* of the class is as follows:

```
1 class Zend\Code\Generator\Member\ContainerGenerator extends ArrayObject
2 {
3     public function __construct($type = self::TYPE_PROPERTY)
4 }
```

Zend\Code\Generator\MethodGenerator

Zend\Code\Generator\MethodGenerator describes a class method, and can generate both the code and the docblock for the method. The visibility and status as static, abstract, or final may be indicated, per its parent class, Zend\Code\Generator\AbstractMemberGenerator. Finally, the parameters and return value for the method may be specified.

Parameters may be set using `setParameter()` or `setParameters()`. In each case, a parameter should either be an array of information to pass to the Zend\Code\Generator\ParameterGenerator constructor or an instance of that class.

The *API* of the class is as follows:

```
1 class Zend\Code\Generator\MethodGenerator
2     extends Zend\Code\Generator\AbstractMemberGenerator
3 {
4     public static function fromReflection(
5         Zend\Code\Reflection\MethodReflection $reflectionMethod
6     )
7     public function setDocblock(Zend\Code\Generator\DocBlockGenerator $docblock)
8     public function getDocblock()
9     public function setFinal($isFinal)
10    public function setParameters(Array $parameters)
11    public function setParameter($parameter)
12    public function getParameters()
13    public function setBody($body)
14    public function getBody()
15    public function generate()
16 }
```

Zend\Code\Generator\ParameterGenerator

Zend\Code\Generator\ParameterGenerator may be used to specify method parameters. Each parameter may have a position (if unspecified, the order in which they are registered with the method will be used), a default value, and a data type; a parameter name is required.

The *API* of the class is as follows:

```
1 class Zend\Code\Generator\ParameterGenerator extends
2     ↪ Zend\Code\Generator\AbstractGenerator
3 {
4     public static function fromReflection(
5         Zend\Code\Reflection\ParameterReflection $reflectionParameter
```



```

5     )
6     public function setType($type)
7     public function getType()
8     public function setName($name)
9     public function getName()
10    public function setDefaultValue($defaultValue)
11    public function getDefaultValue()
12    public function setPosition($position)
13    public function getPosition()
14    public function getPassedByReference()
15    public function setPassedByReference($passedByReference)
16    public function generate()
17 }

```

There are several problems that might occur when trying to set NULL, booleans or arrays as default values. For this the value holder object `Zend\Code\Generator\ParameterDefaultValueGenerator` can be used, for example:

```

1 $parameter = new Zend\Code\Generator\ParameterGenerator();
2 $parameter->setDefaultValue(
3     new Zend\Code\Generator\ValueGenerator("null")
4 );
5 $parameter->setDefaultValue(
6     new Zend\Code\Generator\ValueGenerator("array('foo', 'bar')")
7 );

```

Internally `setDefaultValue()` also converts the values which can't be expressed in *PHP* into the value holder.

Zend\Code\Generator\PropertyGenerator

`Zend\Code\Generator\PropertyGenerator` describes a class property, which may be either a constant or a variable. In each case, the property may have an optional default value associated with it. Additionally, the visibility of variable properties may be set, per the parent class, `Zend\Code\Generator\AbstractMemberGenerator`.

The *API* of the class is as follows:

```

1 class Zend\Code\Generator\PropertyGenerator
2     extends Zend\Code\Generator\AbstractMemberGenerator
3 {
4     public static function fromReflection(
5         Zend\Code\Reflection\PropertyReflection $reflectionProperty
6     )
7     public function setConst($const)
8     public function isConst()
9     public function setDefaultValue($defaultValue)
10    public function getDefaultValue()
11    public function generate()
12 }

```


Generating PHP classes

The following example generates an empty class with a class-level DocBlock.

```
1 use Zend\Code\Generator\ClassGenerator;
2 use Zend\Code\Generator\DocBlockGenerator;
3
4 $foo = new ClassGenerator();
5 $docblock = DocBlockGenerator::fromArray(array(
6     'shortDescription' => 'Sample generated class',
7     'longDescription' => 'This is a class generated with Zend\Code\Generator.',
8     'tags' => array(
9         array(
10             'name' => 'version',
11             'description' => '$Rev:$',
12         ),
13         array(
14             'name' => 'license',
15             'description' => 'New BSD',
16         ),
17     ),
18 ));
19 $foo->setName('Foo')
20     ->setDocblock($docblock);
21 echo $foo->generate();
```

The above code will result in the following:

```
1 /**
2  * Sample generated class
3  *
4  * This is a class generated with Zend\Code\Generator.
5  *
6  * @version $Rev:$
```

```
7  * @license New BSD
8  *
9  */
10 class Foo
11 {
12
13 }
```

Generating PHP classes with class properties

Building on the previous example, we now add properties to our generated class.

```
1  use Zend\Code\Generator\ClassGenerator;
2  use Zend\Code\Generator\DocBlockGenerator;
3  use Zend\Code\Generator\PropertyGenerator;
4
5  $foo      = new ClassGenerator();
6  $docblock = DocBlockGenerator::fromArray(array(
7      'shortDescription' => 'Sample generated class',
8      'longDescription'  => 'This is a class generated with Zend\Code\Generator.',
9      'tags'             => array(
10         array(
11             'name'          => 'version',
12             'description' => '$Rev:$',
13         ),
14         array(
15             'name'          => 'license',
16             'description' => 'New BSD',
17         ),
18     ),
19 ));
20 $foo->setName('Foo')
21     ->setDocblock($docblock)
22     ->addProperties(array(
23         array('_bar', 'baz',          PropertyGenerator::FLAG_PROTECTED),
24         array('baz', 'bat',          PropertyGenerator::FLAG_PUBLIC)
25     ))
26     ->addConstants(array(
27         array('bat', 'foobarbazbat')
28     ));
29 echo $foo->generate();
```

The above results in the following class definition:

```
1  /**
2   * Sample generated class
3   *
4   * This is a class generated with Zend\Code\Generator.
5   *
6   * @version $Rev:$
7   * @license New BSD
8   *
9   */
10 class Foo
11 {
12 }
```

```

13     protected $_bar = 'baz';
14
15     public $baz = 'bat';
16
17     const bat = 'foobarbazbat';
18
19 }

```

Generating PHP classes with class methods

Zend\Code\Generator\ClassGenerator allows you to attach methods with optional content to your classes. Methods may be attached as either arrays or concrete Zend\Code\Generator\MethodGenerator instances.

```

1  use Zend\Code\Generator\ClassGenerator;
2  use Zend\Code\Generator\DocBlockGenerator;
3  use Zend\Code\Generator\DocBlock\Tag;
4  use Zend\Code\Generator\MethodGenerator;
5  use Zend\Code\Generator\PropertyGenerator;
6
7  $foo      = new ClassGenerator();
8  $docblock = DocBlockGenerator::fromArray(array(
9      'shortDescription' => 'Sample generated class',
10     'longDescription'  => 'This is a class generated with Zend\Code\Generator.',
11     'tags'             => array(
12         array(
13             'name'          => 'version',
14             'description' => '$Rev:$',
15         ),
16         array(
17             'name'          => 'license',
18             'description' => 'New BSD',
19         ),
20     ),
21 ));
22 $foo->setName('Foo')
23     ->setDocblock($docblock)
24     ->addProperties(array(
25         array('_bar', 'baz', PropertyGenerator::FLAG_PROTECTED),
26         array('baz', 'bat', PropertyGenerator::FLAG_PUBLIC)
27     ))
28     ->addConstants(array(
29         array('bat', 'foobarbazbat', PropertyGenerator::FLAG_CONSTANT)
30     ))
31     ->addMethods(array(
32         // Method passed as array
33         MethodGenerator::fromArray(array(
34             'name'          => 'setBar',
35             'parameters' => array('bar'),
36             'body'          => '$this->_bar = $bar;' . "\n" . 'return $this;',
37             'docblock'      => DocBlockGenerator::fromArray(array(
38                 'shortDescription' => 'Set the bar property',
39                 'longDescription'  => null,
40                 'tags'             => array(
41                     new Tag\ParamTag(array(
42                         'paramName' => 'bar',
43                         'datatype'  => 'string'

```

```

44        )),
45         new Tag\ReturnTag(array(
46             'datatype' => 'string',
47         )),
48     ),
49 ),
50 ),
51 // Method passed as concrete instance
52 new MethodGenerator(
53     'getBar',
54     array(),
55     MethodGenerator::FLAG_PUBLIC,
56     'return $this->_bar;',
57     DocBlockGenerator::fromArray(array(
58         'shortDescription' => 'Retrieve the bar property',
59         'longDescription' => null,
60         'tags' => array(
61             new Tag\ReturnTag(array(
62                 'datatype' => 'string|null',
63             )),
64         ),
65     ))
66 ),
67 ));
68
69 echo $foo->generate();

```

The above generates the following output:

```

1  /**
2   * Sample generated class
3   *
4   * This is a class generated with Zend\Code\Generator.
5   *
6   * @version $Rev:$
7   * @license New BSD
8   */
9  class Foo
10 {
11
12     protected $_bar = 'baz';
13
14     public $baz = 'bat';
15
16     const bat = 'foobarbazbat';
17
18     /**
19      * Set the bar property
20      *
21      * @param string bar
22      * @return string
23      */
24     public function setBar($bar)
25     {
26         $this->_bar = $bar;
27         return $this;
28     }
29

```

```

30  /**
31   * Retrieve the bar property
32   *
33   * @return string|null
34   */
35  public function getBar()
36  {
37      return $this->_bar;
38  }
39
40  }

```

Generating PHP files

Zend\Code\Generator\FileGenerator can be used to generate the contents of a *PHP* file. You can include classes as well as arbitrary content body. When attaching classes, you should attach either concrete Zend\Code\Generator\ClassGenerator instances or an array defining the class.

In the example below, we will assume you've defined `$foo` per one of the class definitions in a previous example.

```

1  use Zend\Code\Generator\DocBlockGenerator;
2  use Zend\Code\Generator\FileGenerator;
3
4  $file = FileGenerator::fromArray(array(
5      'classes' => array($foo),
6      'docblock' => DocBlockGenerator::fromArray(array(
7          'shortDescription' => 'Foo class file',
8          'longDescription' => null,
9          'tags' => array(
10         array(
11             'name' => 'license',
12             'description' => 'New BSD',
13         ),
14     ),
15 ),
16 'body' => 'define(\'APPLICATION_ENV\', \'testing\');',
17 ));

```

Calling `generate()` will generate the code – but not write it to a file. You will need to capture the contents and write them to a file yourself. As an example:

```

1  $code = $file->generate();
2  file_put_contents('Foo.php', $code);

```

The above will generate the following file:

```

1  <?php
2  /**
3   * Foo class file
4   *
5   * @license New BSD
6   */
7
8  /**
9   * Sample generated class

```

```
10  *
11  * This is a class generated with Zend\Code\Generator.
12  *
13  * @version $Rev:$
14  * @license New BSD
15  */
16  class Foo
17  {
18
19      protected $_bar = 'baz';
20
21      public $baz = 'bat';
22
23      const bat = 'foobarbazbat';
24
25      /**
26       * Set the bar property
27       *
28       * @param string bar
29       * @return string
30       */
31      public function setBar($bar)
32      {
33          $this->_bar = $bar;
34          return $this;
35      }
36
37      /**
38       * Retrieve the bar property
39       *
40       * @return string|null
41       */
42      public function getBar()
43      {
44          return $this->_bar;
45      }
46
47  }
48
49  define('APPLICATION_ENV', 'testing');
```

Add code to existing PHP files and classes

Seeding PHP file code generation via reflection

You can add *PHP* code to an existing *PHP* file using the code generator. To do so, you need to first do reflection on it. The static method `fromReflectedFileName()` allows you to do this.

```
1  $generator = Zend\Code\Generator\FileGenerator::fromReflectedFileName($path);
2  $generator->setBody("\$foo->bar();");
3  file_put_contents($path, $generator->generate());
```


Seeding PHP class generation via reflection

You may add code to an existing class. To do so, first use the static `fromReflection()` method to map the class into a generator object. From there, you may add additional properties or methods, and then regenerate the class.

```

1  use Zend\Code\Generator\ClassGenerator;
2  use Zend\Code\Generator\DocBlockGenerator;
3  use Zend\Code\Generator\DocBlock\Tag;
4  use Zend\Code\Generator\MethodGenerator;
5  use Zend\Code\Reflection\ClassReflection;
6
7  $generator = ClassGenerator::fromReflection(
8      new ClassReflection($class)
9  );
10 $generator->addMethod(
11     'setBaz',
12     array('baz'),
13     MethodGenerator::FLAG_PUBLIC,
14     '$this->_baz = $baz;' . "\n" . 'return $this;',
15     DocBlockGenerator::fromArray(array(
16         'shortDescription' => 'Set the baz property',
17         'longDescription'  => null,
18         'tags'             => array(
19             new Tag\ParamTag(array(
20                 'paramName' => 'baz',
21                 'datatype'  => 'string'
22             )),
23             new Tag\ReturnTag(array(
24                 'datatype'  => 'string',
25             )),
26         ),
27     ))
28 );
29 $code = $generator->generate();

```

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.

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:

```
1 // An array of configuration data is given
2 $configArray = array(
3     'webhost' => 'www.example.com',
4     'database' => array(
5         'adapter' => 'pdo_mysql',
6         'params' => array(
7             'host' => 'db.example.com',
8             'username' => 'dbuser',
9             'password' => 'secret',
10            'dbname' => 'mydatabase'
11        )
12    )
13 );
14
15 // Create the object-oriented wrapper using the configuration data
16 $config = new Zend\Config\Config($configArray);
17
18 // Print a configuration datum (results in 'www.example.com')
19 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:

```
1 $host = $config->database->get('host', 'localhost');
```

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:

```
1 // config.php
2 return array(
3     'webhost' => 'www.example.com',
4     'database' => array(
5         'adapter' => 'pdo_mysql',
6         'params' => array(
7             'host' => 'db.example.com',
8             'username' => 'dbuser',
9             'password' => 'secret',
10            'dbname' => 'mydatabase'
11         )
12     )
13 );
```

```
1 // Consumes the configuration array
2 $config = new Zend\Config\Config(include 'config.php');
3
4 // Print a configuration datum (results in 'www.example.com')
5 echo $config->webhost;
```

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.

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
- Zend\Config\Reader\JavaProperties

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.
-

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:

```
1 $reader = new Zend\Config\Reader\Ini();
2 $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:

```
1 webhost           = 'www.example.com'
2 database.adapter  = 'pdo_mysql'
3 database.params.host = 'db.example.com'
4 database.params.username = 'dbuser'
5 database.params.password = 'secret'
6 database.params.dbname = 'dbproduction'
```

We can use the `Zend\Config\Reader\Ini` to read this INI file:

```
1 $reader = new Zend\Config\Reader\Ini();
2 $data   = $reader->fromFile('/path/to/config.ini');
3
4 echo $data['webhost']; // prints "www.example.com"
5 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:

```
1 database.adapter      = 'pdo_mysql'
2 database.params.host  = 'db.example.com'
3 database.params.username = 'dbuser'
4 database.params.password = 'secret'
5 database.params.dbname = 'dbproduction'
```

We can include this configuration in another INI file, for instance:

```
1 webhost = 'www.example.com'
2 @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:

```
1 adapter      = 'pdo_mysql'
2 params.host  = 'db.example.com'
3 params.username = 'dbuser'
4 params.password = 'secret'
5 params.dbname = 'dbproduction'
```

And assign the `@include` as subelement of the database value:


```

1 webhost          = 'www.example.com'
2 database.@include = 'database.ini'

```

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:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <config>
3     <webhost>www.example.com</webhost>
4     <database>
5         <adapter value="pdo_mysql"/>
6         <params>
7             <host value="db.example.com"/>
8             <username value="dbuser"/>
9             <password value="secret"/>
10            <dbname value="dbproduction"/>
11        </params>
12    </database>
13 </config>

```

We can use the Zend\Config\Reader\Xml to read this XML file:

```

1 $reader = new Zend\Config\Reader\Xml();
2 $data   = $reader->fromFile('/path/to/config.xml');
3
4 echo $data['webhost']; // prints "www.example.com"
5 echo $data['database']['params']['dbname']['value']; // 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:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <config>
3     <database>
4         <adapter>pdo_mysql</adapter>
5         <params>
6             <host>db.example.com</host>
7             <username>dbuser</username>
8             <password>secret</password>
9             <dbname>dbproduction</dbname>
10        </params>
11    </database>
12 </config>

```

We can include this configuration in another XML file, for instance:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <config xmlns:xi="http://www.w3.org/2001/XInclude">
3     <webhost>www.example.com</webhost>
4     <xi:include href="database.xml"/>
5 </config>
```

The syntax to include an XML file in a specific element is `<xi:include href="file-to-include.xml"/>`

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:

```
1 {
2     "webhost" : "www.example.com",
3     "database" : {
4         "adapter" : "pdo_mysql",
5         "params" : {
6             "host" : "db.example.com",
7             "username" : "dbuser",
8             "password" : "secret",
9             "dbname" : "dbproduction"
10        }
11    }
12 }
```

We can use the Zend\Config\Reader\Json to read this JSON file:

```
1 $reader = new Zend\Config\Reader\Json();
2 $data   = $reader->fromFile('/path/to/config.json');
3
4 echo $data['webhost']; // prints "www.example.com"
5 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:

```
1 {
2     "database" : {
3         "adapter" : "pdo_mysql",
4         "params" : {
5             "host" : "db.example.com",
6             "username" : "dbuser",
7             "password" : "secret",
8             "dbname" : "dbproduction"
9         }
10    }
11 }
```

We can include this configuration in another JSON file, for instance:

```
1 {
2     "webhost" : "www.example.com",
3     "@include" : "database.json"
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:

```
1 webhost: www.example.com
2 database:
3     adapter: pdo_mysql
4     params:
5         host:      db.example.com
6         username: dbuser
7         password: secret
8         dbname:   dbproduction
```

We can use the Zend\Config\Reader\Yaml to read this *YAML* file:

```
1 $reader = new Zend\Config\Reader\Yaml();
2 $data   = $reader->fromFile('/path/to/config.yaml');
3
4 echo $data['webhost']; // prints "www.example.com"
5 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:

```
1 // include the Spyc library
2 require_once ('path/to/spyc.php');
3
4 $reader = new Zend\Config\Reader\Yaml(array('Spyc', 'YAMLLoadString'));
5 $data   = $reader->fromFile('/path/to/config.yaml');
6
7 echo $data['webhost']; // prints "www.example.com"
8 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:

```
1 database:
2     adapter: pdo_mysql
3     params:
4         host:      db.example.com
```

```
5     username: dbuser
6     password: secret
7     dbname:  dbproduction
```

We can include this configuration in another YAML file, for instance:

```
1 webhost:  www.example.com
2 @include: database.yaml
```

Zend\Config\Reader\JavaProperties

`Zend\Config\Reader\JavaProperties` enables developers to read configuration data in a familiar *JavaProperties* format and read them in the application by using an array syntax.

The following example illustrates a basic use of `Zend\Config\Reader\JavaProperties` for loading configuration data from an *JavaProperties* file. Suppose we have the following *JavaProperties* configuration file:

```
1  #comment
2  !comment
3  webhost:www.example.com
4  database.adapter:pdo_mysql
5  database.params.host:db.example.com
6  database.params.username:dbuser
7  database.params.password:secret
8  database.params.dbname:dbproduction
```

We can use the `Zend\Config\Reader\JavaProperties` to read this *JavaProperties* file:

```
1 $reader = new Zend\Config\Reader\JavaProperties();
2 $data   = $reader->fromFile('/path/to/config.properties');
3
4 echo $data['webhost']; // prints "www.example.com"
5 echo $data['database.params.dbname']; // prints "dbproduction"
```

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

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:

```
1 // Create the config object
2 $config = new Zend\Config\Config(array(), true);
3 $config->production = array();
4
5 $config->production->webhost = 'www.example.com';
6 $config->production->database = array();
7 $config->production->database->params = array();
8 $config->production->database->params->host = 'localhost';
9 $config->production->database->params->username = 'production';
10 $config->production->database->params->password = 'secret';
11 $config->production->database->params->dbname = 'dbproduction';
12
13 $writer = new Zend\Config\Writer\Ini();
14 echo $writer->toString($config);
```

The result of this code is an INI string contains the following values:

```
1 [production]
2 webhost = "www.example.com"
3 database.params.host = "localhost"
4 database.params.username = "production"
5 database.params.password = "secret"
6 database.params.dbname = "dbproduction"
```

You can use the method `toFile()` to store the INI data in a file.

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\Xml

This example illustrates the basic use of `Zend\Config\Writer\Xml` to create a new config file:

```
1 // Create the config object
2 $config = new Zend\Config\Config(array(), true);
3 $config->production = array();
4
5 $config->production->webhost = 'www.example.com';
6 $config->production->database = array();
7 $config->production->database->params = array();
8 $config->production->database->params->host = 'localhost';
9 $config->production->database->params->username = 'production';
10 $config->production->database->params->password = 'secret';
11 $config->production->database->params->dbname = 'dbproduction';
12
13 $writer = new Zend\Config\Writer\Xml();
14 echo $writer->toString($config);
```

The result of this code is an XML string contains the following data:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <zend-config>
```

```

3      <production>
4          <webhost>www.example.com</webhost>
5          <database>
6              <params>
7                  <host>localhost</host>
8                  <username>production</username>
9                  <password>secret</password>
10                 <dbname>dbproduction</dbname>
11             </params>
12         </database>
13     </production>
14 </zend-config>

```

You can use the method `toFile()` to store the XML data in a file.

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:

```

1  // Create the config object
2  $config = new Zend\Config\Config(array(), true);
3  $config->production = array();
4
5  $config->production->webhost = 'www.example.com';
6  $config->production->database = array();
7  $config->production->database->params = array();
8  $config->production->database->params->host = 'localhost';
9  $config->production->database->params->username = 'production';
10 $config->production->database->params->password = 'secret';
11 $config->production->database->params->dbname = 'dbproduction';
12
13 $writer = new Zend\Config\Writer\PhpArray();
14 echo $writer->toString($config);

```

The result of this code is a PHP script that returns an array as follow:

```

1  <?php
2  return array (
3      'production' =>
4          array (
5              'webhost' => 'www.example.com',
6              'database' =>
7                  array (
8                      'params' =>
9                          array (
10                             'host' => 'localhost',
11                             'username' => 'production',
12                             'password' => 'secret',
13                             'dbname' => 'dbproduction',

```

```
14         ),  
15     ),  
16 ),  
17 );
```

You can use the method `toFile()` to store the PHP script in a file.

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:

```
1 // Create the config object  
2 $config = new Zend\Config\Config(array(), true);  
3 $config->production = array();  
4  
5 $config->production->webhost = 'www.example.com';  
6 $config->production->database = array();  
7 $config->production->database->params = array();  
8 $config->production->database->params->host = 'localhost';  
9 $config->production->database->params->username = 'production';  
10 $config->production->database->params->password = 'secret';  
11 $config->production->database->params->dbname = 'dbproduction';  
12  
13 $writer = new Zend\Config\Writer\Json();  
14 echo $writer->toString($config);
```

The result of this code is a JSON string contains the following values:

```
1 { "webhost" : "www.example.com",  
2   "database" : {  
3     "params" : {  
4       "host" : "localhost",  
5       "username" : "production",  
6       "password" : "secret",  
7       "dbname" : "dbproduction"  
8     }  
9   }  
10 }
```

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.

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:

```

1 // Create the config object
2 $config = new Zend\Config\Config(array(), true);
3 $config->production = array();
4
5 $config->production->webhost = 'www.example.com';
6 $config->production->database = array();
7 $config->production->database->params = array();
8 $config->production->database->params->host = 'localhost';
9 $config->production->database->params->username = 'production';
10 $config->production->database->params->password = 'secret';
11 $config->production->database->params->dbname = 'dbproduction';
12
13 $writer = new Zend\Config\Writer\Yaml();
14 echo $writer->toString($config);

```

The result of this code is a YAML string contains the following values:

```

1 webhost: www.example.com
2 database:
3     params:
4         host:      localhost
5         username: production
6         password: secret
7         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:

```

1 // include the Spyc library
2 require_once ('path/to/spyc.php');
3
4 $writer = new Zend\Config\Writer\Yaml(array('Spyc', 'YAMLDump'));
5 echo $writer->toString($config);

```

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.

Zend\Config\Processor\Constant

Using Zend\Config\Processor\Constant

This example illustrates the basic use of Zend\Config\Processor\Constant:

```
1 define ('TEST_CONST', 'bar');
2 // set true to Zend\Config\Config to allow modifications
3 $config = new Zend\Config\Config(array('foo' => 'TEST_CONST'), true);
4 $processor = new Zend\Config\Processor\Constant();
5
6 echo $config->foo . ',';
7 $processor->process($config);
8 echo $config->foo;
```

This example returns the output: TEST_CONST, bar..

Zend\Config\Processor\Filter

Using Zend\Config\Processor\Filter

This example illustrates the basic use of Zend\Config\Processor\Filter:

```
1 use Zend\Filter\StringToUpper;
2 use Zend\Config\Processor\Filter as FilterProcessor;
3 use Zend\Config\Config;
4
5 $config = new Config(array ('foo' => 'bar'), true);
6 $upper = new StringToUpper();
7
8 $upperProcessor = new FilterProcessor($upper);
9
10 echo $config->foo . ', ';
11 $upperProcessor->process($config);
12 echo $config->foo;
```

This example returns the output: bar, BAR.

Zend\Config\Processor\Queue

Using Zend\Config\Processor\Queue

This example illustrates the basic use of Zend\Config\Processor\Queue:

```
1 use Zend\Filter\StringToLower;
2 use Zend\Filter\StringToUpper;
3 use Zend\Config\Processor\Filter as FilterProcessor;
4 use Zend\Config\Processor\Queue;
5 use Zend\Config\Config;
6
7 $config = new Config(array ('foo' => 'bar'), true);
8 $upper = new StringToUpper();
9 $lower = new StringToLower();
10
11 $lowerProcessor = new FilterProcessor($lower);
12 $upperProcessor = new FilterProcessor($upper);
13
14 $queue = new Queue();
15 $queue->insert($upperProcessor);
16 $queue->insert($lowerProcessor);
17 $queue->process($config);
18
19 echo $config->foo;
```

This example returns the output: bar. The filters in the queue are applied with a *FIFO* logic (First In, First Out).

Zend\Config\Processor\Token

Using Zend\Config\Processor\Token

This example illustrates the basic use of Zend\Config\Processor\Token:

```

1 // set the Config to true to allow modifications
2 $config = new Config(array('foo' => 'Value is TOKEN'), true);
3 $processor = new TokenProcessor();
4
5 $processor->addToken('TOKEN', 'bar');
6 echo $config->foo . ',';
7 $processor->process($config);
8 echo $config->foo;
```

This example returns the output: Value is TOKEN,Value is bar.

Zend\Config\Processor\Translator

Using Zend\Config\Processor\Translator

This example illustrates the basic use of Zend\Config\Processor\Translator:

```

1 use Zend\Config\Config;
2 use Zend\Config\Processor\Translator as TranslatorProcessor;
3 use Zend\I18n\Translator\Translator;
4
5 $config = new Config(array('animal' => 'dog'), true);
6
7 /*
8  * The following mapping would exist for the translation
9  * loader you provide to the translator instance
10  * $italian = array(
11  *     'dog' => 'cane'
12  * );
13  */
14
15 $translator = new Translator();
16 // ... configure the translator ...
17 $processor = new TranslatorProcessor($translator);
18
19 echo "English: {$config->animal}, ";
20 $processor->process($config);
21 echo "Italian: {$config->animal}";
```

This example returns the output: English: dog, Italian: cane.

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.

Loading configuration file

The next example illustrates how to load a single configuration file

```
1 //Load a php file as array
2 $config = Zend\Config\Factory::fromFile(__DIR__.'/config/my.config.php');
3
4 //Load a xml file as Config object
5 $config = Zend\Config\Factory::fromFile(__DIR__.'/config/my.config.xml', true);
```

For merging multiple configuration files

Storing configuration file

Sometimes you want to store the configuration to a file. Also this is really easy to do

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.

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:

```
1 array(  
2     'router' => array(  
3         'routes' => array(  
4             // HTTP routes are defined here  
5         )  
6     ),  
7  
8     'console' => array(  
9         'router' => array(  
10            'routes' => array(  
11                // Console routes go here  
12            )  
13        )  
14    ),  
15 )
```

Let’s create our console route and point it to `Application\Controller\IndexController::resetpasswordAction()`

```
1 // we could define routes for Application\Controller\IndexController in Application_  
2 ↪module config file  
3 // which is usually located at modules/application/config/module.config.php  
4 array(  
5     'console' => array(  
6         'router' => array(  
7             'routes' => array(  
8                 'user-reset-password' => array(  
9                     'options' => array(  
10                        'route' => 'user resetpassword [--verbose|-v] <userEmail>',  
11                        'defaults' => array(  
12                            'controller' => 'Application\Controller\Index',  
13                            'action' => 'resetpassword'  
14                        )  
15                    )  
16                )  
17            )  
18        )  
19    )  
20 )
```

```

18     )
19 )
    
```

See also:

To learn more about console routes and how to use them, please read this chapter: *Console routes and routing*

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`:

```

1  <?php
2  namespace Application\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6  use Zend\Console\Request as ConsoleRequest;
7  use Zend\Math\Rand;
8
9  class IndexController extends AbstractActionController
10 {
11     public function indexAction()
12     {
13         return new ViewModel(); // display standard index page
14     }
15
16     public function resetpasswordAction()
17     {
18         $request = $this->getRequest();
19
20         // Make sure that we are running in a console and the user has not tricked our
21         // application into running this action from a public web server.
22         if (!$request instanceof ConsoleRequest) {
23             throw new \RuntimeException('You can only use this action from a console!
24         ↪');
25         }
26
27         // Get user email from console and check if the user used --verbose or -v flag
28         $userEmail = $request->getParam('userEmail');
29         $verbose = $request->getParam('verbose') || $request->getParam('v');
30
31         // reset new password
32         $newPassword = Rand::getString(16);
33
34         // Fetch the user and change his password, then email him ...
35         // [...]
36
37         if (!$verbose) {
38             return "Done! $userEmail has received an email with his new password.\n";
39         } else {
40             return "Done! New password for user $userEmail is '$newPassword'. It has
41 ↪also been emailed to him. \n";
    
```

```
41     }  
42 }
```

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](#)*

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\Module` to provide usage info:

```
1 <?php  
2  
3 namespace Application;  
4  
5 use Zend\ModuleManager\Feature\AutoloaderProviderInterface;  
6 use Zend\ModuleManager\Feature\ConfigProviderInterface;  
7 use Zend\ModuleManager\Feature\ConsoleUsageProviderInterface;  
8 use Zend\Console\Adapter\AdapterInterface as Console;  
9  
10 class Module implements  
11     AutoloaderProviderInterface,  
12     ConfigProviderInterface,  
13     ConsoleUsageProviderInterface // <- our module implement this feature and  
14     ↪ provides console usage info  
15 {  
16     public function getConfig()  
17     {  
18         // [...]  
19     }  
20  
21     public function getAutoloaderConfig()  
22     {  
23         // [...]  
24     }  
25  
26     public function getConsoleUsage(Console $console)  
27     {  
28         return array(  
29
```

```
28         // Describe available commands
29         'user resetpassword [--verbose|-v] EMAIL'    => 'Reset password for a user
↪',
30
31         // Describe expected parameters
32         array( 'EMAIL',          'Email of the user for a password reset' ),
33         array( '--verbose|-v',    '(optional) turn on verbose mode'      ),
34     );
35 }
36 }
```

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](#)

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*.

Router configuration

All Console Routes are automatically read from the following configuration location:

```
1 // This can sit inside of modules/Application/config/module.config.php or any other
2 ↪module's config.
3 array(
4     'router' => array(
5         'routes' => array(
6             // HTTP routes are here
7         )
8     ),
9     'console' => array(
10        'router' => array(
11            'routes' => array(
12                // Console routes go here
13            )
14        )
15    )
16 )
```

```
15     ),
16 )
```

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:

```
1 // inside config.console.router.routes:
2 // [...]
3 'my-first-route' => array(
4     'type'      => 'simple',          // <- simple route is created by default, we can_
5     ↪ skip that
6     'options' => array(
7         'route'      => 'foo bar',
8         'defaults' => array(
9             'controller' => 'Application\Controller\Index',
10            'action'      => 'password'
11        )
12    )
13 )
```

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*.

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]`)

Literal parameters

These parameters are expected to appear on the command line exactly the way they are spelled in the route. For example:

```
1 'show-users' => array(
2     'options' => array(
3         'route'      => 'show users',
4         'defaults' => array(
5             'controller' => 'Application\Controller\Users',
6             'action'      => 'show'
7         )
8     )
9 )
```


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:

```
1 'show-users' => array(
2     'options' => array(
3         'route' => 'show [all] users',
4         'defaults' => array(
5             'controller' => 'Application\Controller\Users',
6             'action' => 'show'
7         )
8     )
9 )
```

Now this route will match for both of these commands:

```
> zf show users
> zf show all users
```

We can also provide **parameter alternative**:

```
1 'show-users' => array(
2     'options' => array(
3         'route' => 'show [all|deleted|locked|admin] users',
4         'defaults' => array(
5             'controller' => 'Application\Controller\Users',
6             'action' => 'show'
7         )
8     )
9 )
```

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`

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. They 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**

```
1 'check-users' => array(  
2     'options' => array(  
3         'route'      => 'check users [--verbose] [--fast] [--thorough]',  
4         'defaults' => array(  
5             'controller' => 'Application\Controller\Users',  
6             'action'      => 'check'  
7         )  
8     )  
9 )
```

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:

```
1 'check-users' => array(  
2     'options' => array(  
3         'route'      => 'check users (--suspicious|--expired) [--verbose] [--fast] [--  
4         thorough]',  
5         'defaults' => array(  
6             'controller' => 'Application\Controller\Users',  
7             'action'      => 'check'  
8         )  
9     )  
10 )
```

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:

```
1 'check-users' => array(  
2     'options' => array(  
3         'route'      => 'check users [--verbose|-v] [--fast|-f] [--thorough|-t]',  
4         'defaults' => array(  
5             'controller' => 'Application\Controller\Users',  
6             'action'      => 'check'  
7         )  
8     )  
9 )
```

Now we can use short versions of our flags:

```
> zf check users -f  
> zf check users -v --thorough  
> zf check users -t -f -v
```

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:

```

1 'delete-user' => array(
2     'options' => array(
3         'route' => 'delete user <userEmail>',
4         'defaults' => array(
5             'controller' => 'Application\Controller\Users',
6             'action' => 'delete'
7         )
8     )
9 )

```

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:

```

1 'delete-user' => array(
2     'options' => array(
3         'route' => 'delete user [<userEmail>]',
4         'defaults' => array(
5             'controller' => 'Application\Controller\Users',
6             'action' => 'delete'
7         )
8     )
9 )

```

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:

```

1 'create-user' => array(
2     'options' => array(
3         'route' => 'create user <firstName> <lastName> <email> <position>',
4         'defaults' => array(
5             'controller' => 'Application\Controller\Users',
6             'action' => 'create'
7         )
8     )
9 )

```

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 "Johndan Tom" Bravo john@acme.org "Head of the Entertainment
↳ Department"

```

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:

```
1 'find-user' => array(  
2     'options' => array(  
3         'route' => 'find user [--id=] [--firstName=] [--lastName=] [--email=] [--  
4         ↪position=] ',  
5         'defaults' => array(  
6             'controller' => 'Application\Controller\Users',  
7             'action' => 'find'  
8         )  
9     )  
10 )
```

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=Johnny --lastName=Bravo  
> zf find user --lastName Bravo --firstName Johnny  
> 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**:

```
1 'rename-user' => array(  
2     'options' => array(  
3         'route' => 'rename user --id= [--firstName=] [--lastName=]',  
4         'defaults' => array(  
5             'controller' => 'Application\Controller\Users',  
6             'action' => 'rename'  
7         )  
8     )  
9 )
```

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
```

Catchall route

This special route will catch all console requests, regardless of the parameters provided.

```
1 'default-route' => array(  
2     'type'      => 'catchall',  
3     'options'   => array(  
4         'route'   => '',  
5         'defaults' => array(  
6             'controller' => 'Application\Controller\Index',  
7             'action'     => 'consoledefault'  
8         )  
9     )  
10 )
```

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.

Console routes cheat-sheet

Param type	Example route definition	Explanation
Literal params		
Literal	<code>foo bar</code>	“foo” followed by “bar”
Literal alternative	<code>foo (bar baz)</code>	“foo” followed by “bar” or “baz”
Literal, optional	<code>foo [bar]</code>	“foo”, optional “bar”
Literal, optional alternative	<code>foo [bar baz]</code>	“foo”, optional “bar” or “baz”
Flags		
Flag long	<code>foo --bar</code>	“foo” as first parameter, “-bar” flag before or after
Flag long, optional	<code>foo [--bar]</code>	“foo” as first parameter, optional “-bar” flag before or after
Flag long, optional, alternative	<code>foo [--bar --baz]</code>	“foo” as first parameter, optional “-bar” or “-baz”, before or after
Flag short	<code>foo -b</code>	“foo” as first parameter, “-b” flag before or after
Flag short, optional	<code>foo [-b]</code>	“foo” as first parameter, optional “-b” flag before or after
Flag short, optional, alternative	<code>foo [-b -z]</code>	“foo” as first parameter, optional “-b” or “-z”, before or after
Flag long/short alternative	<code>foo [--bar -b]</code>	“foo” as first parameter, optional “-bar” or “-b” before or after
Value parameters		
Value positional param	<code>foo <bar></code>	“foo” followed by any text (stored as “bar” param)
Value positional param, optional	<code>foo [<bar>]</code>	“foo”, optionally followed by any text (stored as “bar” param)
Value Flag	<code>foo --bar=</code>	“foo” as first parameter, “-bar” with a value, before or after
Value Flag, optional	<code>foo [--bar=]</code>	“foo” as first parameter, optionally “-bar” with a value, before or after
Parameter groups		
Literal params group	<code>foo (bar baz) :myParam</code>	“foo” followed by “bar” or “baz” (stored as “myParam” param)
Literal optional params group	<code>foo [bar baz] :myParam</code>	“foo” followed by optional “bar” or “baz” (stored as “myParam” param)
Long flags group	<code>foo (--bar --baz) :myParam</code>	“foo”, “bar” or “baz” flag before or after (stored as “myParam” param)
Long optional flags group	<code>foo [--bar --baz] :myParam</code>	“foo”, optional “bar” or “baz” flag before or after (as “myParam” param)
Short flags group	<code>foo (-b -z) :myParam</code>	“foo”, “-b” or “-z” flag before or after (stored as “myParam” param)
Short optional flags group	<code>foo [-b -z] :myParam</code>	“foo”, optional “-b” or “-z” flag before or after (stored as “myParam” param)

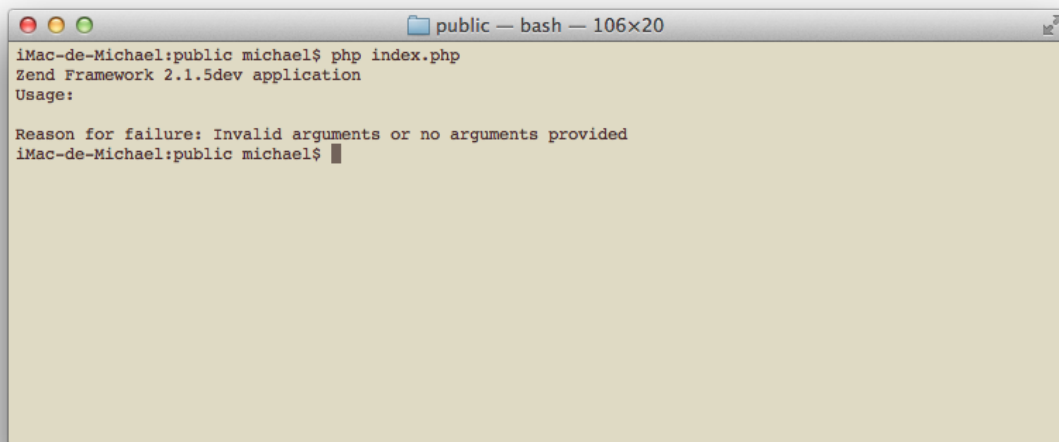
Console-aware modules

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.

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:



```
iMac-de-Michael:public michael$ php index.php
Zend Framework 2.1.5dev application
Usage:

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$
```

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.

```

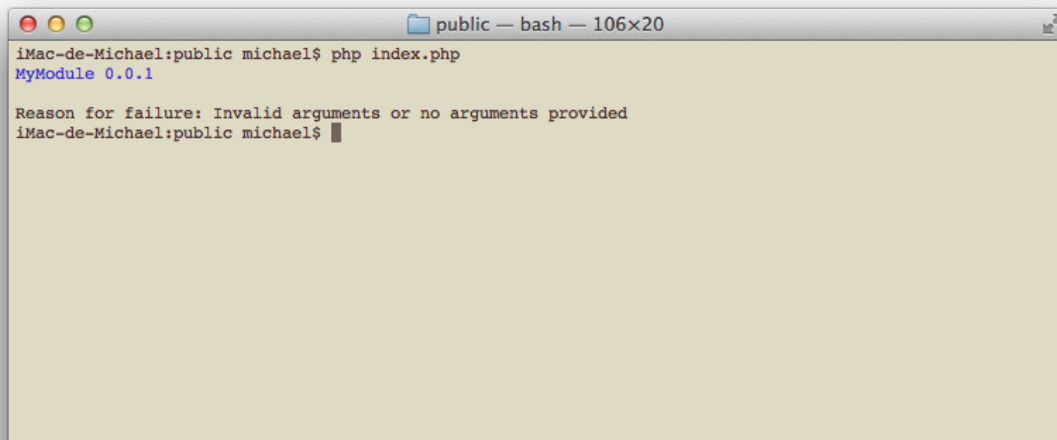
1 // modules/Application/Module.php
2 <?php
3 namespace Application;
4
5 use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
6 use Zend\Console\Adapter\AdapterInterface as Console;
7
8 class Module implements ConsoleBannerProviderInterface
9 {
10     /**
11      * This method is defined in ConsoleBannerProviderInterface
12      */
13     public function getConsoleBanner(Console $console)
14     {
15         return 'MyModule 0.0.1';
16     }
17 }

```

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.



```

iMac-de-Michael:public michael$ php index.php
MyModule 0.0.1

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$

```

Let's create and load second module that provides a banner.

```

1 <?php
2 // config/application.config.php
3 return array(
4     'modules' => array(
5         'Application',
6         'User',      // < load user module in modules/User

```

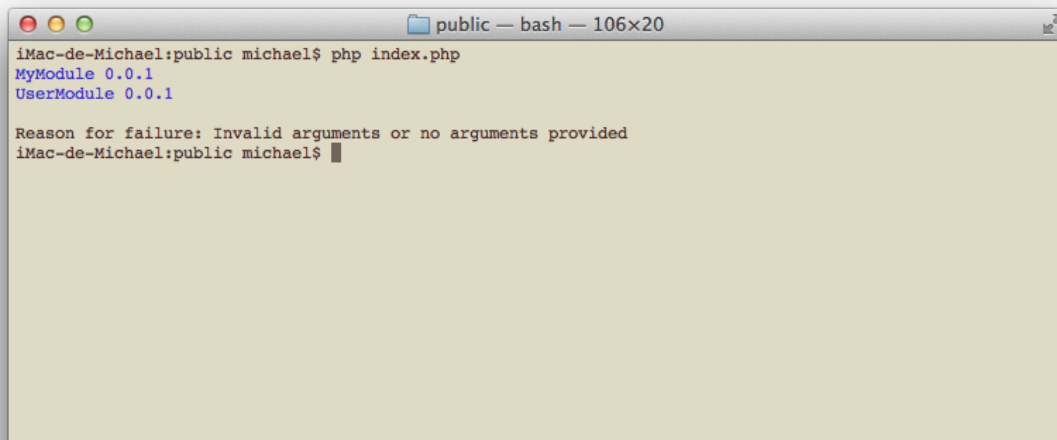

),

User module will add-on a short info about itself:

```

1 // modules/User/Module.php
2 <?php
3 namespace User;
4
5 use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
6 use Zend\Console\Adapter\AdapterInterface as Console;
7
8 class Module implements ConsoleBannerProviderInterface
9 {
10     /**
11      * This method is defined in ConsoleBannerProviderInterface
12      */
13     public function getConsoleBanner(Console $console){
14         return "User Module 0.0.1";
15     }
16 }
```

Because User module is loaded after Application module, the result will look like this:



```

iMac-de-Michael:public michael$ php index.php
MyModule 0.0.1
UserModule 0.0.1

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$
```

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.

Basic usage

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:

```
1 // modules/Application/Module.php
2 <?php
3 namespace Application;
4
5 use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
6 use Zend\ModuleManager\Feature\ConsoleUsageProviderInterface;
7 use Zend\Console\Adapter\AdapterInterface as Console;
8
9 class Module implements ConsoleBannerProviderInterface, ConsoleUsageProviderInterface
10 {
11     public function getConsoleBanner(Console $console){ // ... }
12
13     /**
14      * This method is defined in ConsoleUsageProviderInterface
15      */
16     public function getConsoleUsage(Console $console)
17     {
18         return array(
19             'show stats'          => 'Show application statistics',
20             'run cron'            => 'Run automated jobs',
21             '(enable|disable) debug' => 'Enable or disable debug mode for the_
↪application.'
22         );
23     }
24 }
```

This will display the following information:

```

iMac-de-Michael:public michael$ php index.php
MyModule 0.0.1
UserModule 0.0.1

-----
Application
-----

index.php show stats          Show application statistics
index.php run cron           Run automated jobs
index.php (enable|disable) debug  Enable or disable debug mode
                                   for the application.

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$

```

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*.

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:

```

1 public function getConsoleUsage(Console $console)
2 {
3     return 'User module expects exactly one argument - user name. It will display_
4     ↪information for this user.';
5 }

```

```

iMac-de-Michael:public michael$ php index.php
MyModule 0.0.1
UserModule 0.0.1

-----
Application
-----

index.php show stats          Show application statistics
index.php run cron           Run automated jobs
index.php (enable|disable) debug  Enable or disable debug mode
                                   for the application.

-----
User
-----

User module expects exactly one argument - user name. It will display information for this
user.

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$

```

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()`.

List of commands

If `getConsoleUsage()` returns an 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.

```

1 public function getConsoleUsage(Console $console)
2 {
3     return array(
4         'delete user <userEmail>'          => 'Delete user with email <userEmail>',
5         'disable user <userEmail>'          => 'Disable user with email <userEmail>',
6         'list [all|disabled] users'         => 'Show a list of users',
7         'find user [--email=] [--name=]'    => 'Attempt to find a user by email or name',
8     );
9 }

```

```

iMac-de-Michael:public michael$ php index.php
MyModule 0.0.1
UserModule 0.0.1

-----
Application
-----

index.php show stats          Show application statistics
index.php run cron           Run automated jobs
index.php (enable|disable) debug  Enable or disable debug mode for the application.

-----
User
-----

index.php delete user <userEmail>      Delete user with email <userEmail>
index.php disable user <userEmail>     Disable user with email <userEmail>
index.php list [all|disabled] users    Show a list of users
index.php find user [--email=] [--name=] Attempt to find a user by email or name

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$

```

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.

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:

```

1 public function getConsoleUsage(Console $console)
2 {
3     return array(
4         array( '<userEmail>' , 'email of the user' ),
5         array( '--verbose'   , 'Turn on verbose mode' ),
6         array( '--quick'     , 'Perform a "quick" operation' ),
7         array( '-v'          , 'Same as --verbose' ),
8         array( '-w'          , 'Wide output' )
9     );
10 }

```

```

iMac-de-Michael:public michael$ php index.php
MyModule 0.0.1
UserModule 0.0.1

-----
Application
-----

index.php show stats          Show application statistics
index.php run cron           Run automated jobs
index.php (enable|disable) debug  Enable or disable debug mode for the application.

-----
User
-----

<userEmail>    email of the user
--verbose      Turn on verbose mode
--quick        Perform a "quick" operation
-v            Same as --verbose
-w            Wide output

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$

```

Using this method, we can display more than 2 columns of information, for example:

```

1 public function getConsoleUsage(Console $console)
2 {
3     return array(
4         array( '<userEmail>' , 'user email'          , 'Full email address of the user_
↳to find.' ),
5         array( '--verbose'   , 'verbose mode'      , 'Display additional information_
↳during processing' ),
6         array( '--quick'     , '"quick" operation' , 'Do not check integrity, just_
↳make changes and finish' ),
7         array( '-v'          , 'Same as --verbose' , 'Display additional information_
↳during processing' ),
8         array( '-w'          , 'wide output'       , 'When listing users, use the_
↳whole available screen width' )
9     );
10 }

```

```

iMac-de-Michael:public michael$ php index.php
MyModule 0.0.1
UserModule 0.0.1

-----
Application
-----

index.php show stats      Show application statistics
index.php run cron       Run automated jobs
index.php (enable|disable) debug  Enable or disable debug mode for the application.

-----
User
-----

<userEmail>    user email      Full email address of the user to find.
--verbose      verbose mode    Display additional information during processing
--quick        "quick" operation  Do not check integrity, just make changes and finish
-v            Same as --verbose  Display additional information during processing
-w            wide output      When listing users, use the whole available screen width

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$

```

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.

Mixing styles

You can use mix together all of the above styles to provide comprehensive usage information, for example:

```

1 public function getConsoleUsage(Console $console)
2 {
3     return array(
4         'Finding and listing users',
5         'list [all|disabled] users [-w]'      => 'Show a list of users',
6         'find user [--email=] [--name=]'     => 'Attempt to find a user by email or_
↳name',
7
8         array('all|disabled]',             'Display all users or only disabled accounts'),
9         array('--email=EMAIL',             'Email of the user to find'),
10        array('--name=NAME',               'Full name of the user to find.'),
11        array('-w',                         'Wide output - When listing users use the whole_
↳available screen width' ),
12
13        'Manipulation of user database:',
14        'delete user <userEmail> [--verbose|-v] [--quick]' => 'Delete user with_
↳email <userEmail>',
15        'disable user <userEmail> [--verbose|-v]'          => 'Disable user with_
↳email <userEmail>',

```

```

16         array( '<userEmail>' , 'user email'          , 'Full email address of the user_
17 ↪to change.' ),
18         array( '--verbose'      , 'verbose mode'      , 'Display additional information_
19 ↪during processing' ),
20         array( '--quick'        , '"quick" operation' , 'Do not check integrity, just_
21 ↪make changes and finish' ),
22         array( '-v'             , 'Same as --verbose' , 'Display additional information_
23 ↪during processing' ),
24     );
25 }

```

```

public — bash — 119x32

index.php show stats          Show application statistics
index.php run cron           Run automated jobs
index.php (enable|disable) debug  Enable or disable debug mode for the application.

-----
User
-----

Finding and listing users
index.php list [all|disabled] users [-w]  Show a list of users
index.php find user [--email=] [--name=]  Attempt to find a user by email or name

[all|disabled]  Display all users or only disabled accounts
--email=EMAIL   Email of the user to find
--name=NAME     Full name of the user to find.
-w             Wide output - When listing users use the whole available screen width

Manipulation of user database:
index.php delete user <userEmail> [--verbose|-v] [--quick]  Delete user with email
                                                             <userEmail>
index.php disable user <userEmail> [--verbose|-v]           Disable user with email
                                                             <userEmail>

<userEmail>    user email          Full email address of the user to change.
--verbose      verbose mode        Display additional information during processing
--quick        "quick" operation   Do not check integrity, just make changes and finish
-v            Same as --verbose    Display additional information during processing

Reason for failure: Invalid arguments or no arguments provided
iMac-de-Michael:public michael$

```

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.

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*

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 Service-Manager, 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:

```
1 // FILE: modules/Application/config/module.config.php
2 array(
3     'router' => array(
4         'routes' => array(
5             // HTTP routes are here
6         )
7     ),
8
9     'console' => array(
```

```

10         'router' => array(
11             'routes' => array(
12                 'list-users' => array(
13                     'options' => array(
14                         'route'      => 'show [all|disabled|deleted]:mode users [--
↪ verbose|-v]',
15                         'defaults' => array(
16                             'controller' => 'Application\Controller\Index',
17                             'action'      => 'show-users'
18                         )
19                     )
20                 )
21             )
22         ),
23     ),
24 )
    
```

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.

```

1  <?php
2  namespace Application\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6
7  class IndexController extends AbstractActionController
8  {
9      public function indexAction()
10     {
11         return new ViewModel(); // display standard index page
12     }
13
14     public function showUsersAction()
15     {
16         $request = $this->getRequest();
17
18         // Check verbose flag
19         $verbose = $request->getParam('verbose') || $request->getParam('v');
20
21         // Check mode
22         $mode = $request->getParam('mode', 'all'); // defaults to 'all'
23
24         $users = array();
25         switch ($mode) {
26             case 'disabled':
27                 $users = $this->getServiceLocator()->get('users')->
↪ fetchDisabledUsers();
28                 break;
29             case 'deleted':
30                 $users = $this->getServiceLocator()->get('users')->
↪ fetchDeletedUsers();
    
```

```

31         break;
32     case 'all':
33     default:
34         $users = $this->getServiceLocator()->get('users')->fetchAllUsers();
35         break;
36     }
37 }
38 }

```

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.

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:

```

1 public function showUsersAction()
2 {
3     $request = $this->getRequest();
4
5     // Check verbose flag
6     $verbose = $request->getParam('verbose') || $request->getParam('v');
7
8     // Check mode
9     $mode = $request->getParam('mode', 'all'); // defaults to 'all'
10
11     $users = array();
12     switch ($mode) {
13         case 'disabled':
14             $users = $this->getServiceLocator()->get('users')->fetchDisabledUsers();
15             break;
16         case 'deleted':
17             $users = $this->getServiceLocator()->get('users')->fetchDeletedUsers();
18             break;
19         case 'all':
20         default:
21             $users = $this->getServiceLocator()->get('users')->fetchAllUsers();
22             break;
23     }
24
25     if (count($users) == 0) {
26         // Show an error message in the console
27         return "There are no users in the database\n";
28     }
29
30     $result = '';
31
32     foreach ($users as $user) {
33         $result .= $user->name . ' ' . $user->email . "\n";
34     }
35
36     return $result; // show it in the console
37 }

```

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.

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:

```
1 namespace Application\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\View\Model\ViewModel;
5 use Zend\Console\Request as ConsoleRequest;
6 use RuntimeException;
7
8 class IndexController extends AbstractActionController
9 {
10     public function showUsersAction()
11     {
12         $request = $this->getRequest();
13
14         // Make sure that we are running in a console and the user has not tricked our
15         // application into running this action from a public web server.
16         if (!$request instanceof ConsoleRequest) {
17             throw new RuntimeException('You can only use this action from a console!
18 ↪');
19         }
20         // ...
21     }
22 }
```

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**:

```
1 namespace Application\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\View\Model\ViewModel;
5 use Zend\Console\Request as ConsoleRequest;
6 use Zend\Http\Request as HttpRequest;
7 use RuntimeException;
8
9 class IndexController extends AbstractActionController
10 {
11     public function showUsersAction()
```

```

12 {
13     $request = $this->getRequest();
14
15     $users = array();
16     // ... fetch users from database ...
17
18     if ($request instanceof HttpRequest) {
19         // display a web page with users list
20         return new ViewModel($result);
21     } elseif ($request instanceof ConsoleRequest) {
22         // ... prepare console output and return it ...
23         return $result;
24     } else {
25         throw new RuntimeException('Cannot handle request of type ' . get_class(
26 ↪ $request));
27     }
28 }

```

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.

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:

```

1 // inside of config.console.router.routes:
2 'show-users' => array(
3     'options' => array(
4         'route' => 'show (all|deleted|locked|admin) [<groupName>]'
5         'defaults' => array(
6             'controller' => 'Application\Controller\Users',
7             'action' => 'showusers'
8         )
9     )
10 )

```

If this route matches, our action can now query parameters in the following way:

```

1 // an action inside Application\Controller\UsersController:
2 public function showUsersAction()
3 {
4     $request = $this->getRequest();
5
6     // We can access named value parameters directly by their name:
7     $showUsersFromGroup = $request->getParam('groupName');
8
9     // Literal parameters can be checked with isset() against their exact spelling
10    if (isset($request->getParam('all'))) {

```

```
11     // show all users
12 } elseif (isset($request->getParam('deleted'))) {
13     // show deleted users
14 }
15 // ...
16 }
```

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:

```
1 // inside of config.console.router.routes:
2 'show-users' => array(
3     'options' => array(
4         'route' => 'show (all|deleted|locked|admin):userTypeFilter [<groupName>]'
5         'defaults' => array(
6             'controller' => 'Application\Controller\Users',
7             'action' => 'showusers'
8         )
9     )
10 )
```

Now we can use a the group name `userTypeFilter` to check which option has been selected by the user:

```
1 public function showUsersAction()
2 {
3     $request = $this->getRequest();
4
5     // We can access named value parameters directly by their name:
6     $showUsersFromGroup = $request->getParam('groupName');
7
8     // The selected option from second parameter is now stored under 'userTypeFilter'
9     $userTypeFilter = $request->getParam('userTypeFilter');
10
11     switch ($userTypeFilter) {
12         case 'all':
13             // all users
14         case 'deleted':
15             // deleted users
16         case 'locked':
17             // ...
18             // ...
19     }
20 }
```

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:

```
1 'find-user' => array(
2     'options' => array(
3         'route' => 'find user [--fast] [--verbose] [--id=] [--firstName=] [--
4         ↪lastName=] [--email=] ',
5         'defaults' => array(
```

```

5         'controller' => 'Application\Controller\Users',
6         'action'      => 'find',
7     )
8 )
9 )

```

We can easily retrieve values in the following fashion:

```

1 public function findAction()
2 {
3     $request = $this->getRequest();
4
5     // We can retrieve values from value flags using their name
6     $searchId      = $request->getParam('id',      null); // default null
7     $searchFirstName = $request->getParam('firstName', null);
8     $searchLastName = $request->getParam('lastName', null);
9     $searchEmail    = $request->getParam('email',    null);
10
11     // Standard flags that have been matched will be equal to TRUE
12     $isFast         = (bool) $request->getParam('fast', false); // default false
13     $isVerbose       = (bool) $request->getParam('verbose', false);
14
15     if ($isFast) {
16         // perform a fast query ...
17     } else {
18         // perform standard query ...
19     }
20 }

```

In case of **flag alternatives**, we have to check each alternative separately:

```

1 // Assuming our route now reads:
2 //     'route' => 'find user [--fast|-f] [--verbose|-v] ... ',
3 //
4 public function findAction()
5 {
6     $request = $this->getRequest();
7
8     // Check both alternatives
9     $isFast      = $request->getParam('fast', false) || $request->getParam('f', false);
10    $isVerbose    = $request->getParam('verbose', false) || $request->getParam('v', false);
11
12    // ...
13 }

```

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.

Retrieving console adapter

If you are using *MVC controllers* you can obtain Console adapter instance using Service Manager.

```
1 namespace Application;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\Console\Adapter\AdapterInterface as Console;
5 use Zend\Console\Exception\RuntimeException;
6
7 class ConsoleController extends AbstractActionController
8 {
9     public function testAction()
10     {
11         $console = $this->getServiceLocator()->get('console');
12         if (!$console instanceof Console) {
13             throw new RuntimeException('Cannot obtain console adapter. Are we running
14 ↪ in a console?');
15         }
16     }
17 }
```

If you are using `Zend\Console` without MVC, we can get adapter using the following code:

```

1 use Zend\Console\Console;
2 use Zend\Console\Exception\RuntimeException as ConsoleException;
3
4 try {
5     $console = Console::getInstance();
6 } catch (ConsoleException $e) {
7     // Could not get console adapter - most likely we are not running inside a
8     ↪ console window.
9 }

```

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.

Using console adapter

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.

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.

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`; `$y = 1`. To retrieve far-right and bottom coordinates, use `getWidth()` and `getHeight()` methods.

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**.

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.

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.

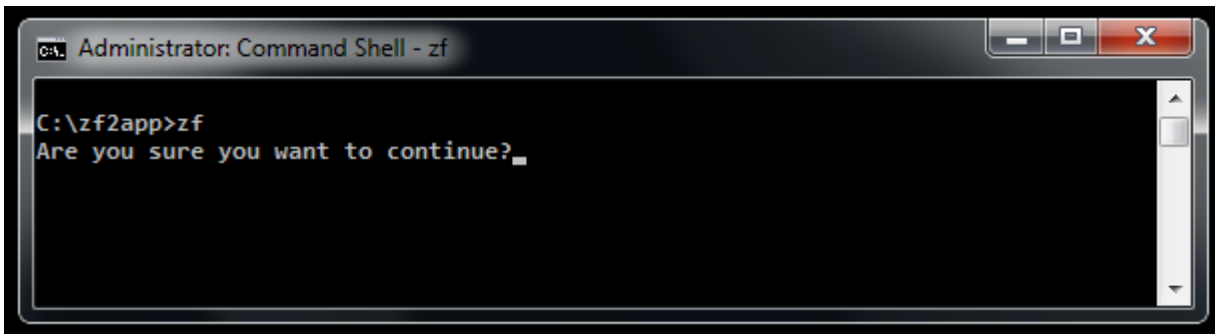
All prompts can be instantiated as objects and provide `show()` method.

```
1 use Zend\Console\Prompt;
2
3 $confirm = new Prompt\Confirm('Are you sure you want to continue?');
4 $result = $confirm->show();
5 if ($result) {
6     // the user chose to continue
7 }
```

There is also a shorter method of displaying prompts, using static `prompt()` method:

```
1 use Zend\Console\Prompt;
2
3 $result = Prompt\Confirm::prompt('Are you sure you want to continue?');
4 if ($result) {
5     // the user chose to continue
6 }
```

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.

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");
}
```



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)

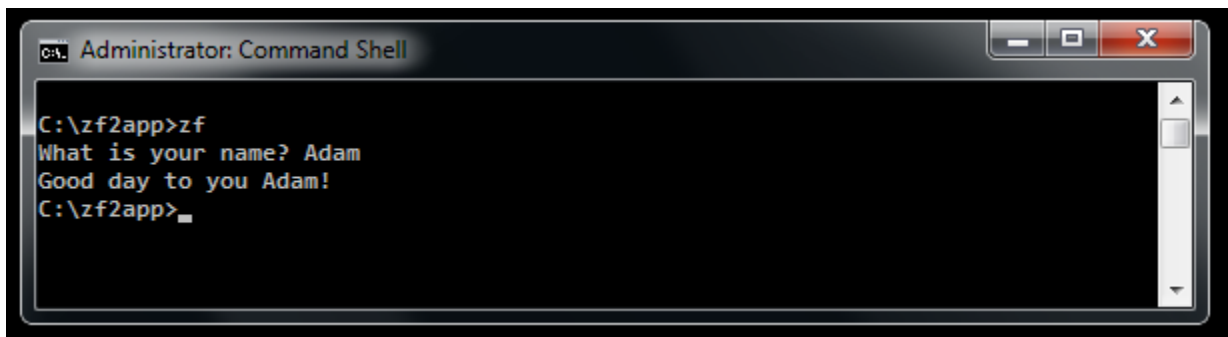
\$maxLength (*integer*) Maximum length of the input. Anything above this limit will be truncated.

Example usage:

```
use Zend\Console\Prompt\Line;

$name = Line::prompt(
    'What is your name?',
    false,
    100
);

$console->write("Good day to you $name!");
```



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)

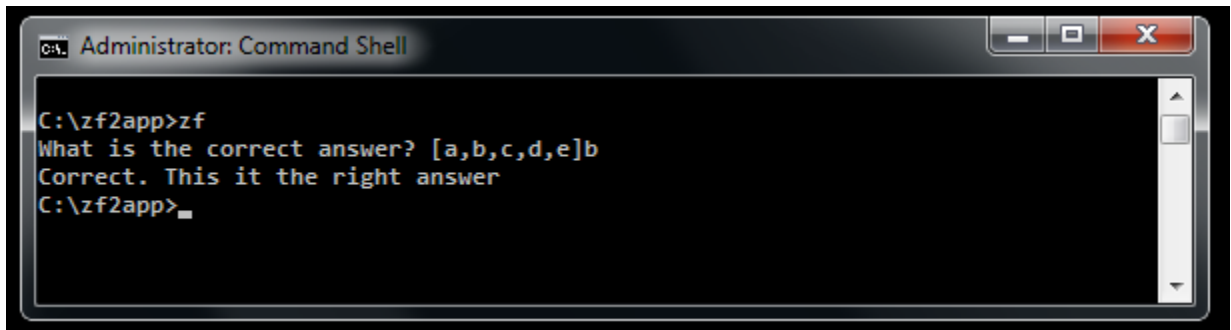
\$echo (*boolean*) Should the selection be displayed on the screen ?

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.');
```



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 ?

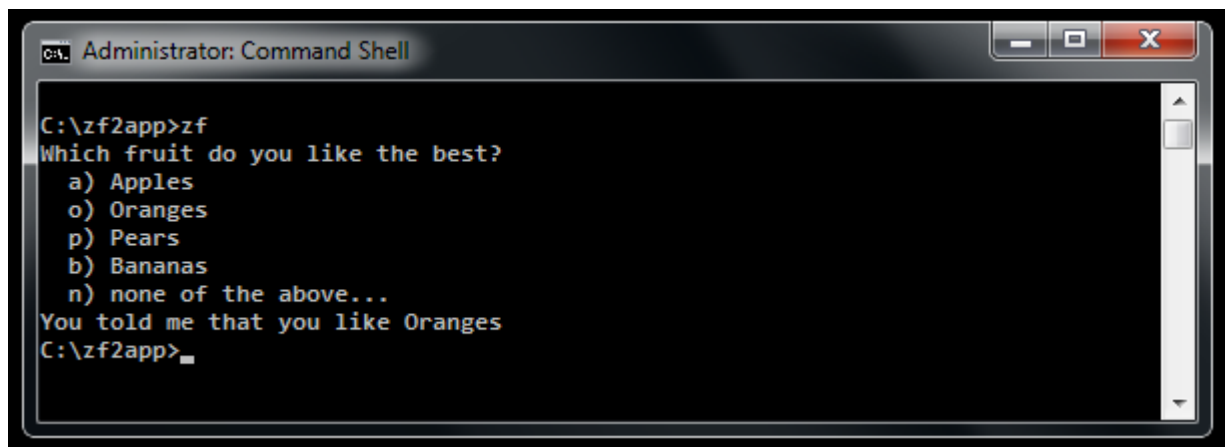
Example usage:

```
$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]);
```



Password

This prompt reads in a string, without echoing that string back to the console. Useful for password prompts.

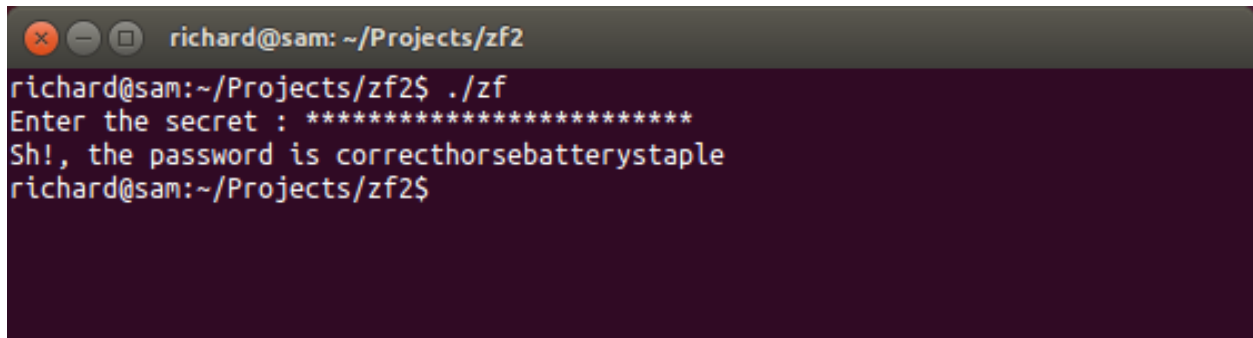
```
1 Password(
2     string $promptText = 'Password : ',
3     boolean $echo = true
4 )
```

\$promptText (*string*) The text to show with the prompt

\$echo (*bool*) Display * in place of characters. Can be skipped, defaults to true

Example usage:

```
1 use Zend\Console\Prompt>Password;
2
3 $password = Password::prompt('Enter the secret', true);
4
5 $console->write("Sh!, the password is $password");
```

A terminal window with a dark background and light text. The window title bar shows three icons (close, maximize, and a third icon) followed by the text 'richard@sam: ~/Projects/zf2'. The terminal content shows a prompt 'richard@sam:~/Projects/zf2\$' followed by the command './zf'. The next line is 'Enter the secret : *****'. The following line is 'Sh!, the password is correcthorsebatterystaple'. The final line is the prompt 'richard@sam:~/Projects/zf2\$' again.

```
richard@sam: ~/Projects/zf2
richard@sam:~/Projects/zf2$ ./zf
Enter the secret : *****
Sh!, the password is correcthorsebatterystaple
richard@sam:~/Projects/zf2$
```

See also:

To learn more about accessing console, writing to and reading from it, make sure to read the following chapter: *Console adapters*.

Introduction

The `Zend\Console\Getopt` class helps command-line applications to parse their options and arguments.

Users may specify command-line arguments when they execute your application. These arguments have meaning to the application, to change the behavior in some way, or choose resources, or specify parameters. Many options have developed customary meaning, for example `--verbose` enables extra output from many applications. Other options may have a meaning that is different for each application. For example, `-c` enables different features in `grep`, `ls`, and `tar`.

Below are a few definitions of terms. Common usage of the terms varies, but this documentation will use the definitions below.

- “argument”: a string that occurs on the command-line following the name of the command. Arguments may be options or else may appear without an option, to name resources on which the command operates.
- “option”: an argument that signifies that the command should change its default behavior in some way.
- “flag”: the first part of an option, identifies the purpose of the option. A flag is preceded conventionally by one or two dashes (`-` or `--`). A single dash precedes a single-character flag or a cluster of single-character flags. A double-dash precedes a multi-character flag. Long flags cannot be clustered.
- “parameter”: the secondary part of an option; a data value that may accompany a flag, if it is applicable to the given option. For example, many commands accept a `--verbose` option, but typically this option has no parameter. However, an option like `--user` almost always requires a following parameter.

A parameter may be given as a separate argument following a flag argument, or as part of the same argument string, separated from the flag by an equals symbol (`=`). The latter form is supported only by long flags. For example, `-u username`, `--user username`, and `--user=username` are forms supported by `Zend\Console\Getopt`.

- “cluster”: multiple single-character flags combined in a single string argument and preceded by a single dash. For example, `ls -lstr` uses a cluster of four short flags. This command is equivalent to `ls -l -s -t -r`. Only single-character flags can be clustered. You cannot make a cluster of long flags.

For example, in `mysql --user=root mydatabase`, `mysql` is a **command**, `--user=root` is an **option**, `--user` is a **flag**, `root` is a **parameter** to the option, and `mydatabase` is an argument but not an option by our definition.

`Zend\Console\Getopt` provides an interface to declare which flags are valid for your application, output an error and usage message if they use an invalid flag, and report to your application code which flags the user specified.

Note: Getopt is not an Application Framework

`Zend\Console\Getopt` does **not** interpret the meaning of flags and parameters, nor does this class implement application workflow or invoke application code. You must implement those actions in your own application code. You can use the `Zend\Console\Getopt` class to parse the command-line and provide object-oriented methods for querying which options were given by a user, but code to use this information to invoke parts of your application should be in another *PHP* class.

The following sections describe usage of `Zend\Console\Getopt`.

Declaring Getopt Rules

The constructor for the `Zend\Console\Getopt` class takes from one to three arguments. The first argument declares which options are supported by your application. This class supports alternative syntax forms for declaring the options. See the sections below for the format and usage of these syntax forms.

The constructor takes two more arguments, both of which are optional. The second argument may contain the command-line arguments. This defaults to `$_SERVER['argv']`.

The third argument of the constructor may contain an configuration options to customize the behavior of `Zend\Console\Getopt`. See *Adding Configuration* for reference on the options available.

Declaring Options with the Short Syntax

`Zend\Console\Getopt` supports a compact syntax similar to that used by *GNU Getopt* (see http://www.gnu.org/software/libc/manual/html_node/Getopt.html). This syntax supports only single-character flags. In a single string, you type each of the letters that correspond to flags supported by your application. A letter followed by a colon character (:) indicates a flag that requires a parameter.

Using the Short Syntax

```
1 $opts = new Zend\Console\Getopt('abp:');
```

The example above shows using `Zend\Console\Getopt` to declare that options may be given as `-a`, `-b`, or `-p`. The latter flag requires a parameter.

The short syntax is limited to flags of a single character. Aliases, parameter types, and help strings are not supported in the short syntax.

Declaring Options with the Long Syntax

A different syntax with more features is also available. This syntax allows you to specify aliases for flags, types of option parameters, and also help strings to describe usage to the user. Instead of the single string used in the short syntax to declare the options, the long syntax uses an associative array as the first argument to the constructor.

The key of each element of the associative array is a string with a format that names the flag, with any aliases, separated by the pipe symbol (“|”). Following this series of flag aliases, if the option requires a parameter, is an equals symbol (“=”) with a letter that stands for the **type** of the parameter:

- “=s” for a string parameter
- “=w” for a word parameter (a string containing no whitespace)
- “=i” for an integer parameter

If the parameter is optional, use a dash (“-”) instead of the equals symbol.

The value of each element in the associative array is a help string to describe to a user how to use your program.

Using the Long Syntax

```
1 $opts = new Zend\Console\Getopt(  
2     array(  
3         'apple|a'    => 'apple option, with no parameter',  
4         'banana|b=i' => 'banana option, with required integer parameter',  
5         'pear|p-s'   => 'pear option, with optional string parameter'  
6     )  
7 );
```

In the example declaration above, there are three options. `--apple` and `-a` are aliases for each other, and the option takes no parameter. `--banana` and `-b` are aliases for each other, and the option takes a mandatory integer parameter. Finally, `--pear` and `-p` are aliases for each other, and the option may take an optional string parameter.

Fetching Options and Arguments

After you have declared the options that the `Zend\Console\Getopt` object should recognize, and supply arguments from the command-line or an array, you can query the object to find out which options were specified by a user in a given command-line invocation of your program. The class implements magic methods so you can query for options by name.

The parsing of the data is deferred until the first query you make against the `Zend\Console\Getopt` object to find out if an option was given, the object performs its parsing. This allows you to use several method calls to configure the options, arguments, help strings, and configuration options before parsing takes place.

Handling Getopt Exceptions

If the user gave any invalid options on the command-line, the parsing function throws a `Zend\Console\Exception\RuntimeException`. You should catch this exception in your application code. You can use the `parse()` method to force the object to parse the arguments. This is useful because you can invoke `parse()` in a **try** block. If it passes, you can be sure that the parsing won't throw an exception again. The exception thrown has a custom method `getUsageMessage()`, which returns as a string the formatted set of usage messages for all declared options.

Catching Getopt Exceptions

```
1 try {  
2     $opts = new Zend\Console\Getopt('abp:');  
3     $opts->parse();  
4 } catch (Zend\Console\Exception\RuntimeException $e) {  
5     echo $e->getUsageMessage();  
6     exit;  
7 }
```

Cases where parsing throws an exception include:

- Option given is not recognized.

- Option requires a parameter but none was given.
- Option parameter is of the wrong type. E.g. a non-numeric string when an integer was required.

Fetching Options by Name

You can use the `getOption()` method to query the value of an option. If the option had a parameter, this method returns the value of the parameter. If the option had no parameter but the user did specify it on the command-line, the method returns `TRUE`. Otherwise the method returns `NULL`.

Using `getOption()`

```
1 $opts = new Zend\Console\Getopt('abp:');
2 $b = $opts->getOption('b');
3 $p_parameter = $opts->getOption('p');
```

Alternatively, you can use the magic `__get()` function to retrieve the value of an option as if it were a class member variable. The `__isset()` magic method is also implemented.

Using `__get()` and `__isset()` Magic Methods

```
1 $opts = new Zend\Console\Getopt('abp:');
2 if (isset($opts->b)) {
3     echo "I got the b option.\n";
4 }
5 $p_parameter = $opts->p; // null if not set
```

If your options are declared with aliases, you may use any of the aliases for an option in the methods above.

Reporting Options

There are several methods to report the full set of options given by the user on the current command-line.

- As a string: use the `toString()` method. The options are returned as a space-separated string of `flag=value` pairs. The value of an option that does not have a parameter is the literal string “`TRUE`”.
- As an array: use the `toArray()` method. The options are returned in a simple integer-indexed array of strings, the flag strings followed by parameter strings, if any.
- As a string containing *JSON* data: use the `toJson()` method.
- As a string containing *XML* data: use the `toXml()` method.

In all of the above dumping methods, the flag string is the first string in the corresponding list of aliases. For example, if the option aliases were declared like `verbose|v`, then the first string, `verbose`, is used as the canonical name of the option. The name of the option flag does not include any preceding dashes.

Fetching Non-option Arguments

After option arguments and their parameters have been parsed from the command-line, there may be additional arguments remaining. You can query these arguments using the `getRemainingArgs()` method. This method returns an array of the strings that were not part of any options.

Using `getRemainingArgs()`

```
1 $opts = new Zend\Console\Getopt('abp:');
2 $opts->setArguments(array('-p', 'p_parameter', 'filename'));
3 $args = $opts->getRemainingArgs(); // returns array('filename')
```

`Zend\Console\Getopt` supports the *GNU* convention that an argument consisting of a double-dash signifies the end of options. Any arguments following this signifier must be treated as non-option arguments. This is useful if you might have a non-option argument that begins with a dash. For example: “`rm -- -filename-with-dash`”.

Configuring Zend\Console\Getopt

Adding Option Rules

You can add more option rules in addition to those you specified in the `Zend\Console\Getopt` constructor, using the `addRules()` method. The argument to `addRules()` is the same as the first argument to the class constructor. It is either a string in the format of the short syntax options specification, or else an associative array in the format of a long syntax options specification. See *Declaring Getopt Rules* for details on the syntax for specifying options.

Using `addRules()`

```
1 $opts = new Zend\Console\Getopt('abp:');
2 $opts->addRules(
3     array(
4         'verbose|v' => 'Print verbose output'
5     )
6 );
```

The example above shows adding the `--verbose` option with an alias of `-v` to a set of options defined in the call to the constructor. Notice that you can mix short format options and long format options in the same instance of `Zend\Console\Getopt`.

Adding Help Messages

In addition to specifying the help strings when declaring option rules in the long format, you can associate help strings with option rules using the `setHelp()` method. The argument to the `setHelp()` method is an associative array, in which the key is a flag, and the value is a corresponding help string.

Using setHelp()

```
1 $opts = new Zend\Console\Getopt('abp:');
2 $opts->setHelp(
3     array(
4         'a' => 'apple option, with no parameter',
5         'b' => 'banana option, with required integer parameter',
6         'p' => 'pear option, with optional string parameter'
7     )
8 );
```

If you declared options with aliases, you can use any of the aliases as the key of the associative array.

The `setHelp()` method is the only way to define help strings if you declared the options using the short syntax.

Adding Option Aliases

You can declare aliases for options using the `setAliases()` method. The argument is an associative array, whose key is a flag string declared previously, and whose value is a new alias for that flag. These aliases are merged with any existing aliases. In other words, aliases you declared earlier are still in effect.

An alias may be declared only once. If you try to redefine an alias, a `Zend\Console\Getopt\Exception` is thrown.

Using setAliases()

```
1 $opts = new Zend\Console\Getopt('abp:');
2 $opts->setAliases(
3     array(
4         'a' => 'apple',
5         'a' => 'apfel',
6         'p' => 'pear'
7     )
8 );
```

In the example above, after declaring these aliases, `-a`, `--apple` and `--apfel` are aliases for each other. Also `-p` and `--pear` are aliases for each other.

The `setAliases()` method is the only way to define aliases if you declared the options using the short syntax.

Adding Argument Lists

By default, `Zend\Console\Getopt` uses `$_SERVER['argv']` for the array of command-line arguments to parse. You can alternatively specify the array of arguments as the second constructor argument. Finally, you can append more arguments to those already used using the `addArguments()` method, or you can replace the current array of arguments using the `setArguments()` method. In both cases, the parameter to these methods is a simple array of strings. The former method appends the array to the current arguments, and the latter method substitutes the array for the current arguments.

Using addArguments() and setArguments()

```

1 // By default, the constructor uses $_SERVER['argv']
2 $opts = new Zend\Console\Getopt('abp:');
3
4 // Append an array to the existing arguments
5 $opts->addArguments(array('-a', '-p', 'p_parameter', 'non_option_arg'));
6
7 // Substitute a new array for the existing arguments
8 $opts->setArguments(array('-a', '-p', 'p_parameter', 'non_option_arg'));

```

Adding Configuration

The third parameter to the `Zend\Console\Getopt` constructor is an array of configuration options that affect the behavior of the object instance returned. You can also specify configuration options using the `setOptions()` method, or you can set an individual option using the `setOption()` method.

Note: Clarifying the Term “option”

The term “option” is used for configuration of the `Zend\Console\Getopt` class to match terminology used elsewhere in Zend Framework. These are not the same things as the command-line options that are parsed by the `Zend\Console\Getopt` class.

The currently supported options have const definitions in the class. The options, their const identifiers (with literal values in parentheses) are listed below:

- `Zend\Console\Getopt::CONFIG_DASHDASH` (“dashDash”), if `TRUE`, enables the special flag `--` to signify the end of flags. Command-line arguments following the double-dash signifier are not interpreted as options, even if the arguments start with a dash. This configuration option is `TRUE` by default.
- `Zend\Console\Getopt::CONFIG_IGNORECASE` (“ignoreCase”), if `TRUE`, makes flags aliases of each other if they differ only in their case. That is, `-a` and `-A` will be considered to be synonymous flags. This configuration option is `FALSE` by default.
- `Zend\Console\Getopt::CONFIG_RULEMODE` (“ruleMode”) may have values `Zend\Console\Getopt::MODE_ZEND` (“zend”) and `Zend\Console\Getopt::MODE_GNU` (“gnu”). It should not be necessary to use this option unless you extend the class with additional syntax forms. The two modes supported in the base `Zend\Console\Getopt` class are unambiguous. If the specifier is a string, the class assumes `MODE_GNU`, otherwise it assumes `MODE_ZEND`. But if you extend the class and add more syntax forms, you may need to specify the mode using this option.

More configuration options may be added as future enhancements of this class.

The two arguments to the `setOption()` method are a configuration option name and an option value.

Using setOption()

```

1 $opts = new Zend\Console\Getopt('abp:');
2 $opts->setOption('ignoreCase', true);

```

The argument to the `setOptions()` method is an associative array. The keys of this array are the configuration option names, and the values are configuration values. This is also the array format used in the class constructor. The configuration values you specify are merged with the current configuration; you don’t have to list all options.

Using setOptions()

```
1 $opts = new Zend\Console\Getopt('abp:');
2 $opts->setOptions(
3     array(
4         'ignoreCase' => true,
5         'dashDash'   => false
6     )
7 );
```

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.

Encrypt/decrypt using block ciphers

Zend\Crypt\BlockCipher implements the encrypt-then-authenticate mode using [HMAC](#) to provide authentication.

The symmetric cipher can be chosen 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).

```
1 use Zend\Crypt\BlockCipher;
2
3 $blockCipher = BlockCipher::factory('mcrypt', array('algo' => 'aes'));
4 $blockCipher->setKey('encryption key');
5 $result = $blockCipher->encrypt('this is a secret message');
6 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:

```
1 use Zend\Crypt\BlockCipher;
2
3 $blockCipher = BlockCipher::factory('mcrypt', array(
4     'algo' => 'blowfish',
5     'mode' => 'cfb',
6     'hash' => 'sha512'
7 ));
```

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:

```
1 use Zend\Crypt\BlockCipher;
2 use Zend\Crypt\Symmetric\Mcrypt;
3
4 $blockCipher = new BlockCipher(new Mcrypt(array('algo' => 'aes')));
5 $blockCipher->setKey('encryption key');
6 $result = $blockCipher->encrypt('this is a secret message');
7 echo "Encrypted text: $result \n";
```

Encrypt/decrypt a file

Zend\Crypt\FileCipher implements the encryption of decryption of a file using a symmetric cipher in CBC mode with the encrypt-then-authenticate approach, using HMAC to provide authentication (the same solution used by Zend\Crypt\BlockCipher component).

Encrypt and decrypt a file is not an easy task, especially a big file. For instance, in CBC mode you must be sure to handle the IV correctly for each block. That means, if you are reading a big file you need to use a buffer and be sure to use the last block of the buffer as new IV for the next encryption step.

The FileCipher uses a symmetric cipher, with the Zend\Crypt\Symmetric\Mcrypt component.

The usage of this component is very simple, you just need to create an instance of FileCipher and specify the key, and you are ready to encrypt/decrypt any file:

```
1 use Zend\Crypt\FileCipher;
2
3 $fileCipher = new FileCipher;
4 $fileCipher->setKey('encryption key');
5 // encryption
6 if ($fileCipher->encrypt('path/to/file_to_encrypt', 'path/to/output')) {
7     echo "The file has been encrypted successfully\n";
8 }
9 // decryption
10 if ($fileCipher->decrypt('path/to/file_to_decrypt', 'path/to/output')) {
11     echo "The file has been decrypted successfully\n";
12 }
```

By default FileCipher uses the AES encryption algorithm (with a key of 256 bit) and the SHA-256 hash algorithm to authenticate the data using the HMAC function. This component uses the PBKDF2 key derivation algorithm to generate the encryption key and the authentication key, for the HMAC, based on the key specified using the method setKey().

If you want to change the encryption algorithm, you can use the setCipherAlgorithm() function, for instance you can specify to use the Blowfish encryption algorithm using setCipherAlgorithm('blowfish'). You can retrieve the list of all the supported encryption algorithm in your environment using the function getCipherSupportedAlgorithms(), it will return an array of all the algorithm name.

If you need to customize the cipher algorithm, for instance changing the Padding mode, you can inject your Mcrypt object in the `FileCipher` using the `setCipher()` method. The only parameter of the cipher that you cannot change is the cipher mode, that will be CBC in any case.

Note: Output format

The output of the encryption file is in binary format. We used this format to do not impact on the output size. If you encrypt a file using the `FileCipher` component, you will notice that the output file size is almost the same of the input size, just some bytes more to store the HMAC and the IV vector. The format of the output is the concatenation of the HMAC, the IV and the encrypted file.

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.

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.

```
1 use Zend\Crypt\Key\Derivation\Pbkdf2;
2 use Zend\Math\Rand;
3
4 $pass = 'password';
5 $salt = Rand::getBytes(32, true);
6 $key = Pbkdf2::calc('sha256', $pass, $salt, 10000, 32);
7
8 printf ("Original password: %s\n", $pass);
9 printf ("Derived key (hex): %s\n", bin2hex($key));
```

The Pbkdf2 adapter takes the password (`$pass`) and generate a binary key of 32 bytes. The syntax is `calc($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 string of 32 bytes for the salt, using a strong generator (the `true` value means the usage of a cryptographically strong generator).

The number of iterations is a very important parameter for the security of the algorithm. Bigger values guarantee 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.

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 to generate a key of 32 bytes.

```
1 use Zend\Crypt\Key\Derivation\SaltedS2k;
2 use Zend\Math\Rand;
3
4 $pass = 'password';
5 $salt = Rand::getBytes(32, true);
6 $key = SaltedS2k::calc('sha256', $pass, $salt, 32);
7
8 printf ("Original password: %s\n", $pass);
9 printf ("Derived key (hex): %s\n", bin2hex($key));
```

Script adapter

The `script` 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 `script` 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 `script` 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 `script` algorithm)

This algorithm uses 4 parameters to generate a key of 32 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 `Script` adapter.

```
1 use Zend\Crypt\Key\Derivation\Script;
2 use Zend\Math\Rand;
3
4 $pass = 'password';
5 $salt = Rand::getBytes(32, true);
6 $key = Script::calc($pass, $salt, 2048, 2, 1, 32);
7
8 printf ("Original password: %s\n", $pass);
9 printf ("Derived key (hex): %s\n", bin2hex($key));
```

Note: Performance of the script implementation

The aim of the `script` 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 `script` 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 `script` algorithm we suggest to install the [script PECL](#) extension. The `Script` adapter of Zend Framework is able to recognize if the PECL extension is loaded and use it instead of the pure PHP implementation.

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).

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 10, that means about 0.07 second using a CPU Intel i5 at 3.3Ghz (the *cost* parameter is a relative value according to the speed of the CPU used). We changed the default value of the cost parameter from 14 to 10, starting from Zend Framework 2.3.0, due to high computational time to prevent potential denial-of-service attacks (you can read this article [Aggressive password stretching](#) for more information).

If you want to change the *cost* parameter of the *bcrypt* algorithm you can use the `setCost()` method. Please note, 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:

```
1 use Zend\Crypt\Password\Bcrypt;
2
3 $bcrypt = new Bcrypt();
4 $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).

Note: Bcrypt truncates input > 72 bytes

The input string of the bcrypt algorithm is limited to 72 bytes. If you use a string with a length more than this limit, bcrypt will consider only the first 72 bytes. If you need to use a longer string, you should pre-hash it using SHA256 prior to passing it to the bcrypt algorithm: `$hashedPassword = \Zend\Crypt\Hash::compute('sha256', $password);`

To verify if a given password is valid against a bcrypt value you can use the `verify()` method. An example is reported below:

```
1 use Zend\Crypt\Password\Bcrypt;
2
3 $bcrypt = new Bcrypt();
4 $securePass = 'the stored bcrypt value';
5 $password = 'the password to check';
6
7 if ($bcrypt->verify($password, $securePass)) {
8     echo "The password is correct! \n";
9 } else {
10     echo "The password is NOT correct.\n";
11 }
```

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:

```
1 use Zend\Crypt\Password\Bcrypt;
2
3 $bcrypt = new Bcrypt(array(
4     'salt' => 'random value',
5     'cost' => 11
6 ));
```

Note: Bcrypt with non-ASCII passwords (8-bit characters)

The bcrypt implementation used by PHP < 5.3.7 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.

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:

```

1 use Zend\Crypt\Password\Apache;
2
3 $apache = new Apache();
4
5 $apache->setFormat('crypt');
6 printf("CRYPT output: %s\n", $apache->create('password'));
7
8 $apache->setFormat('sha1');
9 printf("SHA1 output: %s\n", $apache->create('password'));
10
11 $apache->setFormat('md5');
12 printf("MD5 output: %s\n", $apache->create('password'));
13
14 $apache->setFormat('digest');
15 $apache->setUserName('enrico');
16 $apache->setAuthName('test');
17 printf("Digest output: %s\n", $apache->create('password'));
```

You can also specify the format of the password during the constructor of the class:

```

1 use Zend\Crypt\Password\Apache;
2
3 $apache = new Apache(array(
4     'format' => 'md5'
5 ));
```

Other possible parameters to pass in the constructor are *username* and *authname*, for the digest format.

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](#).

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.

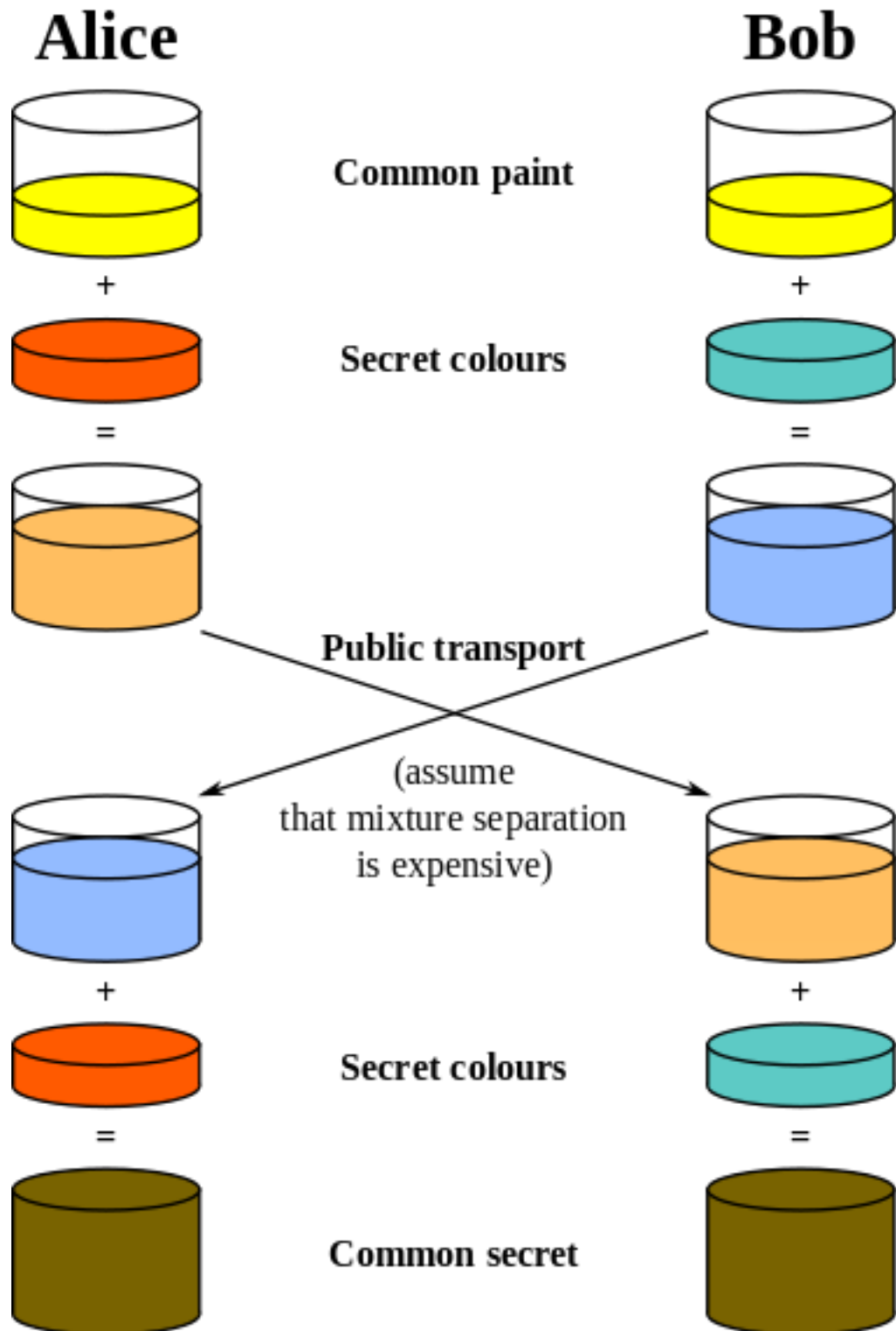
The diagram of operation of the Diffie-Hellman algorithm can be defined by the following picture (taken by the [Diffie-Hellman](#) Wikipedia page):

The schema's colors represent the parameters of the algorithm. Here is reported an example of usage using the `Zend\Crypt\PublicKey\DiffieHellman` class:

```

1  use Zend\Crypt\PublicKey\DiffieHellman;
2
3  $aliceOptions = array(
4      'prime' =>
5          '155172898181473697471232257763715539915724801966915404479707795314057629378541917580651227
6          ' .
7          '423698188993727816152646631438561595825688188889951272158842675419950341258706556549803580
8          ' .
9          '104870537681476726513255747040765857479291291572334510643245094715007229621094194349783925
10         ' .

```



```

7         '984760375594985848253359305585439638443',
8         'generator'=> '2',
9         'private' =>
10         ↪ '992093140665725952364085695919679885571412495614942674862518080355353963322786201435363176
11         ↪ ' .
12         ↪ '813127128916726230726309951803243888416814918577455156967890911274095150092503589658166661
13         ↪ ' .
14         ↪ '463420498381785213791321533481399080168191962194483101070726325157493390557981225386151351
15         ↪ ' .
16         ↪ '04828702523796951800575031871051678091'
17     );
18
19     $bobOptions = array(
20         'prime' => $aliceOptions['prime'],
21         'generator'=> '2',
22         'private' =>
23         ↪ '334117357926395586257336357178925636125481806504021611510774783148414637079488997861035889
24         ↪ ' .
25         ↪ '123256347304105519467727528801778689728169635518217403867000760342134081539246925625431179
26         ↪ ' .
27         ↪ '634647331566005454845108330724270034742070646507148310833044977371603820970833568760781462
28         ↪ ' .
29         ↪ '31616972608703322302585471319261275664'
30     );
31
32     $alice = new DiffieHellman($aliceOptions['prime'], $aliceOptions['generator'],
33         ↪ $aliceOptions['private']);
34     $bob = new DiffieHellman($bobOptions['prime'], $bobOptions['generator'],
35         ↪ $bobOptions['private']);
36
37     $alice->generateKeys();
38     $bob->generateKeys();
39
40     $aliceSecretKey = $alice->computeSecretKey($bob->getPublicKey(DiffieHellman::FORMAT_
41         ↪ BINARY),
42         ↪ DiffieHellman::FORMAT_BINARY,
43         ↪ DiffieHellman::FORMAT_BINARY);
44
45     $bobSecretKey = $bob->computeSecretKey($alice->getPublicKey(DiffieHellman::FORMAT_
46         ↪ BINARY),
47         ↪ DiffieHellman::FORMAT_BINARY,
48         ↪ DiffieHellman::FORMAT_BINARY);
49
50     if ($aliceSecretKey !== $bobSecretKey) {
51         echo "ERROR!\n";
52     } else {
53         printf("The secret key is: %s\n", base64_encode($aliceSecretKey));
54     }

```

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`.

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.

Generate a public key and a private key

In order to generate a public and private key you can use the following code:

```
1 use Zend\Crypt\PublicKey\RsaOptions;
2
3 $rsaOptions = new RsaOptions(array(
4     'pass_phrase' => 'test'
5 ));
6
7 $rsaOptions->generateKeys(array(
8     'private_key_bits' => 2048,
9 ));
10
11 file_put_contents('private_key.pem', $rsaOptions->getPrivateKey());
12 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
```


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.

```

1  use Zend\Crypt\PublicKey\Rsa;
2
3  $rsa = Rsa::factory(array(
4      'public_key'    => 'public_key.pub',
5      'private_key'   => 'private_key.pem',
6      'pass_phrase'   => 'test',
7      'binary_output' => false
8  ));
9
10 $text = 'This is the message to encrypt';
11
12 $encrypt = $rsa->encrypt($text);
13 printf("Encrypted message:\n%s\n", $encrypt);
14
15 $decrypt = $rsa->decrypt($encrypt);
16
17 if ($text !== $decrypt) {
18     echo "ERROR\n";
19 } else {
20     echo "Encryption and decryption performed successfully!\n";
21 }

```

Generate a digital signature of a file

Below is reported an example of how to generate a digital signature of a file.

```

1  use Zend\Crypt\PublicKey\Rsa;
2
3  $rsa = Rsa::factory(array(
4      'private_key'    => 'path/to/private_key',
5      'pass_phrase'    => 'passphrase of the private key',
6      'binary_output'  => false
7  ));
8
9  $file = file_get_contents('path/file/to/sign');
10
11 $signature = $rsa->sign($file, $rsa->getOptions()->getPrivateKey());
12 $verify     = $rsa->verify($file, $signature, $rsa->getOptions()->getPublicKey());
13
14 if ($verify) {
15     echo "The signature is OK\n";
16     file_put_contents($filename . '.sig', $signature);
17     echo "Signature save in $filename.sig\n";
18 } else {
19     echo "The signature is not valid!\n";
20 }

```

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.

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.

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`:

```
1 $adapter = new Zend\Db\Adapter\Adapter(array(  
2     'driver' => 'Mysqli',  
3     'database' => 'zend_db_example',  
4     'username' => 'developer',  
5     'password' => 'developer-password'  
6 ));
```

Another example, of a Sqlite connection via PDO:

```
1 $adapter = new Zend\Db\Adapter\Adapter(array(  
2     'driver' => 'Pdo_Sqlite',  
3     'database' => 'path/to/sqlite.db'  
4 ));
```

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 through the PDO extension
- `Pdo_Pgsql`: PostgreSQL through the PDO extension

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):

```
1 use Zend\Db\Adapter\Platform\PlatformInterface;  
2 use Zend\Db\ResultSet\ResultSet;  
3  
4 class Zend\Db\Adapter\Adapter {  
5     public function __construct($driver, PlatformInterface $platform = null,  
6     ↪ResultSet $queryResultSetPrototype = null)  
7 }
```

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

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:

```
1 $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

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:

```
1 $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.

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.

```
1 // with optional parameters to bind up-front  
2 $statement = $adapter->createStatement($sql, $optionalParameters);  
3 $result = $statement->execute();
```

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:

```

1 namespace Zend\Db\Adapter\Driver;
2
3 interface DriverInterface
4 {
5     const PARAMETERIZATION_POSITIONAL = 'positional';
6     const PARAMETERIZATION_NAMED = 'named';
7     const NAME_FORMAT_CAMEL_CASE = 'camelCase';
8     const NAME_FORMAT_NATURAL = 'natural';
9     public function getDatabasePlatformName($nameFormat = self::NAME_FORMAT_
10 ↪ CAMEL_CASE);
11     public function checkEnvironment();
12     public function getConnection();
13     public function createStatement($sqlOrResource = null);
14     public function createResult($resource);
15     public function getPrepareType();
16     public function formatParameterName($name, $type = null);
17     public function getLastGeneratedValue();
18 }
```

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:

```

1 namespace Zend\Db\Adapter\Driver;
2
3 interface StatementInterface extends StatementContainerInterface
4 {
```

```

5     public function getResource();
6     public function prepare($sql = null);
7     public function isPrepared();
8     public function execute($parameters = null);
9
10    /** Inherited from StatementContainerInterface */
11    public function setSql($sql);
12    public function getSql();
13    public function setParameterContainer(ParameterContainer $parameterContainer);
14    public function getParameterContainer();
15 }

```

Result objects generally look like this:

```

1 namespace Zend\Db\Adapter\Driver;
2
3 interface ResultInterface extends \Countable, \Iterator
4 {
5     public function buffer();
6     public function isQueryResult();
7     public function getAffectedRows();
8     public function getGeneratedValue();
9     public function getResource();
10    public function getFieldCount();
11 }

```

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:

```

1 namespace Zend\Db\Adapter\Platform;
2
3 interface PlatformInterface
4 {
5     public function getName();
6     public function getQuoteIdentifierSymbol();
7     public function quoteIdentifier($identifier);
8     public function quoteIdentifierChain($identifierChain);
9     public function getQuoteValueSymbol();
10    public function quoteValue($value);
11    public function quoteValueList($valueList);
12    public function getIdentifierSeparator();
13    public function quoteIdentifierInFragment($identifier, array $additionalSafeWords,
14    => array());
15 }

```

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):

```

1 $platform = $adapter->getPlatform();
2 // or
3 $platform = $adapter->platform; // magic property access

```

The following is a couple of example of Platform usage:

```

1  /** @var $adapter Zend\Db\Adapter\Adapter */
2  /** @var $platform Zend\Db\Adapter\Platform\Sql92 */
3  $platform = $adapter->getPlatform();
4
5  // "first_name"
6  echo $platform->quoteIdentifier('first_name');
7
8  // "
9  echo $platform->getQuoteIdentifierSymbol();
10
11 // "schema"."mytable"
12 echo $platform->quoteIdentifierChain(array('schema', 'mytable'));
13
14 // '
15 echo $platform->getQuoteValueSymbol();
16
17 // 'myvalue'
18 echo $platform->quoteValue('myvalue');
19
20 // 'value', 'Foo O\'Bar'
21 echo $platform->quoteValueList(array('value', "Foo O'Bar"));
22
23 // .
24 echo $platform->getIdentifierSeparator();
25
26 // "foo" as "bar"
27 echo $platform->quoteIdentifierInFragment('foo as bar');
28
29 // additionally, with some safe words:
30 // ("foo"."bar" = "boo"."baz")
31 echo $platform->quoteIdentifierInFragment('(foo.bar = boo.baz)', array('(', ')', '=',
    ↪ '));

```

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.

Examples

Creating a Driver and Vendor portable Query, Preparing and Iterating Result

```

1 $adapter = new Zend\Db\Adapter\Adapter($driverConfig);
2
3 $qi = function($name) use ($adapter) { return $adapter->platform->quoteIdentifier(
4     ↪$name); };
5
6 $fp = function($name) use ($adapter) { return $adapter->driver->formatParameterName(
7     ↪$name); };
8
9 $sql = 'UPDATE ' . $qi('artist')
10       . ' SET ' . $qi('name') . ' = ' . $fp('name')
11       . ' WHERE ' . $qi('id') . ' = ' . $fp('id');

```

```
10  /** @var $statement Zend\Db\Adapter\Driver\StatementInterface */
11  $statement = $adapter->query($sql);
12
13  $parameters = array(
14      'name' => 'Updated Artist',
15      'id' => 1
16  );
17
18  $statement->execute($parameters);
19
20  // DATA INSERTED, NOW CHECK
21
22  /** @var $statement Zend\Db\Adapter\DriverStatementInterface */
23  $statement = $adapter->query('SELECT * FROM '
24      . $qi('artist')
25      . ' WHERE id = ' . $fp('id'));
26
27  /** @var $results Zend\Db\ResultSet\ResultSet */
28  $results = $statement->execute(array('id' => 1));
29
30  $row = $results->current();
31  $name = $row['name'];
```

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:

```
1 interface ResultSetInterface extends \Traversable, \Countable
2 {
3     public function initialize($dataSource);
4     public function getFieldCount();
5 }
```

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():

```
1 use Zend\Db\Adapter\Driver\ResultInterface;
2 use Zend\Db\ResultSet\ResultSet;
3
4 $stmt = $driver->createStatement('SELECT * FROM users');
5 $stmt->prepare();
6 $result = $stmt->execute($parameters);
```

```
7
8 if ($result instanceof ResultInterface && $result->isQueryResult()) {
9     $resultSet = new ResultSet;
10    $resultSet->initialize($result);
11
12    foreach ($resultSet as $row) {
13        echo $row->my_column . PHP_EOL;
14    }
15 }
```

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:

```
1 abstract class AbstractResultSet implements Iterator, ResultSetInterface
2 {
3     public function initialize($dataSource)
4     public function getDataSource()
5     public function getFieldCount()
6
7     /** Iterator */
8     public function next()
9     public function key()
10    public function current()
11    public function valid()
12    public function rewind()
13
14    /** countable */
15    public function count()
16
17    /** get rows as array */
18    public function toArray()
19 }
```

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:

```
1 use Zend\Db\Adapter\Driver\ResultInterface;
2 use Zend\Db\ResultSet\HydratingResultSet;
3 use Zend\Stdlib\Hydrator\Reflection as ReflectionHydrator;
4
5 class UserEntity {
6     protected $first_name;
7     protected $last_name;
```

```

8     public function getFirstName() { return $this->first_name; }
9     public function getLastName() { return $this->last_name; }
10    public function setFirstName($first_name) { $this->first_name = $first_name; }
11    public function setLastName($last_name) { $this->last_name = $last_name; }
12 }
13
14 $stmt = $driver->createStatement($sql);
15 $stmt->prepare($parameters);
16 $result = $stmt->execute();
17
18 if ($result instanceof ResultInterface && $result->isQueryResult()) {
19     $resultSet = new HydratingResultSet(new ReflectionHydrator, new UserEntity);
20     $resultSet->initialize($result);
21
22     foreach ($resultSet as $user) {
23         echo $user->getFirstName() . ' ' . $user->getLastName() . PHP_EOL;
24     }
25 }

```

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.

Zend\Db\Sql is a SQL abstraction layer for building platform specific SQL queries via an 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.

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.

```
1 use Zend\Db\Sql\Sql;  
2 $sql = new Sql($adapter);  
3 $select = $sql->select(); // @return Zend\Db\Sql\Select  
4 $insert = $sql->insert(); // @return Zend\Db\Sql\Insert  
5 $update = $sql->update(); // @return Zend\Db\Sql\Update  
6 $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):

```
1 use Zend\Db\Sql\Sql;  
2 $sql = new Sql($adapter);  
3 $select = $sql->select();  
4 $select->from('foo');  
5 $select->where(array('id' => 2));  
6
```

```

7 $statement = $sql->prepareStatementForSqlObject($select);
8 $results = $statement->execute();

```

To execute (using a Select object)

```

1 use Zend\Db\Sql\Sql;
2 $sql = new Sql($adapter);
3 $select = $sql->select();
4 $select->from('foo');
5 $select->where(array('id' => 2));
6
7 $selectString = $sql->buildSqlString($select);
8 $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.

```

1 use Zend\Db\Sql\Sql;
2 $sql = new Sql($adapter, 'foo');
3 $select = $sql->select();
4 $select->where(array('id' => 2)); // $select already has the from('foo') applied

```

Zend\Db\Sql's Select, Insert, Update and Delete

Each of these objects implements the following (2) interfaces:

```

1 interface PreparableSqlInterface {
2     public function prepareStatement(Adapter $adapter, StatementInterface
3     ↪ $statement);
4 }
5 interface SqlInterface {
6     public function getSqlString(PlatformInterface $adapterPlatform = null);
7 }

```

These are the functions you can call to either produce (a) a prepared statement, or (b) a string to be executed.

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:

```

1 use Zend\Db\Sql\Select;
2 $select = new Select();
3 // or, to produce a $select bound to a specific table
4 $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:

```

1 class Select extends AbstractSql implements SqlInterface, PreparableSqlInterface
2 {
3     const JOIN_INNER = 'inner';

```



```

4     const JOIN_OUTER = 'outer';
5     const JOIN_LEFT = 'left';
6     const JOIN_RIGHT = 'right';
7     const SQL_STAR = '*';
8     const ORDER_ASCENDING = 'ASC';
9     const ORDER_DESCENDING = 'DESC';
10
11     public $where; // @param Where $where
12
13     public function __construct($table = null);
14     public function from($table);
15     public function columns(array $columns, $prefixColumnsWithTable = true);
16     public function join($name, $on, $columns = self::SQL_STAR, $type = self::JOIN_
17 ↪ INNER);
18     public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
19     public function group($group);
20     public function having($predicate, $combination = Predicate\PredicateSet::OP_AND);
21     public function order($order);
22     public function limit($limit);
23     public function offset($offset);
24 }

```

from():

```

1 // as a string:
2 $select->from('foo');
3
4 // as an array to specify an alias:
5 // produces SELECT "t".* FROM "table" AS "t"
6
7 $select->from(array('t' => 'table'));
8
9 // using a Sql\TableIdentifier:
10 // same output as above
11
12 $select->from(new TableIdentifier(array('t' => 'table')));

```

columns():

```

1 // as array of names
2 $select->columns(array('foo', 'bar'));
3
4 // as an associative array with aliases as the keys:
5 // produces 'bar' AS 'foo', 'bax' AS 'baz'
6
7 $select->columns(array('foo' => 'bar', 'baz' => 'bax'));

```

join():

```

1 $select->join(
2     'foo', // table name
3     'id = bar.id', // expression to join on (will be quoted by platform object
4 ↪ before insertion),

```

```

4      array('bar', 'baz'), // (optional) list of columns, same requirements as
↳ columns() above
5      $select::JOIN_OUTER // (optional), one of inner, outer, left, right also
↳ represented by constants in the API
6  );
7
8  $select->from(array('f' => 'foo')) // base table
9      ->join(array('b' => 'bar'), // join table with alias
10         'f.foo_id = b.foo_id'); // join expression

```

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:

```

1  /**
2   * Create where clause
3   *
4   * @param Where|\Closure|string|array $predicate
5   * @param string $combination One of the OP_* constants from Predicate\PredicateSet
6   * @return Select
7   */
8  public function where($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:

```

1  $spec = function (Where $where) {
2      $where->like('username', 'ralph%');
3  };
4
5  $select->where($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:

```

1  // SELECT "foo".* FROM "foo" WHERE x = 5
2  $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:

```

1  // SELECT "foo".* FROM "foo" WHERE x = 5 AND y = z
2  $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:

```

1 // SELECT "foo".* FROM "foo" WHERE "c1" IS NULL AND "c2" IN (?, ?, ?) AND "c3" IS NOT NULL
2 $select->from('foo')->where(array(
3     'c1' => null,
4     'c2' => array(1, 2, 3),
5     new \Zend\Db\Sql\Predicate\IsNotNull('c3')
6 ));
    
```

order():

```

1 $select = new Select;
2 $select->order('id DESC'); // produces 'id' DESC
3
4 $select = new Select;
5 $select->order('id DESC')
6     ->order('name ASC, age DESC'); // produces 'id' DESC, 'name' ASC, 'age' DESC
7
8 $select = new Select;
9 $select->order(array('name ASC', 'age DESC')); // produces 'name' ASC, 'age' DESC
    
```

limit() and offset():

```

1 $select = new Select;
2 $select->limit(5); // always takes an integer/numeric
3 $select->offset(10); // similarly takes an integer/numeric
    
```

Zend\Db\Sql\Insert

The Insert API:

```

1 class Insert implements SqlInterface, PreparableSqlInterface
2 {
3     const VALUES_MERGE = 'merge';
4     const VALUES_SET    = 'set';
5
6     public function __construct($table = null);
7     public function into($table);
8     public function columns(array $columns);
9     public function values(array $values, $flag = self::VALUES_SET);
10 }
    
```

Similarly to Select objects, the table can be set at construction time or via `into()`.

columns():

```
1 $insert->columns(array('foo', 'bar')); // set the valid columns
```

values():

```
1 // default behavior of values is to set the values
2 // successive calls will not preserve values from previous calls
3 $insert->values(array(
4     'col_1' => 'value1',
5     'col_2' => 'value2'
6 ));
```

```
1 // merging values with previous calls
2 $insert->values(array('col_2' => 'value2'), $insert::VALUES_MERGE);
```

Zend\Db\Sql\Update

```
1 class Update
2 {
3     const VALUES_MERGE = 'merge';
4     const VALUES_SET    = 'set';
5
6     public $where; // @param Where $where
7     public function __construct($table = null);
8     public function table($table);
9     public function set(array $values, $flag = self::VALUES_SET);
10    public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
11 }
```

set():

```
1 $update->set(array('foo' => 'bar', 'baz' => 'bax'));
```

where():

See where section below.

Zend\Db\Sql\Delete

```
1 class Delete
2 {
3     public $where; // @param Where $where
4     public function __construct($table = null);
5     public function from($table);
6     public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
7 }
```

where():

See where section below.

Zend\Db\Sql\Where & Zend\Db\Sql\Having

In the following, we will talk about Where, Having 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.

Note: In ZF 2.1, an actual `Literal` type was added. `Zend\Db\Sql` now makes the distinction that `Literals` will not have any parameters that need interpolating whereas it is expected that `Expression` objects *might* have parameters that need interpolating. In cases where there are parameters in an `Expression`, `Zend\Db\Sql\AbstractSql` will do its best to identify placeholders when the `Expression` is processed during statement creation. In short, if you don't have parameters, use `Literal` objects.

The `Zend\Db\Sql\Where` (Predicate/PredicateSet) API:

```

1 // Where & Having:
2 class Predicate extends PredicateSet
3 {
4     public $and;
5     public $or;
6     public $AND;
7     public $OR;
8     public $NEST;
9     public $UNNEST;
10
11     public function nest();
12     public function setUnnest(Predicate $predicate);
13     public function unnest();
14     public function equalTo($left, $right, $leftType = self::TYPE_IDENTIFIER,
15 ↪ $rightType = self::TYPE_VALUE);
16     public function notEqualTo($left, $right, $leftType = self::TYPE_IDENTIFIER,
17 ↪ $rightType = self::TYPE_VALUE);
18     public function lessThan($left, $right, $leftType = self::TYPE_IDENTIFIER,
19 ↪ $rightType = self::TYPE_VALUE);
20     public function greaterThan($left, $right, $leftType = self::TYPE_IDENTIFIER,
21 ↪ $rightType = self::TYPE_VALUE);
22     public function lessThanOrEqualTo($left, $right, $leftType = self::TYPE_
23 ↪ IDENTIFIER, $rightType = self::TYPE_VALUE);
24     public function greaterThanOrEqualTo($left, $right, $leftType = self::TYPE_
25 ↪ IDENTIFIER, $rightType = self::TYPE_VALUE);
26     public function like($identifier, $like);

```

```

21 public function literal($literal);
22 public function expression($expression, $parameter);
23 public function isNull($identifier);
24 public function isNotNull($identifier);
25 public function in($identifier, array $valueSet = array());
26 public function between($identifier, $minValue, $maxValue);
27
28
29 // Inherited From PredicateSet
30
31 public function addPredicate(PredicateInterface $predicate, $combination = null);
32 public function getPredicates();
33 public function orPredicate(PredicateInterface $predicate);
34 public function andPredicate(PredicateInterface $predicate);
35 public function getExpressionData();
36 public function count();
37 }

```

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:

equalTo(), lessThan(), greaterThan(), lessThanOrEqualTo(), greaterThanOrEqualTo():

```

1 $where->equalTo('id', 5);
2
3 // same as the following workflow
4 $where->addPredicate(
5     new Predicate\Operator($left, Operator::OPERATOR_EQUAL_TO, $right, $leftType,
6     ↪ $rightType)
7 );
8
9 class Operator implements PredicateInterface
10 {
11     const OPERATOR_EQUAL_TO          = '=';
12     const OP_EQ                      = '=';
13     const OPERATOR_NOT_EQUAL_TO      = '!=';
14     const OP_NE                      = '!=';
15     const OPERATOR_LESS_THAN         = '<';
16     const OP_LT                      = '<';
17     const OPERATOR_LESS_THAN_OR_EQUAL_TO = '<=';
18     const OP_LTE                     = '<=';
19     const OPERATOR_GREATER_THAN      = '>';
20     const OP_GT                      = '>';
21     const OPERATOR_GREATER_THAN_OR_EQUAL_TO = '>=';
22     const OP_GTE                     = '>=';
23
24     public function __construct($left = null, $operator = self::OPERATOR_EQUAL_TO,
25     ↪ $right = null, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE);
26     public function setLeft($left);
27     public function getLeft();
28     public function setLeftType($type);
29     public function getLeftType();
30     public function setOperator($operator);
31     public function getOperator();
32     public function setRight($value);

```

```

31     public function getRight();
32     public function setRightType($type);
33     public function getRightType();
34     public function getExpressionData();
35 }

```

like(\$identifier, \$like):

```

1  $where->like($identifier, $like):
2
3  // same as
4  $where->addPredicate(
5      new Predicate\Like($identifier, $like)
6  );
7
8  // full API
9
10 class Like implements PredicateInterface
11 {
12     public function __construct($identifier = null, $like = null);
13     public function setIdentifier($identifier);
14     public function getIdentifier();
15     public function setLike($like);
16     public function getLike();
17 }

```

literal(\$literal);

```

1  $where->literal($literal);
2
3  // same as
4  $where->addPredicate(
5      new Predicate\Literal($literal)
6  );
7
8  // full API
9  class Literal implements ExpressionInterface, PredicateInterface
10 {
11     const PLACEHOLDER = '?';
12     public function __construct($literal = '');
13     public function setLiteral($literal);
14     public function getLiteral();
15 }

```

expression(\$expression, \$parameter);

```

1  $where->expression($expression, $parameter);
2
3  // same as
4  $where->addPredicate(
5      new Predicate\Expression($expression, $parameter)
6  );

```

```
7
8 // full API
9 class Expression implements ExpressionInterface, PredicateInterface
10 {
11     const PLACEHOLDER = '?';
12     public function __construct($expression = null, $valueParameter = null /*[,
13     ↪ $valueParameter, ... ]*/);
14     public function setExpression($expression);
15     public function getExpression();
16     public function setParameters($parameters);
17     public function getParameters();
18     public function setTypes(array $types);
19     public function getTypes();
20 }
```

isNull(\$identifier);

```
1 $where->isNull($identifier);
2
3 // same as
4 $where->addPredicate(
5     new Predicate\IsNull($identifier)
6 );
7
8 // full API
9 class IsNull implements PredicateInterface
10 {
11     public function __construct($identifier = null);
12     public function setIdentifier($identifier);
13     public function getIdentifier();
14 }
```

isNotNull(\$identifier);

```
1 $where->isNotNull($identifier);
2
3 // same as
4 $where->addPredicate(
5     new Predicate\IsNotNull($identifier)
6 );
7
8 // full API
9 class IsNotNull implements PredicateInterface
10 {
11     public function __construct($identifier = null);
12     public function setIdentifier($identifier);
13     public function getIdentifier();
14 }
```


in(\$identifier, array \$valueSet = array());

```

1  $where->in($identifier, array $valueSet = array());
2
3  // same as
4  $where->addPredicate(
5      new Predicate\In($identifier, $valueSet)
6  );
7
8  // full API
9  class In implements PredicateInterface
10 {
11     public function __construct($identifier = null, array $valueSet = array());
12     public function setIdentifier($identifier);
13     public function getIdentifier();
14     public function setValueSet(array $valueSet);
15     public function getValueSet();
16 }

```

between(\$identifier, \$minValue, \$maxValue);

```

1  $where->between($identifier, $minValue, $maxValue);
2
3  // same as
4  $where->addPredicate(
5      new Predicate\Between($identifier, $minValue, $maxValue)
6  );
7
8  // full API
9  class Between implements PredicateInterface
10 {
11     public function __construct($identifier = null, $minValue = null, $maxValue = ↪null);
12     public function setIdentifier($identifier);
13     public function getIdentifier();
14     public function setMinValue($minValue);
15     public function getMinValue();
16     public function setMaxValue($maxValue);
17     public function getMaxValue();
18     public function setSpecification($specification);
19 }

```


Zend\Db\Sql\Ddl is a sub-component of Zend\Db\Sql that allows consumers to create statement objects that will produce DDL (Data Definition Language) SQL statements. When combined with a platform specific Zend\Db\Sql\Sql object, these DDL objects are capable of producing platform-specific CREATE TABLE statements, with specialized data types, constraints, and indexes for a database/schema.

The following platforms have platform specializations for DDL:

- MySQL
- All databases compatible with ANSI SQL92

Creating Tables

Like Zend\Db\Sql objects, each statement type is represented by a class. For example, CREATE TABLE is modeled by a CreateTable object; this is likewise the same for ALTER TABLE (as AlterTable), and DROP TABLE (as DropTable). These classes exist in the Zend\Db\Sql\Ddl namespace. To initiate the building of a DDL statement, such as CreateTable, one needs to instantiate the object. There are a couple of valid patterns for this:

```
1 use Zend\Db\Sql\Ddl;
2
3 $table = new Ddl\CreateTable();
4
5 // or with table
6 $table = new Ddl\CreateTable('bar');
7
8 // optionally, as a temporary table
9 $table = new Ddl\CreateTable('bar', true);
```

You can also set the table after instantiation:

```
1 $table->setTable('bar');
```

Currently, columns are added by creating a column object, described in the data type table in the data type section below:

```
1 use Zend\Db\Sql\Ddl\Column;
2 $table->addColumn(new Column\Integer('id'));
3 $table->addColumn(new Column\Varchar('name', 255));
```

Beyond adding columns to a table, constraints can also be added:

```
1 use Zend\Db\Sql\Ddl\Constraint;
2 $table->addConstraint(new Constraint\PrimaryKey('id'));
3 $table->addConstraint(
4     new Constraint\UniqueKey(['name', 'foo'], 'my_unique_key')
5 );
```

Altering Tables

Similarly to CreateTable, you may also instantiate AlterTable:

```
1 use Zend\Db\Sql\Ddl;
2
3 $table = new Ddl\AlterTable();
4
5 // or with table
6 $table = new Ddl\AlterTable('bar');
7
8 // optionally, as a temporary table
9 $table = new Ddl\AlterTable('bar', true);
```

The primary difference between a CreateTable and AlterTable is that the AlterTable takes into account that the table and its assets already exist. Therefore, while you still have addColumn() and addConstraint(), you will also see the ability to change existing columns:

```
1 use Zend\Db\Sql\Ddl\Column;
2 $table->changeColumn('name', Column\Varchar('new_name', 50));
```

You may also drop existing columns or constraints:

```
1 $table->dropColumn('foo');
2 $table->dropConstraint('my_index');
```

Dropping Tables

To drop a table, create a DropTable statement object:

```
1 $drop = new Ddl\DropTable('bar');
```

Executing DDL Statements

After a DDL statement object has been created and configured, at some point you will want to execute the statement. To do this, you will need two other objects: an Adapter instance, and a properly seeded Sql instance.

The workflow looks something like this, with `$ddl` being a `CreateTable`, `AlterTable`, or `DropTable` instance:

```

1 use Zend\Db\Sql\Sql;
2
3 // existence of $adapter is assumed
4 $sql = new Sql($adapter);
5
6 $adapter->query(
7     $sql->getSqlStringForSqlObject($ddl),
8     $adapter::QUERY_MODE_EXECUTE
9 );

```

By passing the `$ddl` object through the `$sql` object's `getSqlStringForSqlObject()` method, we ensure that any platform specific specializations/modifications are utilized to create a platform specific SQL statement.

Next, using the constant `Zend\Db\Adapter\Adapter::QUERY_MODE_EXECUTE` ensures that the SQL statement is not prepared, as many DDL statements on a variety of platforms cannot be prepared, only executed.

Currently Supported Data Types

These types exist in the `Zend\Db\Sql\Ddl\Column` namespace. Data types must implement `Zend\Db\Sql\Ddl\Column\ColumnInterface`.

In alphabetical order:

Type	Arguments For Construction
BigInteger	<code>\$name</code> , <code>\$nullable = false</code> , <code>\$default = null</code> , array <code>\$options = array()</code>
Blob	<code>\$name</code> , <code>\$length</code> , <code>\$nullable = false</code> , <code>\$default = null</code> , array <code>\$options = array()</code>
Boolean	<code>\$name</code>
Char	<code>\$name</code> , <code>\$length</code>
Column (generic)	<code>\$name = null</code>
Date	<code>\$name</code>
Decimal	<code>\$name</code> , <code>\$precision</code> , <code>\$scale = null</code>
Float	<code>\$name</code> , <code>\$digits</code> , <code>\$decimal</code> (Note: this class is deprecated as of 2.4.0; use <code>Floating</code> instead)
Floating	<code>\$name</code> , <code>\$digits</code> , <code>\$decimal</code>
Integer	<code>\$name</code> , <code>\$nullable = false</code> , <code>\$default = null</code> , array <code>\$options = array()</code>
Time	<code>\$name</code>
Varchar	<code>\$name</code> , <code>\$length</code>

Each of the above types can be utilized in any place that accepts a `Column\ColumnInterface` instance. Currently, this is primarily in `CreateTable::addColumn()` and `AlterTable`'s `addColumn()` and `changeColumn()` methods.

Currently Supported Constraint Types

These types exist in the `Zend\Db\Sql\Ddl\Constraint` namespace. Data types must implement `Zend\Db\Sql\Ddl\Constraint\ConstraintInterface`.

In alphabetical order:

Type	Arguments For Construction
Check	<code>\$expression, \$name</code>
For- eignKey	<code>\$name, \$column, \$referenceTable, \$referenceColumn, \$onDeleteRule = null, \$onUpdateRule = null</code>
Prima- ryKey	<code>\$columns</code>
UniqueKey	<code>\$column, \$name = null</code>

Each of the above types can be utilized in any place that accepts a `Column\ConstraintInterface` instance. Currently, this is primarily in `CreateTable::addConstraint()` and `AlterTable::addConstraint()`.

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:

```
1 interface Zend\Db\TableGateway\TableGatewayInterface
2 {
3     public function getTable();
4     public function select($where = null);
5     public function insert($set);
6     public function update($set, $where = null);
7     public function delete($where);
8 }
```

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.

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:

```
1 class TableGateway extends AbstractTableGateway
2 {
3     public $lastInsertValue;
4     public $table;
```

```

5     public $adapter;
6
7     public function __construct($table, Adapter $adapter, $features = null,
    ↳ResultSet $resultSetPrototype = null, Sql $sql = null)
8
9         /** Inherited from AbstractTableGateway */
10
11     public function isInitialized();
12     public function initialize();
13     public function getTable();
14     public function getAdapter();
15     public function getColumns();
16     public function getFeatureSet();
17     public function getResultSetPrototype();
18     public function getSql();
19     public function select($where = null);
20     public function selectWith(Select $select);
21     public function insert($set);
22     public function insertWith(Insert $insert);
23     public function update($set, $where = null);
24     public function updateWith(Update $update);
25     public function delete($where);
26     public function deleteWith(Delete $delete);
27     public function getLastInsertValue();
28 }
    
```

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.

```

1  use Zend\Db\TableGateway\TableGateway;
2  $projectTable = new TableGateway('project', $adapter);
3  $rowset = $projectTable->select(array('type' => 'PHP'));
4
5  echo 'Projects of type PHP: ';
6  foreach ($rowset as $projectRow) {
7      echo $projectRow['name'] . PHP_EOL;
8  }
9
10 // or, when expecting a single row:
11 $artistTable = new TableGateway('artist', $adapter);
12 $rowset = $artistTable->select(array('id' => 2));
13 $artistRow = $rowset->current();
14
15 var_dump($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:

```

1  use Zend\Db\TableGateway\TableGateway;
2  use Zend\Db\Sql\Select;
3  $artistTable = new TableGateway('artist', $adapter);
4
    
```



```

5 // search for at most 2 artists who's name starts with Brit, ascending
6 $rowset = $artistTable->select(function (Select $select) {
7     $select->where->like('name', 'Brit%');
8     $select->order('name ASC')->limit(2);
9 });

```

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 through 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:

```

1 use Zend\Db\TableGateway\AbstractTableGateway;
2 use Zend\Db\TableGateway\Feature;
3
4 class MyTableGateway extends AbstractTableGateway
5 {
6     public function __construct()
7     {
8         $this->table = 'my_table';
9         $this->featureSet = new Feature\FeatureSet();
10        $this->featureSet->addFeature(new Feature\GlobalAdapterFeature());
11        $this->initialize();
12    }
13 }
14
15 // elsewhere in code, in a bootstrap
16 Zend\Db\TableGateway\Feature\GlobalAdapterFeature::setStaticAdapter($adapter);
17
18 // in a controller, or model somewhere
19 $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.

```

1 $table = new TableGateway('artist', $adapter, new Feature\MasterSlaveFeature(
2     ↪ $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.

```

1 $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.

```
1 $table = new TableGateway('artist', $adapter, new Feature\EventFeature(  
    ↪$eventManagerInterface));
```

- RowGatewayFeature: the ability for `select()` to return a `ResultSet` object that upon iteration will return a `RowGateway` object for each row.

```
1 $table = new TableGateway('artist', $adapter, new Feature\RowGatewayFeature('id'));  
2 $results = $table->select(array('id' => 2));  
3  
4 $artistRow = $results->current();  
5 $artistRow->name = 'New Name';  
6 $artistRow->save();
```

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:

```
1 interface RowGatewayInterface
2 {
3     public function save();
4     public function delete();
5 }
```

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:

```
1 use Zend\Db\RowGateway\RowGateway;
2
3 // query the database
4 $resultSet = $adapter->query('SELECT * FROM `user` WHERE `id` = ?', array(2));
5
6 // get array of data
7 $rowData = $resultSet->current()->getArrayCopy();
8
9 // row gateway
10 $rowGateway = new RowGateway('id', 'my_table', $adapter);
11 $rowGateway->populate($rowData, true);
```

```
12
13 $rowGateway->first_name = 'New Name';
14 $rowGateway->save();
15
16 // or delete this row:
17 $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:

```
1 use Zend\Db\TableGateway\Feature\RowGatewayFeature;
2 use Zend\Db\TableGateway\TableGateway;
3
4 $table = new TableGateway('artist', $adapter, new RowGatewayFeature('id'));
5 $results = $table->select(array('id' => 2));
6
7 $artistRow = $results->current();
8 $artistRow->name = 'New Name';
9 $artistRow->save();
```

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:

```
1 use Zend\Db\TableGateway\Feature\RowGatewayFeature;
2 use Zend\Db\TableGateway\TableGateway;
3 use Zend\Db\RowGateway\RowGatewayInterface;
4
5 class Artist implements RowGatewayInterface
6 {
7     protected $adapter;
8
9     public function __construct($adapter)
10     {
11         $this->adapter = $adapter;
12     }
13
14     // ... save() and delete() implementations
15 }
16
17 $table = new TableGateway('artist', $adapter, new RowGatewayFeature(new Artist(
18     ↪ $adapter)));
```

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:

```
1 interface MetadataInterface
2 {
3     public function getSchemas();
4
5     public function getTableNames($schema = null, $includeViews = false);
6     public function getTables($schema = null, $includeViews = false);
7     public function getTable($tableName, $schema = null);
8
9     public function getViewNames($schema = null);
10    public function getViews($schema = null);
11    public function getView($viewName, $schema = null);
12
13    public function getColumnNames($table, $schema = null);
14    public function getColumns($table, $schema = null);
15    public function getColumn($columnName, $table, $schema = null);
16
17    public function getConstraints($table, $schema = null);
18    public function getConstraint($constraintName, $table, $schema = null);
19    public function getConstraintKeys($constraint, $table, $schema = null);
20
21    public function getTriggerNames($schema = null);
22    public function getTriggers($schema = null);
23    public function getTrigger($triggerName, $schema = null);
24 }
```

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.

```
1  $metadata = new Zend\Db\Metadata\Metadata($adapter);
2
3  // get the table names
4  $tableNames = $metadata->getTableNames();
5
6  foreach ($tableNames as $tableName) {
7      echo 'In Table ' . $tableName . PHP_EOL;
8
9      $table = $metadata->getTable($tableName);
10
11
12     echo '    With columns: ' . PHP_EOL;
13     foreach ($table->getColumns() as $column) {
14         echo '        ' . $column->getName()
15             . ' -> ' . $column->getDataType()
16             . PHP_EOL;
17     }
18
19     echo PHP_EOL;
20     echo '    With constraints: ' . PHP_EOL;
21
22     foreach ($metadata->getConstraints($tableName) as $constraint) {
23         /** @var $constraint Zend\Db\Metadata\Object\ConstraintObject */
24         echo '        ' . $constraint->getName()
25             . ' -> ' . $constraint->getType()
26             . PHP_EOL;
27         if (!$constraint->hasColumns()) {
28             continue;
29         }
30         echo '            column: ' . implode(', ', $constraint->getColumns());
31         if ($constraint->isForeignKey()) {
32             $fkCols = array();
33             foreach ($constraint->getReferencedColumns() as $refColumn) {
34                 $fkCols[] = $constraint->getReferencedTableName() . '.' . $refColumn;
35             }
36             echo ' => ' . implode(', ', $fkCols);
37         }
38         echo PHP_EOL;
39     }
40 }
41
42 echo '----' . PHP_EOL;
43 }
```

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`:

```

1 class Zend\Db\Metadata\Object\TableObject
2 {
3     public function __construct($name);
4     public function setColumns(array $columns);
5     public function getColumns();
6     public function setConstraints($constraints);
7     public function getConstraints();
8     public function setName($name);
9     public function getName();
10 }

```

The ColumnObject:

```

1 class Zend\Db\Metadata\Object\ColumnObject {
2     public function __construct($name, $tableName, $schemaName = null);
3     public function setName($name);
4     public function getName();
5     public function getTableName();
6     public function setTableName($tableName);
7     public function setSchemaName($schemaName);
8     public function getSchemaName();
9     public function getOrdinalPosition();
10    public function setOrdinalPosition($ordinalPosition);
11    public function getColumnDefault();
12    public function setColumnDefault($columnDefault);
13    public function getIsNullable();
14    public function setIsNullable($isNullable);
15    public function isNullable();
16    public function getDataType();
17    public function setDataType($dataType);
18    public function getCharacterMaximumLength();
19    public function setCharacterMaximumLength($characterMaximumLength);
20    public function getCharacterOctetLength();
21    public function setCharacterOctetLength($characterOctetLength);
22    public function getNumericPrecision();
23    public function setNumericPrecision($numericPrecision);
24    public function getNumericScale();
25    public function setNumericScale($numericScale);
26    public function getNumericUnsigned();
27    public function setNumericUnsigned($numericUnsigned);
28    public function isNumericUnsigned();
29    public function getErratas();
30    public function setErratas(array $erratas);
31    public function getErrata($errataName);
32    public function setErrata($errataName, $errataValue);
33 }

```

The ConstraintObject:

```

1 class Zend\Db\Metadata\Object\ConstraintObject
2 {
3     public function __construct($name, $tableName, $schemaName = null);
4     public function setName($name);
5     public function getName();
6     public function setSchemaName($schemaName);
7     public function getSchemaName();
8     public function getTableName();
9     public function setTableName($tableName);

```

```

10     public function setType($type);
11     public function getType();
12     public function hasColumns();
13     public function getColumns();
14     public function setColumns(array $columns);
15     public function getReferencedTableSchema();
16     public function setReferencedTableSchema($referencedTableSchema);
17     public function getReferencedTableName();
18     public function setReferencedTableName($referencedTableName);
19     public function getReferencedColumns();
20     public function setReferencedColumns(array $referencedColumns);
21     public function getMatchOption();
22     public function setMatchOption($matchOption);
23     public function getUpdateRule();
24     public function setUpdateRule($updateRule);
25     public function getDeleteRule();
26     public function setDeleteRule($deleteRule);
27     public function getCheckClause();
28     public function setCheckClause($checkClause);
29     public function isPrimaryKey();
30     public function isUnique();
31     public function isForeignKey();
32     public function isCheck();
33
34 }

```

The TriggerObject:

```

1  class Zend\Db\Metadata\Object\TriggerObject
2  {
3      public function getName();
4      public function setName($name);
5      public function getEventManipulation();
6      public function setEventManipulation($eventManipulation);
7      public function getEventObjectCatalog();
8      public function setEventObjectCatalog($eventObjectCatalog);
9      public function getEventObjectSchema();
10     public function setEventObjectSchema($eventObjectSchema);
11     public function getEventObjectTable();
12     public function setEventObjectTable($eventObjectTable);
13     public function getActionOrder();
14     public function setActionOrder($actionOrder);
15     public function getActionCondition();
16     public function setActionCondition($actionCondition);
17     public function getActionStatement();
18     public function setActionStatement($actionStatement);
19     public function getActionOrientation();
20     public function setActionOrientation($actionOrientation);
21     public function getActionTiming();
22     public function setActionTiming($actionTiming);
23     public function getActionReferenceOldTable();
24     public function setActionReferenceOldTable($actionReferenceOldTable);
25     public function getActionReferenceNewTable();
26     public function setActionReferenceNewTable($actionReferenceNewTable);
27     public function getActionReferenceOldRow();
28     public function setActionReferenceOldRow($actionReferenceOldRow);
29     public function getActionReferenceNewRow();
30     public function setActionReferenceNewRow($actionReferenceNewRow);

```



```
31     public function getCreated();  
32     public function setCreated($created);  
33 }
```

Dumping Variables

The static method `Zend\Debug\Debug::dump()` prints or returns information about an expression. This simple technique of debugging is common because it is easy to use in an ad hoc fashion and requires no initialization, special tools, or debugging environment.

Example of `dump()` method

```
Zend\Debug\Debug::dump($var, $label = null, $echo = true);
```

The `$var` argument specifies the expression or variable about which the `Zend\Debug\Debug::dump()` method outputs information.

The `$label` argument is a string to be prepended to the output of `Zend\Debug\Debug::dump()`. It may be useful, for example, to use labels if you are dumping information about multiple variables on a given screen.

The boolean `$echo` argument specifies whether the output of `Zend\Debug\Debug::dump()` is echoed or not. If `TRUE`, the output is echoed. Regardless of the value of the `$echo` argument, the return value of this method contains the output.

It may be helpful to understand that `Zend\Debug\Debug::dump()` method wraps the *PHP* function `var_dump()`. If the output stream is detected as a web presentation, the output of `var_dump()` is escaped using `htmlspecialchars()` and wrapped with (X)HTML `<pre>` tags.

Tip: Debugging with `ZendLog`

Using `Zend\Debug\Debug::dump()` is best for ad hoc debugging during software development. You can add code to dump a variable and then remove the code very quickly.

Also consider the *ZendLog* component when writing more permanent debugging code. For example, you can use the `DEBUG` log level and the *stream log writer* to output the string returned by `Zend\Debug\Debug::dump()`.

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:

```
1 $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.

Note: `Zend\Di` is an example of an Inversion of Control (IoC) container. IoC containers are widely used to create object instances that have all dependencies resolved and injected. Dependency Injection containers are one form of IoC – but not the only form.

Zend Framework 2 ships with another form of IoC as well, [ZendServiceManager](#). Unlike `Zend\Di`, The ServiceManager is code-driven, meaning that you typically tell it what class to instantiate, or provide a factory for the given class. This approach offers several benefits:

- Easier to debug (error stacks take you into your factories, not the dependency injection container).
- Easier to setup (write code to instantiate objects, instead of configuration).
- Faster (`Zend\Di` has known performance issues due to the approaches used).

Unless you have specific needs for a dependency injection container versus more general Inversion of Control, we recommend using `Zend\ServiceManager` for the above reasons.

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 102

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:

```
1 namespace MyLibrary
2 {
3     class DbAdapter
4     {
5         protected $username = null;
6         protected $password = null;
7         public function __construct($username, $password)
8         {
9             $this->username = $username;
10            $this->password = $password;
11        }
12    }
13 }
14
15 namespace MyMovieApp
16 {
17     class MovieFinder
18     {
19         protected $dbAdapter = null;
20         public function __construct(\MyLibrary\DbAdapter $dbAdapter)
21         {
22             $this->dbAdapter = $dbAdapter;
23         }
24     }
25
26     class MovieLister
27     {
28         protected $movieFinder = null;
29         public function __construct(MovieFinder $movieFinder)
```

```

30     {
31         $this->movieFinder = $movieFinder;
32     }
33 }
34 }

```

With the above code, you find yourself writing the following to wire and utilize this code:

```

1  // $config object is assumed
2
3  $dbAdapter = new MyLibrary\DbAdapter($config->username, $config->password);
4  $movieFinder = new MyMovieApp\MovieFinder($dbAdapter);
5  $movieLister = new MyMovieApp\MovieLister($movieFinder);
6  foreach ($movieLister as $movie) {
7      // iterate and display $movie
8  }

```

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:

```

1  // inside a bootstrap somewhere
2  $di = new Zend\Di\Di();
3  $di->instanceManager()->setParameters('MyLibrary\DbAdapter', array(
4      'username' => $config->username,
5      'password' => $config->password
6  ));
7
8  // inside each controller
9  $movieLister = $di->get('MyMovieApp\MovieLister');
10 foreach ($movieLister as $movie) {
11     // iterate and display $movie
12 }

```

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:

```

1  public function __construct(DefinitionList $definitions = null, InstanceManager
2  ↪ $instanceManager = null, Configuration $config = null)
3  {
4      $this->definitions = ($definitions) ? : new DefinitionList(new_
5  ↪ Definition\RuntimeDefinition());
6      $this->instanceManager = ($instanceManager) ? : new InstanceManager();
7
8      if ($config) {
9          $this->configure($config);
10     }
11 }

```

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()`:

```
1 // inside each controller
2 $di = new Zend\Di\Di();
3 $movieLister = $di->get('MyMovieApp\MovieLister', array(
4     'username' => $config->username,
5     'password' => $config->password
6 ));
7 foreach ($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()`.

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.

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.

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:

```
1 public function __construct (IntrospectionStrategy $introspectionStrategy = null,   
    ↪ array $explicitClasses = null)   
2 {   
3     $this->introspectionStrategy = ($introspectionStrategy) ?: new   
    ↪ IntrospectionStrategy();
```

```
4     if ($explicitClasses) {
5         $this->setExplicitClasses($explicitClasses);
6     }
7 }
```

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:

```
1 public function __construct (AnnotationManager $annotationManager = null)
2 {
3     $this->annotationManager = ($annotationManager) ?: $this->
4     ↪createDefaultAnnotationManager();
5 }
```

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.

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:

```
1 // in "package name" format
2 $components = array(
3     'My_MovieApp',
4     'My_OtherClasses',
5 );
6
7 foreach ($components as $component) {
8     $diCompiler = new Zend\Di\Definition\CompilerDefinition;
9     $diCompiler->addDirectory('/path/to/classes/' . str_replace('_', '/',
10 ↪$component));
11
12     $diCompiler->compile();
13     file_put_contents(
```

```

13     __DIR__ . '/../data/di/' . $component . '-definition.php',
14     '<?php return ' . var_export($diCompiler->toArrayDefinition()->toArray(), _
↪true) . ';'
15 );
16 }

```

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:

```

1  protected function setupDi(Application $app)
2  {
3      $definitionList = new DefinitionList(array(
4          new Definition\ArrayDefinition(include __DIR__ . '/path/to/data/di/My_
↪MovieApp-definition.php'),
5          new Definition\ArrayDefinition(include __DIR__ . '/path/to/data/di/My_
↪OtherClasses-definition.php'),
6          $runtime = new Definition\RuntimeDefinition(),
7      ));
8      $di = new Di($definitionList, null, new Config($this->config->di));
9      $di->instanceManager()->addTypePreference('Zend\Di\LocatorInterface', $di);
10     $app->setLocator($di);
11 }

```

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.

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

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.

Parameters

Parameters are simply entry points for either dependencies or instance configuration values. A class consists 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:

```
1 namespace MyLibrary
2 {
3     class DbAdapter
4     {
5         protected $username = null;
6         protected $password = null;
7         public function __construct($username, $password)
8         {
9             $this->username = $username;
10            $this->password = $password;
11        }
12    }
13 }
14
15 namespace MyMovieApp
16 {
17     class MovieFinder
18     {
19         protected $dbAdapter = null;
20         public function __construct(\MyLibrary\DbAdapter $dbAdapter)
```

```
21     {
22         $this->dbAdapter = $dbAdapter;
23     }
24 }
25
26 class MovieLister
27 {
28     protected $movieFinder = null;
29     public function __construct(MovieFinder $movieFinder)
30     {
31         $this->movieFinder = $movieFinder;
32     }
33 }
34 }
```

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:

```
1  // setting instance configuration into the instance manager
2  $di->instanceManager()->setParameters('MyLibrary\DbAdapter', array(
3      'username' => 'myusername',
4      'password' => 'mypassword'
5  ));
6
7  // forcing a particular dependency to be used by the instance manager
8  $di->instanceManager()->setParameters('MyMovieApp\MovieFinder', array(
9      'dbAdapter' => new MyLibrary\DbAdapter('myusername', 'mypassword')
10 ));
11
12 // passing instance parameters at call time
13 $movieLister = $di->get('MyMovieApp\MovieLister', array(
14     'username' => $config->username,
15     'password' => $config->password
16 ));
17
18 // passing a specific instance at call time
19 $movieLister = $di->get('MyMovieApp\MovieLister', array(
20     'dbAdapter' => new MyLibrary\DbAdapter('myusername', 'mypassword')
21 ));
```

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:


```

1 namespace MyMovieApp
2 {
3     interface MovieFinderInterface
4     {
5         // methods required for this type
6     }
7
8     class GenericMovieFinder implements MovieFinderInterface
9     {
10         protected $dbAdapter = null;
11         public function __construct(\MyLibrary\DbAdapter $dbAdapter)
12         {
13             $this->dbAdapter = $dbAdapter;
14         }
15     }
16
17     class MovieLister
18     {
19         protected $movieFinder = null;
20         public function __construct(MovieFinderInterface $movieFinder)
21         {
22             $this->movieFinder = $movieFinder;
23         }
24     }
25 }

```

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:

```

1 $di->instanceManager()->addTypePreference('MyMovieApp\MovieFinderInterface',
2     ↪ 'MyMovieApp\GenericMovieFinder');
3 // assuming all instance config for username, password is setup
4 $di->get('MyMovieApp\MovieLister');

```

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

```
1 // assume the MovieLister example of code from the QuickStart
2
3 $im = $di->instanceManager();
4
5 // add alias for short naming
6 $im->addAlias('movielister', 'MyMovieApp\MovieLister');
7
8 // add aliases for specific instances
9 $im->addAlias('dbadapter-readonly', 'MyLibrary\DbAdapter', array(
10     'username' => $config->db->readAdapter->username,
11     'password' => $config->db->readAdapter->password,
12 ));
13 $im->addAlias('dbadapter-readwrite', 'MyLibrary\DbAdapter', array(
14     'username' => $config->db->readWriteAdapter->username,
15     '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'));
```

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:

```

1 $config = array(
2     'definition' => array(
3         'compiler' => array(/* @todo compiler information */),
4         'runtime'  => array(/* @todo runtime information */),
5         'class' => array(
6             'instantiator' => '', // the name of the instantiator, by default this is
7             ↪__construct
8             'supertypes'  => array(), // an array of supertypes the class implements
9             'methods'    => array(
10                 'setSomeParameter' => array( // a method name
11                     'parameterName' => array(
12                         'name',          // string parameter name
13                         'type',         // type or null
14                         'is-required'  // bool
15                     )
16                 )
17             )
18         )
19     )
20 );

```

Zend\Di Debugging & Complex Use Cases

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:

```
1  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:

```
1  Zend\Di\Display\Console::export($di, array('A\ClassIWanto\GetTheDefinitionFor  
    ↪ '));
```

Complex Use Cases

Interface Injection

```
1  namespace Foo\Bar {  
2      class Baz implements BamAwareInterface  
3      {  
4          public $bam;  
5  
6          public function setBam(Bam $bam)  
7          {  
8              $this->bam = $bam;  
9          }  
10     }  
11     class Bam  
12     {
```

```
13     }
14     interface BamAwareInterface
15     {
16         public function setBam(Bam $bam);
17     }
18 }
19
20 namespace {
21     include 'zf2bootstrap.php';
22     $di = new Zend\Di\Di;
23     $baz = $di->get('Foo\Bar\Baz');
24 }
```

Setter Injection with Class Definition

```
1 namespace Foo\Bar {
2     class Baz
3     {
4         public $bam;
5
6         public function setBam(Bam $bam)
7         {
8             $this->bam = $bam;
9         }
10    }
11    class Bam {
12    }
13 }
14
15 namespace {
16     $di = new Zend\Di\Di;
17     $di->configure(new Zend\Di\Config(array(
18         'definition' => array(
19             'class' => array(
20                 'Foo\Bar\Baz' => array(
21                     'setBam' => array('required' => true)
22                 )
23             )
24         )
25     )));
26     $baz = $di->get('Foo\Bar\Baz');
27 }
```

Multiple Injections To A Single Injection Point

```
1 namespace Application {
2     class Page
3     {
4         public $blocks;
5
6         public function addBlock(Block $block)
7         {
8             $this->blocks[] = $block;
9         }
10    }
```

```

10     }
11     interface Block
12     {
13     }
14 }
15
16 namespace MyModule {
17     class BlockOne implements \Application\Block {}
18     class BlockTwo implements \Application\Block {}
19 }
20
21 namespace {
22     include 'zf2bootstrap.php';
23     $di = new Zend\Di\Di;
24     $di->configure(new Zend\Di\Config(array(
25         'definition' => array(
26             'class' => array(
27                 'Application\Page' => array(
28                     'addBlock' => array(
29                         'block' => array('type' => 'Application\Block', 'required' =>
30 true)
31                     )
32                 )
33             ),
34             'instance' => array(
35                 'Application\Page' => array(
36                     'injections' => array(
37                         'MyModule\BlockOne',
38                         'MyModule\BlockTwo'
39                     )
40                 )
41             )
42         )));
43     $page = $di->get('Application\Page');
44 }

```


CHAPTER 107

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.

`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).

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 `execute()` 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"`. `'<div bar="foo baz">'` would match, but `'<div bar="foo bazbat">'` 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 descendants:** utilize `'>'` between selectors to denote direct descendants. `'div > span'` would select only `'span'` elements that are direct descendants of a `'div'`. Can also be used with any of the selectors above.
- **descendants:** 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 descendant of arbitrary depth beneath a `'span'` element, which is in turn a descendant of arbitrary depth beneath an element with a class of `'foo'`, that is an descendant of arbitrary depth beneath a `'div'` element. For example, it would match the link to the word `'One'` in the listing below:

```
1 <div>
2 <table>
3   <tr>
4     <td class="foo">
5       <div>
6         Lorem ipsum <span class="bar">
7           <a href="/foo/bar" id="one">One</a>
8           <a href="/foo/baz" id="two">Two</a>
9           <a href="/foo/bat" id="three">Three</a>
10          <a href="/foo/bla" id="four">Four</a>
11        </span>
12      </div>
13    </td>
14  </tr>
15 </table>
16 </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:

```
1 use Zend\Dom\Query;
2
3 $dom = new Query($html);
4 $results = $dom->execute('.foo .bar a');
5
6 $count = count($results); // get number of matches: 4
7 foreach ($results as $result) {
8     // $result is a DOMELEMENT
9 }
```

`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.

Methods Available

The `Zend\Dom\Query` family of classes have the following methods available.

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.

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.

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* using `zendframework/zend-escaper`.

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.

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.

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 *ZendFilter component*, *HTMLPurifier* or PHP's *Filter* component should be used.

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 its 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 its 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:

```

1 $escaper = new Zend\Escaper\Escaper('utf-8');
2
3 // <script>alert("zf2")</script>;
4 echo $escaper->escapeHtml('<script>alert("zf2")</script>');
5 // <script>alert(&#x28;&quot;zf2&quot;,&#x29;&lt;&#x2F;script&gt;;
6 echo $escaper->escapeHtmlAttr('<script>alert("zf2")</script>');
7 // \x3Cscript\x3Ealert\x28\x22zf2\x22\x29\x3C\x2Fscript\x3E
8 echo $escaper->escapeJs('<script>alert("zf2")</script>');
9 // \3C script\3E alert\28 \22 zf2\22 \29 \3C \2F script\3E
10 echo $escaper->escapeCss('<script>alert("zf2")</script>');
11 // %3Cscript%3Ealert%28%22zf2%22%29%3C%2Fscript%3E
12 echo $escaper->escapeUrl('<script>alert("zf2")</script>');
```

More detailed examples will be given in later chapters.

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.

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!):

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.

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!

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).

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.

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 112

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.

```
1 // outputting this without escaping would be a bad idea!
2 $input = '<script>alert("zf2")</script>';
3
4 $escaper = new Zend\Escaper\Escaper('utf-8');
5
6 // somewhere in an HTML template
7 <div class="user-provided-input">
8     <?php
9         echo $escaper->escapeHtml($input); // all safe!
10     ?>
11 </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!

Examples of Bad HTML Escaping

An example of incorrect usage:

```
1 <?php
2 $input = '<script>alert("zf2")</script>';
3 $escaper = new Zend\Escaper\Escaper('utf-8');
4 ?>
5 <?php header('Content-Type: text/html; charset=ISO-8859-1'); ?>
6 <!DOCTYPE html>
7 <html>
```

```
8 <head>
9     <title>Encodings set incorrectly!</title>
10     <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
11 </head>
12 <body>
13     <?php
14         // Bad! The escaper's and the document's encodings are different!
15         echo $escaper->escapeHtml($input);
16     ?>
17 </body>
```

Examples of Good HTML Escaping

An example of correct usage:

```
1 <?php
2 $input = '<script>alert("zf2")</script>';
3 $escaper = new Zend\Escaper\Escaper('utf-8');
4 ?>
5 <?php header('Content-Type: text/html; charset=UTF-8'); ?>
6 <!DOCTYPE html>
7 <html>
8 <head>
9     <title>Encodings set correctly!</title>
10     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
11 </head>
12 <body>
13     <?php
14         // Good! The escaper's and the document's encodings are same!
15         echo $escaper->escapeHtml($input);
16     ?>
17 </body>
```

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 its 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.

Examples of Bad HTML Attribute Escaping

An example of incorrect HTML attribute escaping:

```
1 <?php header('Content-Type: text/html; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 ' onmouseover='alert(/ZF2!/);
6 INPUT;
7 /**
8  * NOTE: This is equivalent to using htmlspecialchars($input, ENT_COMPAT)
9  */
10 $output = htmlspecialchars($input);
11 ?>
12 <html>
13 <head>
14     <title>Single Quoted Attribute</title>
15     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
16 </head>
17 <body>
18     <div>
19         <?php
20             // the span tag will look like:
```

```
21         // <span title='' onmouseover='alert(/ZF2!/);'>
22         ?>
23         <span title='<?php echo $output ?>'>
24             What framework are you using?
25         </span>
26     </div>
27 </body>
28 </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:

```
1 <?php header('Content-Type: text/html; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 faketitle onmouseover=alert(/ZF2!/);
6 INPUT;
7 // Tough luck using proper flags when the title attribute is unquoted!
8 $output = htmlspecialchars($input, ENT_QUOTES);
9 ?>
10 <html>
11 <head>
12     <title>Quoteless Attribute</title>
13     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
14 </head>
15 <body>
16     <div>
17         <?php
18         // the span tag will look like:
19         // <span title=faketitle onmouseover=alert(/ZF2!/);>
20         ?>
21         <span title=<?php echo $output ?>>
22             What framework are you using?
23         </span>
24     </div>
25 </body>
26 </html>
```

The above example shows how it is easy to break out from unquoted attributes in HTML5.

Examples of Good HTML Attribute Escaping

Both of the previous examples can be avoided by simply using the `escapeHtmlAttr` method:

```
1 <?php header('Content-Type: text/html; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 faketitle onmouseover=alert(/ZF2!/);
6 INPUT;
7 $escaper = new Zend\Escaper\Escaper('utf-8');
8 $output = $escaper->escapeHtmlAttr($input);
```

```
9  ?>
10 <html>
11 <head>
12     <title>Quoteless Attribute</title>
13     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
14 </head>
15 <body>
16     <div>
17         <?php
18             // the span tag will look like:
19             // <span title=faketitle&#x20;onmouseover&#x3D;alert&#x28;&#x2F;ZF2&#x21;&
20             ↪#x2F;&#x29;&#x3B;>
21             ?>
22             <span title=<?php echo $output ?>>
23                 What framework are you using?
24             </span>
25     </div>
26 </body>
</html>
```

In the above example, the malicious input from the attacker becomes completely harmless as we used proper HTML attribute escaping!

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.

Examples of Bad Javascript Escaping

An example of incorrect Javascript escaping:

```
1 <?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 bar&quot;; alert(&quot;Meow!&quot;); var xss=&quot;true
6 INPUT;
7 $output = json_encode($input);
8 ?>
9 <html xmlns="http://www.w3.org/1999/xhtml">
10 <head>
11     <title>Unescaped Entities</title>
12     <meta charset="UTF-8"/>
13     <script type="text/javascript">
14         <?php
15             // this will result in
16             // var foo = "bar&quot;; alert(&quot;Meow!&quot;); var xss=&quot;true";
```

```

17     ?>
18     var foo = <?php echo $output ?>;
19     </script>
20 </head>
21 <body>
22     <p>json_encode() is not good for escaping javascript!</p>
23 </body>
24 </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.

Examples of Good Javascript Escaping

By using the `escapeJs` method in the Javascript context, such attacks can be prevented:

```

1  <?php header('Content-Type: text/html; charset=UTF-8'); ?>
2  <!DOCTYPE html>
3  <?php
4  $input = <<<INPUT
5  bar"; alert(&quot;Meow!&quot;); var xss=&quot;true
6  INPUT;
7  $escaper = new Zend\Escaper\Escaper('utf-8');
8  $output = $escaper->escapeJs($input);
9  ?>
10 <html xmlns="http://www.w3.org/1999/xhtml">
11 <head>
12     <title>Escaped Entities</title>
13     <meta charset="UTF-8"/>
14     <script type="text/javascript">
15         <?php
16             // this will look like
17             // var foo =
18             ↪bar\x26quot\x3B\x3B\x20alert\x28\x26quot\x3BMeow\x21\x26quot\x3B\x29\x3B\x20var\x20xss\x3D\x26quot
19             ↪
20             ?>
21             var foo = <?php echo $output ?>;
22         </script>
23 </head>
24 <body>
25     <p>Zend\Escaper\Escaper::escapeJs() is good for escaping javascript!</p>
26 </body>
27 </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.

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.

Examples of Bad CSS Escaping

In most cases developers forget to escape CSS completely:

```
1 <?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 body {
6     background-image: url('http://example.com/foo.jpg?</style><script>alert(1)</
7     ↪script>');
8 }
9 INPUT;
10 ?>
11 <html xmlns="http://www.w3.org/1999/xhtml">
12 <head>
13     <title>Unescaped CSS</title>
14     <meta charset="UTF-8"/>
15     <style>
16         <?php echo $input; ?>
17     </style>
18 </head>
19 <body>
20     <p>User controlled CSS needs to be properly escaped!</p>
21 </body>
22 </html>
```

In the above example, by failing to escape the user provided CSS, an attacker can execute an XSS attack fairly easily.

Examples of Good CSS Escaping

By using `escapeCss` method in the CSS context, such attacks can be prevented:

```
1 <?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 body {
6     background-image: url('http://example.com/foo.jpg?</style><script>alert(1)</
7     <script>');
8 }
9 INPUT;
10 $escaper = new Zend\Escaper\Escaper('utf-8');
11 $output = $escaper->escapeCss($input);
12 ?>
13 <html xmlns="http://www.w3.org/1999/xhtml">
14 <head>
15     <title>Escaped CSS</title>
16     <meta charset="UTF-8"/>
17     <style>
18         <?php
19         // output will look something like
20         // body\20 \7B \A \20 \20 \20 \20 background\2D image\3A \20 url\28 ...
21         echo $output;
22         ?>
23     </style>
24 </head>
25 <body>
26     <p>User controlled CSS needs to be properly escaped!</p>
27 </body>
28 </html>
```

By properly escaping user controlled CSS, we can prevent XSS attacks in our web applications.

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.

Examples of Bad URL Escaping

XSS attacks are easy if data inserted into URLs is not escaped properly:

```
1 <?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 " onmouseover="alert('zf2')
6 INPUT;
7 ?>
8 <html xmlns="http://www.w3.org/1999/xhtml">
9 <head>
10     <title>Unescaped URL data</title>
11     <meta charset="UTF-8"/>
12 </head>
13 <body>
14     <a href="http://example.com/?name=<?php echo $input; ?>">Click here!</a>
15 </body>
16 </html>
```

Examples of Good URL Escaping

By properly escaping data in URLs by using `escapeUrl`, we can prevent XSS attacks:

```
1 <?php header('Content-Type: application/xhtml+xml; charset=UTF-8'); ?>
2 <!DOCTYPE html>
3 <?php
4 $input = <<<INPUT
5 " onmouseover="alert('zf2')
6 INPUT;
7 $escaper = new Zend\Escaper\Escaper('utf-8');
8 $output = $escaper->escapeUrl($input);
9 ?>
10 <html xmlns="http://www.w3.org/1999/xhtml">
11 <head>
12     <title>Unescaped URL data</title>
13     <meta charset="UTF-8"/>
14 </head>
15 <body>
16     <a href="http://example.com/?name=<?php echo $output; ?>">Click here!</a>
17 </body>
18 </html>
```

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.

Quick Start

Typically, you will compose an `EventManager` instance in a class.

```
1 use Zend\EventManager\EventManagerInterface;
2 use Zend\EventManager\EventManager;
3 use Zend\EventManager\EventManagerAwareInterface;
4
5 class Foo implements EventManagerAwareInterface
6 {
7     protected $events;
8
9     public function setEventManager(EventManagerInterface $events)
10     {
11         $events->setIdentifiers(array(
12             __CLASS__,
13             get_called_class(),
14         ));
15         $this->events = $events;
16         return $this;
```

```
17     }
18
19     public function getEventManager()
20     {
21         if (null === $this->events) {
22             $this->setEventManager(new EventManager());
23         }
24         return $this->events;
25     }
26 }
```

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.

The instance property *events* is a convention within Zend Framework 2 to refer to the `EventManager` instance.

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 target, which is usually the current object instance; - The arguments, which are usually the arguments provided to the current function/method.

An optional fourth argument; a callback may also be supplied.

```
1 class Foo
2 {
3     // ... assume events definition from above
4
5     public function bar($baz, $bat = null)
6     {
7         $params = compact('baz', 'bat');
8         $this->getEventManager()->trigger(__FUNCTION__, $this, $params);
9     }
10 }
```

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, target, and parameters. Let's add a listener, and trigger the event.

```
1 use Zend\Log\Factory as LogFactory;
2
3 $log = LogFactory($someConfig);
4 $foo = new Foo();
5 $foo->getEventManager()->attach('bar', function ($e) use ($log) {
6     $event = $e->getName();
7     $target = get_class($e->getTarget());
8     $params = json_encode($e->getParams());
9
10    $log->info(sprintf(
11        '%s called on %s, using params %s',
12        $event,
13        $target,
14        $params
15    ));
16 });
17
18 // Results in log message:
```

```

19 $foo->bar('baz', 'bat');
20 // 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 `SharedEventManager`.

Simply put, you can inject individual `EventManager` instances with a well-known `SharedEventManager`, and the `EventManager` instance will query it for additional listeners.

Listeners attach to a `SharedEventManager` in roughly the same way they do to 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 `SharedEventManager`.

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:

```

1  use Zend\Log\Factory as LogFactory;
2
3  // Assume $events is a Zend\EventManager\SharedEventManager instance
4
5  $log = LogFactory($someConfig);
6  $events->attach('Foo', 'bar', function ($e) use ($log) {
7      $event = $e->getName();
8      $target = get_class($e->getTarget());
9      $params = json_encode($e->getParams());
10
11     $log->info(sprintf(
12         '%s called on %s, using params %s',
13         $event,
14         $target,
15         $params
16     ));
17 });
18
19 // Later, instantiate Foo:
20 $foo = new Foo();
21 $foo->getEventManager()->setSharedManager($events);
22
23 // And we can still trigger the above event:
24 $foo->bar('baz', 'bat');
25 // results in log message:
26 // 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 `SharedEventManager`. As such, you do not need to worry about where and how to get access to the `SharedEventManager`; 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.

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

```
1 $events = new EventManager();
2 $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

```
1 $events = new EventManager();
2 $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

```
1 $events = new SharedEventManager();
2 // Attach to many events on the context "foo"
3 $events->attach('foo', array('these', 'are', 'event', 'names'), $callback);
4
5 // Attach to many events on the contexts "foo" and "bar"
6 $events->attach(array('foo', 'bar'), array('these', 'are', 'event', 'names'),
    ↳$callback);
```

Note that if you specify a priority, that priority will be used for all events specified.

Attaching using the wildcard via a SharedEventManager

```
1 $events = new SharedEventManager();
2 // Attach to all events on the context "foo"
3 $events->attach('foo', '*', $callback);
4
5 // Attach to all events on the contexts "foo" and "bar"
6 $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.

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.

Available Methods

__construct `__construct (null|string|int|array|Traversable $identifiers)`

Constructs a new `EventManager` instance, using the given identifiers, if provided, for purposes of use with a `SharedEventManager` instance.

setEventClass `setEventClass (string $class)`

Provide the name of an alternate `Event` class to use when creating events to pass to triggered listeners.

setSharedManager `setSharedManager (SharedEventManagerInterface $sharedEventManager)`

Accepts an instance of `SharedEventManagerInterface` and sets the `sharedManager` property.

unsetSharedManager `unsetSharedManager ()`

Removes the `SharedEventManager` currently attached to the `EventManager` instance.

getSharedManager `getSharedManager ()`

Returns the static `SharedEventManager` is one has been assigned or will return false.

getIdentifiers `getIdentifiers ()`

Returns the identifiers that have been set for this `EventManager` instance

setIdentifiers `setIdentifiers (string|int|array|Traversable $identifiers)`

Sets the identifiers for this `EventManager` instance. This method will clear any existing identifiers that have been previously added to the instance.

addIdentifiers `addIdentifiers (string|int|array|Traversable $identifiers)`

Add identifiers to the `EventManager` instance. This method will merge the identifiers with any that have previously been added to the instance.

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 your listeners to be able to modify arguments such that later listeners or the triggering method can see the changes.

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.

```

1  class ValueObject
2  {
3      // assume a composed event manager
4
5      function inject(array $values)
6      {
7          $argv = compact('values');
8          $argv = $this->getEventManager()->prepareArgs($argv);
9          $this->getEventManager()->trigger(__FUNCTION__, $this, $argv);
10         $date = isset($argv['values']['date']) ? $argv['values']['date'] : new
↪DateTime('now');
11
12         // ...
13     }
14 }
15
16 $v = new ValueObject();
17
18 $v->getEventManager()->attach('inject', function($e) {
19     $values = $e->getParam('values');
20     if (!$values) {
21         return;
22     }
23
24     if (!isset($values['date'])) {
25         $values['date'] = new \DateTime('now');
26     } else {
27         $values['date'] = new \Datetime($values['date']);
28     }
29
30     $e->setParam('values', $values);
31 });
32
33 $v->inject(array(
34     'date' => '2011-08-10 15:30:29',
35 ));

```

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.

```

1  $listener = function($e) {
2      // do some work
3

```

```
4 // Stop propagation and return a response
5 $e->stopPropagation(true);
6 return $response;
7 };
```

Alternately, we could do the check from the method triggering the event.

```
1 class Foo implements DispatchableInterface
2 {
3     // assume composed event manager
4
5     public function dispatch(Request $request, Response $response = null)
6     {
7         $argv = compact('request', 'response');
8         $results = $this->getEventManager()->triggerUntil(__FUNCTION__, $this, $argv,
9 ↪function($v) {
10             return ($v instanceof Response);
11         });
12 }
```

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:

```
1 class Foo implements DispatchableInterface
2 {
3     // assume composed event manager
4
5     public function dispatch(Request $request, Response $response = null)
6     {
7         $argv = compact('request', 'response');
8         $results = $this->getEventManager()->triggerUntil(__FUNCTION__, $this, $argv,
9 ↪function($v) {
10             return ($v instanceof Response);
11         });
12
13         // Test if execution was halted, and return last result:
14         if ($results->stopped()) {
15             return $results->last();
16         }
17
18         // continue...
19 }
```

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:

```

1 class SomeValueObject
2 {
3     // assume it composes an event manager
4
5     public function get($id)
6     {
7         $params = compact('id');
8         $results = $this->getEventManager()->trigger('get.pre', $this, $params);
9
10        // If an event stopped propagation, return the value
11        if ($results->stopped()) {
12            return $results->last();
13        }
14
15        // do some work...
16
17        $params['__RESULT__'] = $someComputedContent;
18        $this->getEventManager()->trigger('get.post', $this, $params);
19    }
20 }
```

Now, let's create a ListenerAggregateInterface that can handle caching for us:

```

1 use Zend\Cache\Cache;
2 use Zend\EventManager\EventManagerInterface;
3 use Zend\EventManager\ListenerAggregateInterface;
4 use Zend\EventManager\EventInterface;
5
6 class CacheListener implements ListenerAggregateInterface
7 {
8     protected $cache;
9
10    protected $listeners = array();
11
12    public function __construct(Cache $cache)
13    {
14        $this->cache = $cache;
15    }
16
17    public function attach(EventManagerInterface $events)
18    {
19        $this->listeners[] = $events->attach('get.pre', array($this, 'load'), 100);
20        $this->listeners[] = $events->attach('get.post', array($this, 'save'), -100);
21    }
22
23    public function detach(EventManagerInterface $events)
24    {
25        foreach ($this->listeners as $index => $listener) {
26            if ($events->detach($listener)) {
27                unset($this->listeners[$index]);
28            }
29        }
30    }
31
32    public function load(EventInterface $e)
```

```
33 {
34     $id = get_class($e->getTarget()) . '-' . json_encode($e->getParams());
35     if (false !== ($content = $this->cache->load($id))) {
36         $e->stopPropagation(true);
37         return $content;
38     }
39 }
40
41 public function save(EventInterface $e)
42 {
43     $params = $e->getParams();
44     $content = $params['__RESULT__'];
45     unset($params['__RESULT__']);
46
47     $id = get_class($e->getTarget()) . '-' . json_encode($params);
48     $this->cache->save($content, $id);
49 }
50 }
```

We can then attach the aggregate to an instance.

```
1 $value = new SomeValueObject();
2 $cacheListener = new CacheListener($cache);
3 $value->getEventManager()->attachAggregate($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.

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

```
1 // Fetch the latest Slashdot headlines
2 try {
3     $slashdotRss =
4         Zend\Feed\Reader\Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
5 } catch (Zend\Feed\Reader\Exception\RuntimeException $e) {
6     // feed import failed
7     echo "Exception caught importing feed: ($e->getMessage())\n";
8     exit;
9 }
10
```

```
11 // Initialize the channel/feed data array
12 $channel = array(
13     'title'      => $slashdotRss->getTitle(),
14     'link'       => $slashdotRss->getLink(),
15     'description' => $slashdotRss->getDescription(),
16     'items'     => array()
17 );
18
19 // Loop over each channel item/entry and store relevant data for each
20 foreach ($slashdotRss as $item) {
21     $channel['items'][] = array(
22         'title'      => $item->getTitle(),
23         'link'       => $item->getLink(),
24         'description' => $item->getDescription()
25     );
26 }
```

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 119

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:

```
1 $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:

```
1 // importing a feed from a text file
2 $feedFromFile = Zend\Feed\Reader\Reader::importFile('feed.xml');
3
4 // importing a feed from a PHP string variable
5 $feedFromPHP = Zend\Feed\Reader\Reader::importString($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.

Dumping the contents of a feed

To dump the contents of a `Zend\Feed\Reader\Feed\AbstractFeed` instance, you may use the `saveXml()` method.

```
1 assert($feed instanceof Zend\Feed\Reader\Feed\AbstractFeed);
2
3 // dump the feed to standard output
4 print $feed->saveXml();
```

Retrieving Feeds from Web Pages

Find Feed Links

Web pages often contain **<link>** tags that refer to feeds with content relevant to the particular page. `Zend\Fee\Reader\Reader` enables you to retrieve all feeds referenced by a web page with one simple method call:

```
1 $feedLinks = Zend\Fee\Reader\Reader::findFeedLinks('http://www.example.com/news.html  
  ↪');
```

Here the `findFeedLinks()` method returns a `Zend\Fee\Reader\FeedSet` object, that is in turn, a collection of other `Zend\Fee\Reader\FeedSet` objects, that are referenced by **<link>** tags on the `news.html` web page. `Zend\Fee\Reader\Reader` will throw a `Zend\Fee\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:

```
1 $rssFeed = null;  
2 $feedLinks = Zend\Fee\Reader\Reader::findFeedLinks('http://www.example.com/news.html  
  ↪');  
3 foreach ($feedLinks as $link) {  
4     if (stripos($link['type'], 'application/rss+xml') !== false) {  
5         $rssFeed = $link['href'];  
6         break;  
7     }  
}
```

Each `Zend\Fee\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:

```
1 $rssFeed = null;  
2 $feedLinks = Zend\Fee\Reader\Reader::findFeedLinks('http://www.example.com/news.html  
  ↪');  
3 $firstAtomFeed = $feedLinks->atom;
```


CHAPTER 121

Consuming an RSS Feed

Reading a feed

Reading an *RSS* feed is as simple as passing the *URL* of the feed to `Zend\Fee\Reader\Reader`'s `import` method.

```
1 $channel = Zend\Fee\Reader\Reader::import('http://rss.example.com/channelName');
```

If any errors occur fetching the feed, a `Zend\Fee\Reader\Exception\RuntimeException` will be thrown.

Get properties

Once you have a feed object, you can access any of the standard *RSS* “channel” properties directly on the object:

```
1 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.

```
1 $author = $channel->getAuthor();  
2 echo $author['name'];
```

Most commonly you'll want to loop through the feed and do something with its entries. `Zend\Fee\Reader\Feed\Rss` internally converts all entries to a `Zend\Fee\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:

```
1 foreach ($channel as $item) {  
2     echo $item->getTitle() . "\n";  
3 }
```

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:

```
1 if ($item->getPropname()) {  
2     // ... proceed.  
3 }
```

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.

Please see the official [RSS 2.0 specification](#) for further information.

Consuming an Atom Feed

`Zend\Food\Reader\Food\Atom` is used in much the same way as `Zend\Food\Reader\Food\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:

```
1 $feed = Zend\Food\Reader\Reader::import('http://atom.example.com/feed/');
2 echo 'The feed contains ' . $feed->count() . ' entries.' . "\n\n";
3 foreach ($feed as $entry) {
4     echo 'Title: ' . $entry->getTitle() . "\n";
5     echo 'Description: ' . $entry->getDescription() . "\n";
6     echo 'URL: ' . $entry->getLink() . "\n\n";
7 }
```

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/>.

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

```
1 $entry = Zend\Feed\Reader\Reader::import('http://atom.example.com/feed/1');
2 echo 'Entry title: ' . $entry->getTitle();
```


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...).

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.

```
1 // Setting HTMLPurifier's options
2 $options = array(
3     // Allow only paragraph tags
4     // and anchor tags wit the href attribute
5     array(
6         'HTML.Allowed',
7         'p,a[href]'
8     ),
```

¹ <http://tools.ietf.org/html/rfc4287#section-8.1>

```
9      // Format end output with Tidy
10     array(
11         'Output.TidyFormat',
12         true
13     ),
14     // Assume XHTML 1.0 Strict Doctype
15     array(
16         'HTML.Doctype',
17         'XHTML 1.0 Strict'
18     ),
19     // Disable cache, but see note after the example
20     array(
21         'Cache.DefinitionImpl',
22         null
23     )
24 );
25
26 // Configuring HTMLPurifier
27 $config = HTMLPurifier_Config::createDefault();
28 foreach ($options as $option) {
29     $config->set($option[0], $option[1]);
30 }
31
32 // Creating a HTMLPurifier with it's config
33 $purifier = new HTMLPurifier($config);
34
35 // Fetch the RSS
36 try {
37     $rss = Zend\Fee\Reader\Reader::import('http://www.planet-php.net/rss/');
38 } catch (Zend\Fee\Exception\Reader\RuntimeException $e) {
39     // feed import failed
40     echo "Exception caught importing feed: {$e->getMessage()}\n";
41     exit;
42 }
43
44 // Initialize the channel data array
45 // See that we're cleaning the description with HTMLPurifier
46 $channel = array(
47     'title'      => $rss->getTitle(),
48     'link'       => $rss->getLink(),
49     'description' => $purifier->purify($rss->getDescription()),
50     'items'      => array()
51 );
52
53 // Loop over each channel item and store relevant data
54 // See that we're cleaning the descriptions with HTMLPurifier
55 foreach ($rss as $item) {
56     $channel['items'][] = array(
57         'title'      => $item->getTitle(),
58         'link'       => $item->getLink(),
59         'description' => $purifier->purify($item->getDescription())
60     );
61 }
```

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.

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 outputting any potentially insecure data to the users.

```

1  try {
2      $rss = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rss/');
3  } catch (Zend\Feed\Exception\Reader\RuntimeException $e) {
4      // feed import failed
5      echo "Exception caught importing feed: {$e->getMessage()}\n";
6      exit;
7  }
8
9  // Validate all URIs
10 $linkValidator = new Zend\Validator\Uri;
11 $link = null;
12 if ($linkValidator->isValid($rss->getLink())) {
13     $link = $rss->getLink();
14 }
15
16 // Escaper used for escaping data
17 $escaper = new Zend\Escaper\Escaper('utf-8');
18
19 // Initialize the channel data array
20 $channel = array(
21     'title'      => $escaper->escapeHtml($rss->getTitle()),
22     'link'       => $escaper->escapeHtml($link),
23     'description' => $escaper->escapeHtml($rss->getDescription()),
24     'items'      => array()
25 );
26
27 // Loop over each channel item and store relevant data
28 foreach ($rss as $item) {
29     $link = null;
30     if ($linkValidator->isValid($rss->getLink())) {
31         $link = $item->getLink();
32     }
33     $channel['items'][] = array(
34         'title'      => $escaper->escapeHtml($item->getTitle()),
35         'link'       => $escaper->escapeHtml($link),
36         'description' => $escaper->escapeHtml($item->getDescription())
37     );
38 }

```

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.

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`.

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(),
5     'dateModified' => $feed->getDateModified(),
6     'description' => $feed->getDescription(),
7     'language'   => $feed->getLanguage(),
8     'entries'    => array(),
9 );
10
11 foreach ($feed as $entry) {
12     $edata = array(
13         'title'      => $entry->getTitle(),
14         'description' => $entry->getDescription(),
15         'dateModified' => $entry->getDateModified(),
16         'authors'    => $entry->getAuthors(),
17         'link'       => $entry->getLink(),
18         'content'    => $entry->getContent()
19     );
20     $data['entries'][] = $edata;
21 }
```

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.

```
1 // from a URI
2 $feed = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rdf/');
3
4 // from a String
5 $feed = Zend\Feed\Reader\Reader::importString($feedXmlString);
6
7 // from a file
8 $feed = Zend\Feed\Reader\Reader::importFile('./feed.xml');
```

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.

```

1 $feed = Zend\Feed\Reader\Reader::import('http://www.planet-php.net/rdf/');
2 $xpathPrefix = $feed->getXpathPrefix();
3 $xpath = $feed->getXpath();
4 $xpath->registerNamespace('admin', 'http://webns.net/mvcb/');
5 $reportErrorsTo = $xpath->evaluate('string('
6     . $xpathPrefix
7     . '/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.

Cache Support and Intelligent Requests

Adding Cache Support to `Zend\Feed\Reader\Reader`

`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*.

```

1 $cache = Zend\Cache\StorageFactory::adapterFactory('Memory');
2
3 Zend\Feed\Reader\Reader::setCache($cache);
```

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 *URIs* 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`.

```
1 $cache = Zend\Cache\StorageFactory::adapterFactory('Memory');
2
3 Zend\Feed\Reader\Reader::setCache($cache);
4 Zend\Feed\Reader\Reader::useHttpConditionalGet();
5
6 $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.

```
1 $lastEtagReceived = '5e6cefe7df5a7e95c8b1bala2ccaff3d';
2 $lastModifiedDateReceived = 'Wed, 08 Jul 2009 13:37:22 GMT';
3 $feed = Zend\Feed\Reader\Reader::import(
4     $uri, $lastEtagReceived, $lastModifiedDateReceived
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”.

```
1 $links = Zend\Feed\Reader\Reader::findFeedLinks('http://www.planet-php.net');
2
3 if (isset($links->rdf)) {
4     echo $links->rdf, "\n"; // http://www.planet-php.org/rdf/
5 }
6 if (isset($links->rss)) {
7     echo $links->rss, "\n"; // http://www.planet-php.org/rss/
8 }
9 if (isset($links->atom)) {
10    echo $links->atom, "\n"; // http://www.planet-php.org/atom/
11 }
```

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.

```

1 $links = Zend\Feed\Reader::findFeedLinks('http://www.planet-php.net');
2
3 foreach ($links as $link) {
4     echo $link['href'], "\n";
5 }

```

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 *URIs*). 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:

```

1 $feed = Zend\Feed\Reader\Reader::import('http://www.example.com/atom.xml');
2 $categories = $feed->getCategories();
3 $labels = array();
4 foreach ($categories as $cat) {
5     $labels[] = $cat['label'];
6 }

```

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 *URIs*). The simple array is flat (just values) and passed through `array_unique()` to remove duplication.

```

1 $feed = Zend\Feed\Reader\Reader::import('http://www.example.com/atom.xml');
2 $categories = $feed->getCategories();
3 $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.

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 125.1: 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 <code>ZendFeedReaderCollectionAuthor</code> which is an <code>ArrayObject</code> 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 <code>\$index</code> parameter any specific index on the array of Authors as described above (returning <code>NULL</code> 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 <code>DateTime</code> object.
getDateModified()	Returns the date on which this feed was last modified. The returned date will be a <code>DateTime</code> object.
getLastBuildDate()	Returns the date on which this feed was last built. The returned date will be a <code>DateTime</code> object. This is only supported by RSS - Atom feeds will always return <code>NULL</code> .
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 <code>ZendFeedReaderCollectionCategory</code> 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 <code>NULL</code> .
getImage()	Returns an array containing data relating to any feed image or logo, or <code>NULL</code> 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\FeeedReader\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\FeeedReader\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.

Table 125.2: Extended Feed Level API Methods

<code>getDomDocument()</code>	Returns the parent <code>DOMDocument</code> object for the entire source XML document
<code>getElement()</code>	Returns the current feed level <code>DOMElement</code> object
<code>saveXml()</code>	Returns a string containing an XML document of the entire feed element (this is not the original document but a rebuilt version)
<code>getXpath()</code>	Returns the <code>DOMXPath</code> object used internally to run queries on the <code>DOMDocument</code> object (this includes core and Extension namespaces pre-registered)
<code>getXpathPrefix()</code>	Returns the valid DOM path prefix prepended to all XPath queries matching the feed being queried
<code>getEncoding()</code>	Returns the encoding of the source XML document (note: this cannot account for errors such as the server sending documents in a different encoding). Where not defined, the default UTF-8 encoding of Unicode is applied.
<code>count()</code>	Returns a count of the entries or items this feed contains (implements <code>SPLCountable</code> interface)
<code>current()</code>	Returns either the current entry (using the current index from <code>key()</code>)
<code>key()</code>	Returns the current entry index
<code>next()</code>	Increments the entry index value by one
<code>rewind()</code>	Resets the entry index to 0
<code>valid()</code>	Checks that the current entry index is valid, i.e. it does fall below 0 and does not exceed the number of entries existing.
<code>getExtensions()</code>	Returns an array of all Extension objects loaded for the current feed (note: both feed-level and entry-level Extensions exist, and only feed-level Extensions are returned here). The array keys are of the form <code>{ExtensionName}_Feed</code> .
<code>getExtension(string \$name)</code>	Returns an Extension object for the feed registered under the provided name. This allows more fine-grained access to Extensions which may otherwise be hidden within the implementation of the standard API methods.
<code>getType()</code>	Returns a static class constant (e.g. <code>ZendFeedReaderReader::TYPE_ATOM_03</code> , i.e. Atom 0.3) indicating exactly what kind of feed is being consumed.

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 125.3: Entry Level API Methods

<code>getId()</code>	Returns a unique ID for the current entry.
<code>getTitle()</code>	Returns the title of the current entry.
<code>getDescription()</code>	Returns a description of the current entry.
<code>getLink()</code>	Returns a URI to the HTML version of the current entry.
<code>getPermaLink()</code>	Returns the permanent link to the current entry. In most cases, this is the same as using <code>getLink()</code> .
<code>getAuthors()</code>	Returns an object of type <code>ZendFeedReaderCollectionAuthor</code> which is an <code>ArrayObject</code> 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.
<code>getAuthor(integer \$index = 0)</code>	Returns either the first author known, or with the optional <code>\$index</code> parameter any specific index on the array of Authors as described above (returning <code>NULL</code> if an invalid index).
<code>getDateCreated()</code>	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.
<code>getDateModified()</code>	Returns the date on which the current entry was last modified
<code>getContent()</code>	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.
<code>getEnclosure()</code>	Returns an array containing the value of all attributes from a multi-media <code><enclosure></code> 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.
<code>getCommentCount()</code>	Returns the number of comments made on this entry at the time the feed was last generated
<code>getCommentLink()</code>	Returns a URI pointing to the HTML page where comments can be made on this entry
<code>getComment-FeedLink([string \$type = 'atom' 'rss'])</code>	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).
<code>getCategories()</code>	Returns a <code>ZendFeedReaderCollectionCategory</code> 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 <code>NULL</code> .

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.

Table 125.4: Extended Entry Level API Methods

<code>getDomDocument()</code>	Returns the parent <code>DOMDocument</code> object for the entire feed (not just the current entry)
<code>getElement()</code>	Returns the current entry level <code>DOMElement</code> object
<code>getXpath()</code>	Returns the <code>DOMXPath</code> object used internally to run queries on the <code>DOMDocument</code> object (this includes core and Extension namespaces pre-registered)
<code>getXpathPrefix()</code>	Returns the valid DOM path prefix prepended to all XPath queries matching the entry being queried
<code>getEncoding()</code>	Returns the encoding of the source XML document (note: this cannot account for errors such as the server sending documents in a different encoding). The default encoding applied in the absence of any other is the UTF-8 encoding of Unicode.
<code>getExtensions()</code>	Returns an array of all Extension objects loaded for the current entry (note: both feed-level and entry-level Extensions exist, and only entry-level Extensions are returned here). The array keys are in the form <code>{ExtensionName}Entry</code> .
<code>getExtension(string \$name)</code>	Returns an Extension object for the entry registered under the provided name. This allows more fine-grained access to Extensions which may otherwise be hidden within the implementation of the standard API methods.
<code>getType()</code>	Returns a static class constant (e.g. <code>ZendFeedReaderReader::TYPE_ATOM_03</code> , i.e. Atom 0.3) indicating exactly what kind of feed is being consumed.

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 125.5: 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 125.6: Non-Core Extensions (must register manually)

Syndication	Implements Syndication 1.0 support for RSS feeds
CreativeCom- mons	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.

```
1 Zend\Feed\Reader\Reader::registerExtension('Syndication');
2 $feed = Zend\Feed\Reader\Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
3 $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.

```
1 Zend\Feed\Reader\Reader::registerExtension('Syndication');
2 $feed = Zend\Feed\Reader\Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
3 $syndication = $feed->getExtension('Syndication');
4 $updatePeriod = $syndication->getUpdatePeriod();
```

Writing Zend\Feed\Reader\Reader 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*:

```
1 JungleBooks 1.0:
2 http://example.com/junglebooks/rss/module/1.0/
```

A snippet of *RSS* containing this extension in practice could be something similar to:

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <rss version="2.0"
3     xmlns:content="http://purl.org/rss/1.0/modules/content/"
4     xmlns:jungle="http://example.com/junglebooks/rss/module/1.0/">
5 <channel>
6     <title>Jungle Books Customer Reviews</title>
7     <link>http://example.com/junglebooks</link>
8     <description>Many book reviews!</description>
9     <pubDate>Fri, 26 Jun 2009 19:15:10 GMT</pubDate>
10    <jungle:dayPopular>
11        http://example.com/junglebooks/book/938
12    </jungle:dayPopular>
13    <item>
14        <title>Review Of Flatland: A Romance of Many Dimensions</title>
15        <link>http://example.com/junglebooks/review/987</link>
16        <author>Confused Physics Student</author>
17        <content:encoded>
18            A romantic square?!
19        </content:encoded>
20        <pubDate>Thu, 25 Jun 2009 20:03:28 -0700</pubDate>
21        <jungle:isbn>048627263X</jungle:isbn>
22    </item>
23 </channel>
24 </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).

```
1 class My\FedReader\Extension\JungleBooks\Entry
2     extends Zend\Feed\Reader\Extension\AbstractEntry
3 {
4     public function getIsbn()
5     {
6         if (isset($this->data['isbn'])) {
7             return $this->data['isbn'];
8         }
9         $isbn = $this->xpath->evaluate(
10             'string(' . $this->getXpathPrefix() . '/jungle:isbn)'
11         );
12         if (!$isbn) {
13             $isbn = null;
14         }
15         $this->data['isbn'] = $isbn;
16         return $this->data['isbn'];
17     }
18 }
```



```

19     protected function registerNamespaces()
20     {
21         $this->xpath->registerNamespace(
22             'jungle', 'http://example.com/junglebooks/rss/module/1.0/'
23         );
24     }
25 }

```

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.

```

1  if (!Zend\Feed\Reader\Reader::isRegistered('JungleBooks')) {
2      $extensions = Zend\Feed\Reader\Reader::getExtensionManager();
3      $extensions->setInvokableClass('JungleBooksEntry',
4      ↪ 'My\FeedReader\Extension\JungleBooks\Entry');
5      Zend\Feed\Reader\Reader::registerExtension('JungleBooks');
6  }
7  $feed = Zend\Feed\Reader\Reader::import('http://example.com/junglebooks/rss');
8
9  // ISBN for whatever book the first entry in the feed was concerned with
10 $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 feed level *API*.

```

1  class My\FeedReader\Extension\JungleBooks\Feed
2      extends Zend\Feed\Reader\Extension\AbstractFeed
3  {
4      public function getDaysPopularBookLink()
5      {
6          if (isset($this->data['dayPopular'])) {
7              return $this->data['dayPopular'];
8          }
9          $dayPopular = $this->xpath->evaluate(
10             'string(' . $this->getXpathPrefix() . '/jungle:dayPopular)'
11         );
12         if (!$dayPopular) {
13             $dayPopular = null;
14         }
15         $this->data['dayPopular'] = $dayPopular;
16         return $this->data['dayPopular'];
17     }
18
19     protected function registerNamespaces()
20     {
21         $this->xpath->registerNamespace(

```

```
22         'jungle', 'http://example.com/junglebooks/rss/module/1.0/'
23     );
24 }
25 }
```

Let's repeat the last example using a custom Extension to show the method being used.

```
1  if (!Zend\Fee\Reader\Reader::isRegistered('JungleBooks')) {
2      $extensions = Zend\Fee\Reader\Reader::getExtensionManager();
3      $extensions->setInvokableClass('JungleBooksFeed',
4      ↪ 'My\FeeReader\Extension\JungleBooks\Feed');
5      Zend\Fee\Reader\Reader::registerExtension('JungleBooks');
6  }
7  $feed = Zend\Fee\Reader\Reader::import('http://example.com/junglebooks/rss');
8
9  // URI to the information page of the day's most popular book with visitors
10 $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\Fee\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`.

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.

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 *URIs*, 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` (Tombstone) 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.

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.

```
1  /**
2   * Create the parent feed
3   */
4  $feed = new Zend\Feed\Writer\Feed;
5  $feed->setTitle('Paddy's Blog');
6  $feed->setLink('http://www.example.com');
7  $feed->setFeedLink('http://www.example.com/atom', 'atom');
8  $feed->addAuthor(array(
9      'name' => 'Paddy',
10     'email' => 'paddy@example.com',
11     'uri'  => 'http://www.example.com',
12 ));
13 $feed->setDateModified(time());
14 $feed->addHub('http://pubsubhubbub.appspot.com/');
15
16 /**
17  * Add one or more entries. Note that entries must
18  * be manually added once created.
19  */
20 $entry = $feed->createEntry();
21 $entry->setTitle('All Your Base Are Belong To Us');
22 $entry->setLink('http://www.example.com/all-your-base-are-belong-to-us');
23 $entry->addAuthor(array(
24     'name' => 'Paddy',
25     'email' => 'paddy@example.com',
26     'uri'  => 'http://www.example.com',
27 ));
```

```

28 $entry->setDateModified(time());
29 $entry->setDateCreated(time());
30 $entry->setDescription('Exposing the difficultly of porting games to English.');
```

31 \$entry->setContent(

```

32     'I am not writing the article. The example is long enough as is ;).'
33 );
34 $feed->addEntry($entry);
35
36 /**
37  * Render the resulting feed to Atom 1.0 and assign to $out.
38  * You can substitute "atom" with "rss" to generate an RSS 2.0 feed.
39  */
40 $out = $feed->export('atom');
```

The output rendered should be as follows:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <feed xmlns="http://www.w3.org/2005/Atom">
3      <title type="text">Paddy's Blog</title>
4      <subtitle type="text">Writing about PC Games since 176 BC.</subtitle>
5      <updated>2009-12-14T20:28:18+00:00</updated>
6      <generator uri="http://framework.zend.com" version="1.10.0alpha">
7          Zend\Feed\Writer
8      </generator>
9      <link rel="alternate" type="text/html" href="http://www.example.com"/>
10     <link rel="self" type="application/atom+xml"
11         href="http://www.example.com/atom"/>
12     <id>http://www.example.com</id>
13     <author>
14         <name>Paddy</name>
15         <email>paddy@example.com</email>
16         <uri>http://www.example.com</uri>
17     </author>
18     <link rel="hub" href="http://pubsubhubbub.appspot.com/">
19     <entry>
20         <title type="html"><![CDATA[All Your Base Are Belong To
21             Us]]></title>
22         <summary type="html">
23             <![CDATA[Exposing the difficultly of porting games to
24                 English.]]>
25         </summary>
26         <published>2009-12-14T20:28:18+00:00</published>
27         <updated>2009-12-14T20:28:18+00:00</updated>
28         <link rel="alternate" type="text/html"
29             href="http://www.example.com/all-your-base-are-belong-to-us"/>
30         <id>http://www.example.com/all-your-base-are-belong-to-us</id>
31         <author>
32             <name>Paddy</name>
33             <email>paddy@example.com</email>
34             <uri>http://www.example.com</uri>
35         </author>
36         <content type="html">
37             <![CDATA[I am not writing the article.
38                 The example is long enough as is ;).]]>
39         </content>
40     </entry>
41 </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.

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 126.1: 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 to the feed.
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

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.

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 126.2: 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.
setDescription()	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.
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. If omitted, the date used will be the current date and time.
setDateModified()	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.
setCopyright()	Sets a copyright notice associated with the feed.
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 encoded during rendering) and is a required key. The "scheme" (called the domain in RSS) is optional but must be a valid URI.
addCategory()	Sets the data for a single category following the same format as described above for a single sub-array.
setCommentCount()	Sets the number of comments associated with this entry. Rendering differs between RSS and Atom 2.0 depending on the element or attribute needed.
setCommentLink()	Set a link to a HTML page containing comments associated with this entry.
setCommentFeedLink()	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".
setCommentFeedLinks()	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.

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.

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.

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 *APIs* are covered in subsequent sections.

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 *URIs* 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 *URLs*, 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.

```
1 $publisher = new Zend\Feed\PubSubHubbub\Publisher;
2 $publisher->addHubUrls(array(
3     'http://pubsubhubbub.appspot.com/',
4     'http://hubbub.example.com',
```

```

5  ));
6  $publisher->addUpdatedTopicUrls(array(
7      'http://www.example.net/rss',
8      'http://www.example.net/atom',
9  ));
10 $publisher->notifyAll();
11
12 if (!$publisher->isSuccess()) {
13     // check for errors
14     $errors      = $publisher->getErrors();
15     $failedHubs = array();
16     foreach ($errors as $error) {
17         $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 *URIs*, 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.

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.

Subscribing and Unsubscribing

Zend\FeeD\PubSubHubbub\Subscriber implements a full PubsSubHubbub 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:

```
1 $storage = new Zend\FeeD\PubSubHubbub\Model\Subscription;
2
3 $subscriber = new Zend\FeeD\PubSubHubbub\Subscriber;
4 $subscriber->setStorage($storage);
5 $subscriber->addHubUrl('http://hubbub.example.com');
6 $subscriber->setTopicUrl('http://www.example.net/rss.xml');
7 $subscriber->setCallbackUrl('http://www.mydomain.com/hubbub/callback');
8 $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:

```
1 CREATE TABLE IF NOT EXISTS `subscription` (
2   `id` varchar(32) COLLATE utf8_unicode_ci NOT NULL DEFAULT '',
3   `topic_url` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
4   `hub_url` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
5   `created_time` datetime DEFAULT NULL,
6   `lease_seconds` bigint(20) DEFAULT NULL,
7   `verify_token` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
8   `secret` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
9   `expiration_time` datetime DEFAULT NULL,
10  `subscription_state` varchar(12) COLLATE utf8_unicode_ci DEFAULT NULL,
11  PRIMARY KEY (`id`)
12 ) 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 127.1: Subscription request parameters

Parameter	Value	Explanation
hub.callback	<code>http://www.mydomain.com/hubbub/callback?xhub.subscription=5536df06b5dcb966edab3a4c4d56213c16a8184</code>	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 <code>http://www.mydomain.com/hubbub/callback/5536df06b5dcb966edab3a4c4d56213c16a8184</code> . 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 <code>ZendPubSubHubbubSubscriberCallback::setSubscriptionKey()</code> . A detailed example is offered later.
hub.lease	<code>2500000</code>	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	<code>subscribe</code>	Simple value indicating this is a subscription request. Unsubscription requests would use the “unsubscribe” value.
hub.topic	<code>http://www.example.net/rss.xml</code>	The URI of the topic (i.e. Atom or RSS feed) which the Subscriber wishes to subscribe to for updates.
hub.verify	<code>sync</code>	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	<code>async</code>	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	<code>3064919804ab-caa7212ae89.879827871253878386</code>	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.

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_ASYNC)`. 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.

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.

```

1  $storage = new Zend\Feed\PubSubHubbub\Model\Subscription;
2  $callback = new Zend\Feed\PubSubHubbub\Subscriber\Callback;
3  $callback->setStorage($storage);
4  $callback->handle();
5  $callback->sendResponse();
6
7  /**
8   * Check if the callback resulting in the receipt of a feed update.
9   * Otherwise it was either a (un)sub verification request or invalid request.
10  * Typically we need do nothing other than add feed update handling - the rest
11  * is handled internally by the class.
12  */
13  if ($callback->hasFeedUpdate()) {
14      $feedString = $callback->getFeedUpdate();
15      /**

```

```
16     * Process the feed update asynchronously to avoid a Hub timeout.
17     */
18 }
```

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.

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.

```
1 use Zend\Mvc\Controller\AbstractActionController;
2
3 class CallbackController extends AbstractActionController
4 {
5
6     public function indexAction()
7     {
8         $storage = new Zend\Feed\PubSubHubbub\Model\Subscription;
9         $callback = new Zend\Feed\PubSubHubbub\Subscriber\Callback;
10        $callback->setStorage($storage);
11        /**
```

```

12      * Inject subscription key parsing from URL path using
13      * a parameter from Router.
14      */
15      $subscriptionKey = $this->params()->fromRoute('subkey');
16      $callback->setSubscriptionKey($subscriptionKey);
17      $callback->handle();
18      $callback->sendResponse();
19
20      /**
21       * Check if the callback resulting in the receipt of a feed update.
22       * Otherwise it was either a (un)sub verification request or invalid
23       * request. Typically we need do nothing other than add feed update
24       * handling - the rest is handled internally by the class.
25       */
26      if ($callback->hasFeedUpdate()) {
27          $feedString = $callback->getFeedUpdate();
28          /**
29           * Process the feed update asynchronously to avoid a Hub timeout.
30           */
31      }
32  }
33
34  }
    
```

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.

```

1  // Callback Route to enable appending a PuSH Subscription's lookup key
2  $route = Zend\Mvc\Router\Http\Segment::factory(array(
3      'route' => '/callback/:subkey',
4      'constraints' => array(
5          'subkey' => '[a-z0-9]+'
6      ),
7      'defaults' => array(
8          'controller' => 'application-index',
9          'action' => 'index'
10     )
11 ));
    
```


Overview

TODO

Available Methods

TODO

Examples

TODO

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.

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.

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:

```
1 $htmlEntities = new Zend\Filter\HtmlEntities();  
2
```

```
3 echo $htmlEntities->filter('&'); // &amp;
4 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:

```
1 $strtolower = new Zend\Filter\StringToLower;
2
3 echo $strtolower('I LOVE ZF2!'); // i love zf2!
4 $zf2love = $strtolower('I LOVE ZF2!');
```

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 its 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.

```
1 echo StaticFilter::execute('&', 'HtmlEntities');
```

You can also pass an array of constructor arguments, if they are needed for the filter class.

```
1 echo StaticFilter::execute('"',
2                          'HtmlEntities',
3                          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.

```
1 $pluginManager = StaticFilter::getPluginManager()->setInvokableClass(
2     'myNewFilter', 'MyCustom\Filter\MyNewFilter'
3 );
4
5 StaticFilter::setPluginManager(new MyFilterPluginManager());
```

This is useful when adding custom filters to be used by the `StaticFilter`.

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:

```
1 $original = "my_original_content";
2
3 // Attach a filter
4 $filter = new Zend\Filter\Word\UnderscoreToCamelCase();
```



```
5 $filtered = $filter->filter($original);
6
7 // Use it's opposite
8 $filter2 = new Zend\Filter\Word\CamelCaseToUnderscore();
9 $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.

Standard Filter Classes

Zend Framework comes with a standard set of filters, which are ready for you to use.

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

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()`

Basic Usage

```
1 // Default settings, deny whitespace
2 $filter = new \Zend\I18n\Filter\Alnum();
3 echo $filter->filter("This is (my) content: 123");
4 // Returns "Thisismycontent123"
5
```

```
6 // First param in constructor is $allowWhiteSpace
7 $filter = new \Zend\I18n\Filter\Alnum(true);
8 echo $filter->filter("This is (my) content: 123");
9 // 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.

Alpha

The Alpha filter can be used to return only alphabetic characters in the unicode “letter” category. All other characters are suppressed.

Supported Options

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()`

Basic Usage

```
1 // Default settings, deny whitespace
2 $filter = new \Zend\I18n\Filter\Alpha();
3 echo $filter->filter("This is (my) content: 123");
4 // Returns "Thisismycontent"
5
6 // Allow whitespace
7 $filter = new \Zend\I18n\Filter\Alpha(true);
8 echo $filter->filter("This is (my) content: 123");
9 // 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.

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:

```
1 $filter = new Zend\Filter\BaseName();
2
3 print $filter->filter('/vol/tmp/filename');
```

This will return 'filename'.

```
1 $filter = new Zend\Filter\BaseName();
2
3 print $filter->filter('/vol/tmp/filename.txt');
```

This will return 'filename.txt'.

Blacklist

This filter will return null if the value being filtered is present in the filter's list of values. If the value is not present, it will return that value.

For the opposite functionality see the Whitelist filter.

Supported Options

The following options are supported for Zend\Filter\Blacklist:

- **strict**: Uses strict mode when comparing: passed through to `in_array`'s third argument.
- **list**: An array of forbidden values.

Basic Usage

This is a basic example:

```
1 $blacklist = new \Zend\Filter\Blacklist(array(
2     'list' => array('forbidden-1', 'forbidden-2')
3 ));
4 echo $blacklist->filter('forbidden-1'); // => null
5 echo $blacklist->filter('allowed');    // => 'allowed'
```

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`.
- **translations**: This option sets the translations 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`.

```
1 $filter = new Zend\Filter\Boolean();
2 $value  = '';
3 $result = $filter->filter($value);
4 // 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:

```

1 // converts 0 to false
2 $filter = new Zend\Filter\Boolean(Zend\Filter\Boolean::TYPE_INTEGER);
3
4 // converts 0 and '0' to false
5 $filter = new Zend\Filter\Boolean(
6     Zend\Filter\Boolean::TYPE_INTEGER + Zend\Filter\Boolean::TYPE_ZERO_STRING
7 );
8
9 // converts 0 and '0' to false
10 $filter = new Zend\Filter\Boolean(array(
11     'type' => array(
12         Zend\Filter\Boolean::TYPE_INTEGER,
13         Zend\Filter\Boolean::TYPE_ZERO_STRING,
14     ),
15 ));
16
17 // converts 0 and '0' to false
18 $filter = new Zend\Filter\Boolean(array(
19     'type' => array(
20         'integer',
21         'zero',
22     ),
23 ));

```

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 translation and the corresponding value, you can use the `translations` option or the method `setTranslations`.

```

1 $filter = new Zend\Filter\Boolean(array(
2     'type'          => Zend\Filter\Boolean::TYPE_LOCALIZED,
3     'translations' => array(
4         'ja'      => true,
5         'nein'   => false,
6         'yes'    => true,
7         'no'     => false,
8     ),
9 ));
10
11 // returns false
12 $result = $filter->filter('nein');
13
14 // returns true
15 $result = $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 130.1: Usage without casting

Type	True	False
<code>Zend\Filter\Boolean::TYPE_BOOLEAN</code>	<code>TRUE</code>	<code>FALSE</code>
<code>Zend\Filter\Boolean::TYPE_EMPTY_ARRAY</code>	<code>Array()</code>	
<code>Zend\Filter\Boolean::TYPE_FALSE_STRING</code>	<code>"false"</code> (case independently)	<code>"true"</code> (case independently)
<code>Zend\Filter\Boolean::TYPE_FLOAT</code>	<code>0.0</code>	<code>1.0</code>
<code>Zend\Filter\Boolean::TYPE_INTEGER</code>	<code>0</code>	<code>1</code>
<code>Zend\Filter\Boolean::TYPE_LOCALIZED</code>	localized <code>"yes"</code> (case independently)	localized <code>"no"</code> (case independently)
<code>Zend\Filter\Boolean::TYPE_NULL</code>	<code>NULL</code>	
<code>Zend\Filter\Boolean::TYPE_STRING</code>	<code>""</code>	
<code>Zend\Filter\Boolean::TYPE_ZERO_STRING</code>	<code>"0"</code>	<code>"1"</code>

The following example shows the behaviour when changing the `casting` option:

```
1 $filter = new Zend\Filter\Boolean(array(  
2     'type'    => Zend\Filter\Boolean::TYPE_ALL,  
3     'casting' => false,  
4 ));  
5  
6 // returns false  
7 $result = $filter->filter(0);  
8  
9 // returns true  
10 $result = $filter->filter(1);  
11  
12 // returns the value  
13 $result = $filter->filter(2);
```

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.
- **callback_params**: 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.

```
1 $filter = new Zend\Filter\Callback('strrev');
2
3 print $filter->filter('Hello!');
4 // 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.

```
1 // Our classdefinition
2 class MyClass
3 {
4     public function Reverse($param);
5 }
6
7 // The filter definition
8 $filter = new Zend\Filter\Callback(array('MyClass', 'Reverse'));
9 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.

```
1 $filter = new Zend\Filter\Callback(
2     array(
3         'callback' => 'MyMethod',
4         'options'  => array('key' => 'param1', 'key2' => 'param2')
5     )
6 );
7 $filter->filter(array('value' => 'Hello'));
```

When you would call the above method definition manually it would look like this:

```
1 $value = MyMethod('Hello', 'param1', 'param2');
```

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.

```
1 $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).

```
1 $filter = new Zend\Filter\Compress(array(  
2     'adapter' => 'Bz2',  
3     'options' => array(  
4         'blocksize' => 8,  
5     ),  
6 ));
```

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.

```
1 $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.

```
1 $filter      = new Zend\Filter\Compress('Bz2');
2 $compressed = $filter->filter('Uncompressed string');
3 // Returns the compressed string
```

Decompression works the same way.

```
1 $filter      = new Zend\Filter\Decompress('Bz2');
2 $compressed = $filter->filter('Compressed string');
3 // 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.

```
1 $filter      = new Zend\Filter\Compress(array(
2     'adapter' => 'Bz2',
3     'options' => array(
4         'archive' => 'filename.bz2',
5     ),
6 ));
7 $compressed = $filter->filter('Uncompressed string');
8 // 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.

```
1 $filter      = new Zend\Filter\Compress(array(
2     'adapter' => 'Bz2',
3     'options' => array(
4         'archive' => 'filename.bz2'
5     ),
6 ));
7 $compressed = $filter->filter('C:\temp\compressme.txt');
8 // 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.

```
1 $filter      = new Zend\Filter\Compress(array(  
2     'adapter' => 'Bz2',  
3     'options' => array(  
4         'archive' => 'filename.bz2'  
5     ),  
6 ));  
7 $compressed = $filter->filter('C:\temp\somedir');  
8 // 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.

```
1 $filter      = new Zend\Filter\Decompress('Bz2');  
2 $decompressed = $filter->filter('filename.bz2');  
3 // 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.

```
1 $filter      = new Zend\Filter\Decompress(array(  
2     'adapter' => 'Zip',  
3     'options' => array(  
4         'target' => 'C:\temp',  
5     )  
6 ));  
7 $decompressed = $filter->filter('filename.zip');  
8 // Returns true on success and decompresses the archive file  
9 // 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.

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:

```
1 $filter = new Zend\Filter\Digits();
2
3 print $filter->filter('October 2012');
```

This returns "2012".

```
1 $filter = new Zend\Filter\Digits();
2
3 print $filter->filter('HTML 5 for Dummies');
```

This returns "5".

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:

```
1 $filter = new Zend\Filter\Dir();
2
3 print $filter->filter('/etc/passwd');
```

This returns “/etc”.

```
1 $filter = new Zend\Filter\Dir();
2
3 print $filter->filter('C:/Temp/x');
```

This returns “C:/Temp”.

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. You can either provide the path and filename 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 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.

```

1 // Use the BlockCipher adapter
2 $filter1 = new Zend\Filter\Encrypt(array('adapter' => 'BlockCipher'));
3
4 // Use the OpenSSL adapter
5 $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.

```

1 // Use the OpenSSL adapter
2 $filter = new Zend\Filter\Encrypt();
3 $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.

```

1 // Use the default AES encryption algorithm
2 $filter = new Zend\Filter\Encrypt(array('adapter' => 'BlockCipher'));
3 $filter->setKey('encryption key');
4
5 // or
6 $filter = new Zend\Filter\Encrypt(array(
7     'adapter' => 'BlockCipher',
8     'key'      => 'encryption key'
9 ));
10
11 $encrypted = $filter->filter('text to be encrypted');
12 printf("Encrypted text: %s\n", $encrypted);

```

You can get and set the encryption values also afterwards with the `getEncryption()` and `setEncryption()` methods.

```

1 // Use the default AES encryption algorithm
2 $filter = new Zend\Filter\Encrypt(array('adapter' => 'BlockCipher'));
3 $filter->setKey('encryption key');
4 var_dump($filter->getEncryption());
5

```

```

6 // Will print:
7 //array(4) {
8 //  ["key_iteration"]=>
9 //  int(5000)
10 //  ["algorithm"]=>
11 //  string(3) "aes"
12 //  ["hash"]=>
13 //  string(6) "sha256"
14 //  ["key"]=>
15 //  string(14) "encryption key"
16 //}

```

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).

```

1 $key = 'encryption key';
2 $text = 'message to encrypt';
3
4 // use the default adapter that is BlockCipher
5 $filter = new \Zend\Filter\Encrypt();
6 $filter->setKey('encryption key');
7 for ($i=0; $i < 10; $i++) {
8     printf("%d) %s\n", $i, $filter->filter($text));
9 }

```

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.

```

1 // use the default adapter that is BlockCipher
2 $filter = new \Zend\Filter\Encrypt();
3 $filter->setKey('encryption key');
4 $filter->setVector('12345678901234567890');
5 printf("%s\n", $filter->filter('message'));
6
7 // output:
8 //_
9 ↪04636a6cb8276fad0787a2e187803b6557f77825d5ca6ed4392be702b9754bb3MTIzNDU2Nzg5MDEyMzQ1NgZ+zPwTGpV6gQ

```

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.

```
1 $content =  
2     ↳ '04636a6cb8276fad0787a2e187803b6557f77825d5ca6ed4392be702b9754bb3MTIzNDU2Nzg5MDEyMzQ1NgZ+zPwTGpV6g';  
3 // use the default adapter that is BlockCipher  
4 $filter = new Zend\Filter\Decrypt();  
5 $filter->setKey('encryption key');  
6 printf("Decrypt: %s\n", $filter->filter($content));  
7  
8 // output:  
9 // 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 key also afterwards with the `getPublicKey()` and `setPublicKey()` methods. The private key can also be get and set with the related `getPrivateKey()` and `setPrivateKey()` methods.

```
1 // Use openssl and provide a private key  
2 $filter = new Zend\Filter\Encrypt(array(  
3     'adapter' => 'openssl',  
4     'private' => '/path/to/mykey/private.pem'  
5 ));  
6  
7 // of course you can also give the public keys at initiation  
8 $filter->setPublicKey('/public/key/path/public.pem');
```

Note: Note that the OpenSSL adapter will not work when you do not provide valid keys.

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.

```
1 // Use openssl and provide a private key  
2 $filter = new Zend\Filter\Encrypt(array(  
3     'adapter' => 'openssl',  
4     'passphrase' => 'enter here the passphrase for the private key',  
5     'private' => '/path/to/mykey/private.pem',  
6     'public' => '/public/key/path/public.pem'  
7 ));
```

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.

```
1 // Use openssl and provide a private key
2 $filter = new Zend\Filter\Encrypt(array(
3     'adapter' => 'openssl',
4     'passphrase' => 'enter here the passphrase for the private key',
5     'private' => '/path/to/mykey/private.pem',
6     'public' => '/public/key/path/public.pem'
7 ));
8
9 $encrypted = $filter->filter('text_to_be_encoded');
10 $envelope = $filter->getEnvelopeKey();
11 print $encrypted;
12
13 // 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.

```
1 // Use openssl and provide a private key
2 $filter = new Zend\Filter\Encrypt(array(
3     'adapter' => 'openssl',
4     'private' => '/path/to/mykey/private.pem',
5     'public' => '/public/key/path/public.pem',
6     'package' => true
7 ));
8
9 $encrypted = $filter->filter('text_to_be_encoded');
10 print $encrypted;
11
12 // 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.

```
1 // Use basic compression adapter
2 $filter1 = new Zend\Filter\Encrypt(array(
3     'adapter' => 'openssl',
4     'private' => '/path/to/mykey/private.pem',
5     'public' => '/public/key/path/public.pem',
6     'package' => true,
7     'compression' => 'bz2'
8 ));
9
```

```

10 // Use basic compression adapter
11 $filter2 = new Zend\Filter\Encrypt(array(
12     'adapter' => 'openssl',
13     'private'  => '/path/to/mykey/private.pem',
14     'public'   => '/public/key/path/public.pem',
15     'package'  => true,
16     'compression' => array('adapter' => 'zip', 'target' => '\usr\tmp\tmp.zip')
17 ));

```

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:

```

1 // Use openssl and provide a private key
2 $filter = new Zend\Filter\Decrypt(array(
3     'adapter' => 'openssl',
4     'private' => '/path/to/mykey/private.pem'
5 ));
6
7 // of course you can also give the envelope keys at initiation
8 $filter->setEnvelopeKey('/key/from/encoder/envelope_key.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 themselves passing the passphrase option.

```

1 // Use openssl and provide a private key
2 $filter = new Zend\Filter\Decrypt(array(
3     'adapter' => 'openssl',
4     'passphrase' => 'enter here the passphrase for the private key',
5     'private' => '/path/to/mykey/private.pem'
6 ));
7
8 // of course you can also give the envelope keys at initiation
9 $filter->setEnvelopeKey('/key/from/encoder/envelope_key.pem');

```

At last, decode the content. Our complete example for decrypting the previously encrypted content looks like this.

```

1 // Use openssl and provide a private key
2 $filter = new Zend\Filter\Decrypt(array(
3     'adapter' => 'openssl',
4     'passphrase' => 'enter here the passphrase for the private key',
5     'private' => '/path/to/mykey/private.pem'
6 ));
7
8 // of course you can also give the envelope keys at initiation

```

```
9 $filter->setEnvelopeKey('/key/from/encoder/envelope_key.pem');
10
11 $decrypted = $filter->filter('encoded_text_normally_unreadable');
12 print $decrypted;
```

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.

```
1 $filter = new Zend\Filter\HtmlEntities();
2
3 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:

```
1 $filter = new Zend\Filter\HtmlEntities(array('quotestyle' => ENT_QUOTES));
2
3 $input = "A 'single' and " . '"double"';
4 print $filter->filter($input);
```

The above example returns A `'single'` and `"double"`; . Notice that 'single' as well as "double" quotes are filtered.

```
1 $filter = new Zend\Filter\HtmlEntities(array('quotestyle' => ENT_COMPAT));
2
3 $input  = "A 'single' and " . '"double"';
4 print $filter->filter($input);
```

The above example returns A 'single' and "double"; . Notice that "double" quotes are filtered while 'single' quotes are not altered.

```
1 $filter = new Zend\Filter\HtmlEntities(array('quotestyle' => ENT_NOQUOTES));
2
3 $input  = "A 'single' and " . '"double"';
4 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`

```
1 $filter = new Zend\Filter\HtmlEntities();
2
3 $filter->setQuoteStyle(ENT_QUOTES);
4 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.

```
1 $filter = new Zend\Filter\HtmlEntities();
2
3 $filter->setQuoteStyle(ENT_QUOTES);
4 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`.

```
1 $filter = new Zend\Filter\HtmlEntities();
2
3 $filter->setQuoteStyle(ENT_QUOTES);
4 print $filter->getQuoteStyle(ENT_QUOTES);
```

ToInt

`Zend\Filter\ToInt` allows you to transform a scalar value which contains into an integer.

Supported Options

There are no additional options for `Zend\Filter\ToInt`.

Basic Usage

A basic example of usage is below:

```
1 $filter = new Zend\Filter\ToInt();
2
3 print $filter->filter('-4 is less than 0');
```

This will return `'-4'`.

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `int` is a reserved keyword, which required renaming the `Int` filter. If you were using the `Int` filter directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `ToInt` class instead.

Users pulling their `Int` filter instance from the filter plugin manager receive a `ToInt` instance instead starting in 2.4.0.

ToNull

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\ToNull`:

- **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.

```
1 $filter = new Zend\Filter\ToNull();
2 $value  = '';
3 $result = $filter->filter($value);
4 // returns null instead of the empty string
```

This means that without providing any configuration, `Zend\Filter\ToNull` 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()`. Therefore `Zend\Filter\ToNull` 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:

```

1 // converts false to null
2 $filter = new Zend\Filter\ToNull(Zend\Filter\ToNull::BOOLEAN);
3
4 // converts false and 0 to null
5 $filter = new Zend\Filter\ToNull(
6     Zend\Filter\ToNull::BOOLEAN + Zend\Filter\ToNull::INTEGER
7 );
8
9 // converts false and 0 to null
10 $filter = new Zend\Filter\ToNull( array(
11     Zend\Filter\ToNull::BOOLEAN,
12     Zend\Filter\ToNull::INTEGER
13 ));
14
15 // converts false and 0 to null
16 $filter = new Zend\Filter\ToNull(array(
17     'boolean',
18     'integer',
19 ));

```

You can also give a `Traversable` or an array to set the wished types. To set types afterwards use `setType()`.

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `null` is a reserved keyword, which required renaming the `Null` filter. If you were using the `Null` filter directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `ToNull` class instead.

Users pulling their `Null` filter instance from the filter plugin manager receive a `ToNull` instance instead starting in 2.4.0.

NumberFormat

The `NumberFormat` filter can be used to return locale-specific number and percentage strings. It extends the `NumberParse` filter, which acts as wrapper for the `NumberFormatter` class within the Internationalization extension (Intl).

Supported Options

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()`

Basic Usage

```
1 $filter = new \Zend\I18n\Filter\NumberFormat("de_DE");
2 echo $filter->filter(1234567.8912346);
3 // Returns "1.234.567,891"
4
5 $filter = new \Zend\I18n\Filter\NumberFormat("en_US", NumberFormatter::PERCENT);
6 echo $filter->filter(0.80);
7 // Returns "80%"
8
9 $filter = new \Zend\I18n\Filter\NumberFormat("fr_FR", NumberFormatter::SCIENTIFIC);
10 echo $filter->filter(0.00123456789);
11 // Returns "1,23456789E-3"
```

NumberParse

The `NumberParse` filter can be used to parse a number from a string. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl).

Supported Options

The following options are supported for `NumberParse`:

```
NumberParse([ string $locale [, int $style [, int $type ]]])
```

- `$locale`: (Optional) Locale in which the number would be parsed (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 parsing, one of the [format style constants](#). If unset, it will use `NumberFormatter::DEFAULT_STYLE` as the default style.

Methods for getting/setting the parse style are also available: `getStyle()` and `setStyle()`

- `$type`: (Optional) The [parsing type](#) to use. If unset, it will use `NumberFormatter::TYPE_DOUBLE` as the default type.

Methods for getting/setting the parse type are also available: `getType()` and `setType()`

Basic Usage

```

1 $filter = new \Zend\I18n\Filter\NumberParse("de_DE");
2 echo $filter->filter("1.234.567,891");
3 // Returns 1234567.8912346
4
5 $filter = new \Zend\I18n\Filter\NumberParse("en_US", NumberFormatter::PERCENT);
6 echo $filter->filter("80%");
7 // Returns 0.80
8
9 $filter = new \Zend\I18n\Filter\NumberParse("fr_FR", NumberFormatter::SCIENTIFIC);
10 echo $filter->filter("1,23456789E-3");
11 // Returns 0.00123456789

```

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.

```

1 $filter = new \Zend\Filter\PregReplace(array(
2     'pattern'      => '/bob/',
3     'replacement' => 'john',
4 ));

```

```
5 $input = 'Hi bob!';
6
7 $filter->filter($input);
8 // returns 'Hi john!'
```

You can use `getPattern()` and `setPattern()` to set the matching pattern afterwards. To set the replacement pattern you can use `getReplacement()` and `setReplacement()`.

```
1 $filter = new Zend\Filter\PregReplace();
2 $filter->setMatchPattern(array('bob', 'Hi'))
3     ->setReplacement(array('john', 'Bye'));
4 $input = 'Hi bob!';
5
6 $filter->filter($input);
7 // returns 'Bye john!'
```

For a more complex usage take a look into *PHP's PCRE Pattern Chapter*.

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`.

```
1 $filter = new Zend\Filter\RealPath();
2 $path   = '/www/var/path/../../mypath';
3 $filtered = $filter->filter($path);
4
5 // 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.

```
1 $filter = new Zend\Filter\RealPath(false);
2 $path   = '/www/var/path/../../non/existing/path';
3 $filtered = $filter->filter($path);
4
```

```
5 // returns '/www/non/existing/path'
6 // even when file_exists or realpath would return false
```

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:

```
1 $filter = new Zend\Filter\StringToLower();
2
3 print $filter->filter('SAMPLE');
4 // 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.

```
1 // using UTF-8
2 $filter = new Zend\Filter\StringToLower('UTF-8');
3
4 // or give an array which can be useful when using a configuration
5 $filter = new Zend\Filter\StringToLower(array('encoding' => 'UTF-8'));
6
7 // or do this afterwards
8 $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.

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:

```
1 $filter = new Zend\Filter\StringToUpper();
2
3 print $filter->filter('Sample');
4 // 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`.

```
1 $filter = new Zend\Filter\StringToUpper(array('encoding' => 'UTF-8'));
2
3 // or do this afterwards
4 $filter->setEncoding('ISO-8859-1');
```

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:

```
1 $filter = new Zend\Filter\StringTrim();
2
3 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

```

1 $filter = new Zend\Filter\StringTrim(':');
2 // or new Zend\Filter\StringTrim(array('charlist' => ':'));
3
4 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.

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:

```

1 $filter = new Zend\Filter\StripNewlines();
2
3 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.

StripTags

This filter can strip XML and HTML tags from given content.

Warning: `Zend\Filter\StripTags` is potentially insecure

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:

```
1 $filter = new Zend\Filter\StripTags();
2
3 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:

```
1 $filter = new Zend\Filter\StripTags();
2
3 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.

```
1 $filter = new Zend\Filter\StripTags(array('allowTags' => 'a'));
2
3 $input = "A text with <br/> a <a href='link.com'>link</a>";
4 print $filter->filter($input);
```

The above will return 'A text with a link' 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.

```
1 $filter = new Zend\Filter\StripTags(array('allowTags' => 'img', 'allowAttribs' => 'src
  ↳ '));
2
3 $input = "A text with <br/> a <img src='picture.com' width='100'>picture</img>";
4 print $filter->filter($input);
```

The above will return 'A text with a picture' 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.

Allowing Advanced Defined Tags with Attributes

You can pass the allowed tags with their attributes in a single array to the constructor.

```

1  $allowedElements = array(
2      'img' => array(
3          'src',
4          'width'
5      ),
6      'a' => array(
7          'href'
8      )
9  );
10 $filter = new Zend\Filter\StripTags($allowedElements);
11
12 $input  = "A text with <br/> a <img src='picture.com' width='100'>picture</img> click
13 ↪ " .
14         "<a href='http://picture.com/zend' id='hereId'>here</a>!";
15 print $filter->filter($input);

```

The above will return 'A text with a picture click here!' as result.

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.

Supported Options

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:

```

1  $filter = new Zend\Filter\UriNormalize(array(
2      'enforcedScheme' => 'https'
3  ));
4
5  echo $filter->filter('www.example.com');

```

As the result the string `https://www.example.com` will be output.

Whitelist

This filter will return `null` if the value being filtered is not present the filter's allowed list of values. If the value is present, it will return that value.

For the opposite functionality see the `Blacklist` filter.

Supported Options

The following options are supported for `Zend\Filter\Whitelist`:

- **strict**: Uses strict mode when comparing: passed through to `in_array`'s third argument.
- **list**: An array of allowed values.

Basic Usage

This is a basic example:

```
1 $whitelist = new \Zend\Filter\Whitelist(array(  
2     'list' => array('allowed-1', 'allowed-2')  
3 ));  
4 echo $whitelist->filter('allowed-2'); // => 'allowed-2'  
5 echo $whitelist->filter('not-allowed'); // => null
```

In addition to the standard set of filters, there are several classes specific to filtering word strings.

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:

```
1 $filter = new Zend\Filter\Word\CamelCaseToDash();  
2  
3 print $filter->filter('ThisIsMyContent');
```

The above example returns ‘This-Is-My-Content’.

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:

```
1 $filter = new Zend\Filter\Word\CamelCaseToSeparator(':');
2 // or new Zend\Filter\Word\CamelCaseToSeparator(array('separator' => ':'));
3
4 print $filter->filter('ThisIsMyContent');
```

The above example returns `'This:Is:My:Content'`.

Default Behavior

```
1 $filter = new Zend\Filter\Word\CamelCaseToSeparator();
2
3 print $filter->filter('ThisIsMyContent');
```

The above example returns `'This Is My Content'`.

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:

```
1 $filter = new Zend\Filter\Word\CamelCaseToUnderscore();
2
3 print $filter->filter('ThisIsMyContent');
```

The above example returns `'This_Is_My_Content'`.

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:

```
1 $filter = new Zend\Filter\Word\DashToCamelCase();
2
3 print $filter->filter('this-is-my-content');
```

The above example returns `'ThisIsMyContent'`.

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:

```
1 $filter = new Zend\Filter\Word\DashToSeparator('+');
2 // or new Zend\Filter\Word\CamelCaseToSeparator(array('separator' => '+'));
3
4 print $filter->filter('this-is-my-content');
```

The above example returns `'this+is+my+content'`.

Default Behavior

```
1 $filter = new Zend\Filter\Word\DashToSeparator();
2
3 print $filter->filter('this-is-my-content');
```

The above example returns `'this is my content'`.

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:

```
1 $filter = new Zend\Filter\Word\DashToUnderscore();
2
3 print $filter->filter('this-is-my-content');
```

The above example returns `'this_is_my_content'`.

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:

```
1 $filter = new Zend\Filter\Word\SeparatorToCamelCase(':');
2 // or new Zend\Filter\Word\SeparatorToCamelCase(array('separator' => ':'));
3
4 print $filter->filter('this:is:my:content');
```

The above example returns `'ThisIsMyContent'`.

Default Behavior

```
1 $filter = new Zend\Filter\Word\SeparatorToCamelCase();
2
3 print $filter->filter('this is my content');
```

The above example returns `'ThisIsMyContent'`.

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:

```
1 $filter = new Zend\Filter\Word\SeparatorToDash(':');
2 // or new Zend\Filter\Word\SeparatorToDash(array('separator' => ':'));
3
4 print $filter->filter('this:is:my:content');
```

The above example returns ‘this-is-my-content’.

Default Behavior

```
1 $filter = new Zend\Filter\Word\SeparatorToDash();
2
3 print $filter->filter('this is my content');
```

The above example returns ‘this-is-my-content’.

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:

```
1 $filter = new Zend\Filter\Word\SeparatorToSeparator(':', '+');
2
3 print $filter->filter('this:is:my:content');
```

The above example returns ‘this+is+my+content’.

Default Behaviour

```
1 $filter = new Zend\Filter\Word\SeparatorToSeparator();
2
3 print $filter->filter('this is my content');
```

The above example returns ‘this-is-my-content’.

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:

```
1 $filter = new Zend\Filter\Word\UnderscoreToCamelCase();
2
3 print $filter->filter('this_is_my_content');
```

The above example returns ‘ThisIsMyContent’.

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:

```
1 $filter = new Zend\Filter\Word\UnderscoreToSeparator('+');
2 // or new Zend\Filter\Word\CamelCaseToSeparator(array('separator' => '+'));
3
4 print $filter->filter('this_is_my_content');
```

The above example returns ‘this+is+my+content’.

Default Behavior

```
1 $filter = new Zend\Filter\Word\UnderscoreToSeparator();
2
3 print $filter->filter('this_is_my_content');
```

The above example returns ‘this is my content’.

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:

```
1 $filter = new Zend\Filter\Word\UnderscoreToDash();
2
3 print $filter->filter('this_is_my_content');
```

The above example returns ‘this-is-my-content’.

CHAPTER 132

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.

Decrypt

TODO

Encrypt

TODO

Lowercase

TODO

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:

```
1 // 'target' option is assumed if param is a string
2 $filter = new \Zend\Filter\File\Rename("/tmp/");
3 echo $filter->filter("./myfile.txt");
4 // File has been moved to "/tmp/myfile.txt"
```

Rename all filtered files to a new name:

```
1 $filter = new \Zend\Filter\File\Rename("/tmp/newfile.txt");
2 echo $filter->filter("./myfile.txt");
3 // File has been renamed to "/tmp/newfile.txt"
```

Move to a new path and randomize file names:

```
1 $filter = new \Zend\Filter\File\Rename(array(
2     "target"    => "/tmp/newfile.txt",
3     "randomize" => true,
4 ));
5 echo $filter->filter("./myfile.txt");
6 // File has been renamed to "/tmp/newfile_4b3403665fea6.txt"
```

Configure different options for several possible source files:

```
1 $filter = new \Zend\Filter\File\Rename(array(
2     array(
3         "source"    => "fileA.txt"
4         "target"    => "/dest1/newfileA.txt",
5         "overwrite" => true,
6     ),
7     array(
8         "source"    => "fileB.txt"
9         "target"    => "/dest2/newfileB.txt",
10        "randomize" => true,
11    ),
12 ));
```

```

12 ));
13 echo $filter->filter("fileA.txt");
14 // File has been renamed to "/dest1/newfileA.txt"
15 echo $filter->filter("fileB.txt");
16 // 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.

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.
- **use_upload_extension (boolean) default:** `false` When true, the uploaded file will maintains its original extension if not specified.
For example, if the uploaded file is `"file.txt"` and the target is something like `"mynewfile"`, the upload will be renamed to `"mynewfile.txt"`.

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:

```
1 use Zend\Http\PhpEnvironment\Request;
2
3 $request = new Request();
4 $files   = $request->getFiles();
5 // i.e. $files['my-upload']['tmp_name'] === '/tmp/php5Wx0aJ'
6 // i.e. $files['my-upload']['name'] === 'myfile.txt'
7
8 // 'target' option is assumed if param is a string
9 $filter = new \Zend\Filter\File\RenameUpload("./data/uploads/");
10 echo $filter->filter($files['my-upload']);
11 // File has been moved to "./data/uploads/php5Wx0aJ"
12
13 // ... or retain the uploaded file name
14 $filter->setUseUploadName(true);
15 echo $filter->filter($files['my-upload']);
16 // File has been moved to "./data/uploads/myfile.txt"
```

Rename all filtered files to a new name:

```
1 use Zend\Http\PhpEnvironment\Request;
2
3 $request = new Request();
4 $files   = $request->getFiles();
5 // i.e. $files['my-upload']['tmp_name'] === '/tmp/php5Wx0aJ'
6
7 $filter = new \Zend\Filter\File\Rename("./data/uploads/newfile.txt");
8 echo $filter->filter($files['my-upload']);
9 // File has been renamed to "./data/uploads/newfile.txt"
```

Move to a new path and randomize file names:

```
1 use Zend\Http\PhpEnvironment\Request;
2
3 $request = new Request();
4 $files   = $request->getFiles();
5 // i.e. $files['my-upload']['tmp_name'] === '/tmp/php5Wx0aJ'
6
7 $filter = new \Zend\Filter\File\Rename(array(
8     "target"    => "./data/uploads/newfile.txt",
9     "randomize" => true,
10 ));
11 echo $filter->filter($files['my-upload']);
12 // File has been renamed to "./data/uploads/newfile_4b3403665fea6.txt"
```

Uppercase

TODO

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:

```
1 // Create a filter chain and add filters to the chain
2 $filterChain = new Zend\Filter\FilterChain();
3 $filterChain->attach(new Zend\I18n\Filter\Alpha())
4                 ->attach(new Zend\Filter\StringToLower());
5
6 // Filter the username
7 $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.

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.

```
1 // Create a filter chain and add filters to the chain
2 $filterChain = new Zend\Filter\FilterChain();
3 $filterChain->attach(new Zend\I18n\Filter\Alpha())
4                 ->attach(new Zend\Filter\StringToLower(), 500);
```

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 known 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.

```
1 // Create a filter chain and add filters to the chain
2 $filterChain = new Zend\Filter\FilterChain();
3 $filterChain->attachByName('alpha')
4     ->attachByName('stringtolower', array('encoding' => 'utf-8'), 500);
```

The following example shows how to add a custom filter to the `FilterPluginManager` and the `FilterChain`.

```
1 $filterChain = new Zend\Filter\FilterChain();
2 $filterChain->getPluginManager()->setInvokableClass(
3     'myNewFilter', 'MyCustom\Filter\MyNewFilter'
4 );
5 $filterChain->attachByName('alpha')
6     ->attachByName('myNewFilter');
```

You can also add your own `FilterPluginManager` implementation.

```
1 $filterChain = new Zend\Filter\FilterChain();
2 $filterChain->setPluginManager(new MyFilterPluginManager());
3 $filterChain->attach(new Zend\I18n\Filter\Alpha())
4     ->attach(new MyCustom\Filter\MyNewFilter());
```

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

```
1 $inflector = new Zend\Filter\Inflector('pages/:page.:suffix');
2 $inflector->setRules(array(
3     ':page' => array('Word\CamelCaseToDash', 'StringToLower'),
4     'suffix' => 'html',
5 ));
6
7 $string = 'camelCasedWords';
8 $filtered = $inflector->filter(array('page' => $string));
9 // pages/camel-cased-words.html
10
11 $string = 'this_is_not_camel_cased';
12 $filtered = $inflector->filter(array('page' => $string));
13 // pages/this_is_not_camel_cased.html
```

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\I18n\Filter\Alpha'` or `'Zend\Filter\StringToLower'`, you'd specify only `'Alpha'` or `'StringToLower'`.

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:

```
1 // use Zend\Filter\Word\CamelCaseToDash as a rule
2 $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.

```
1 $filters = $inflector->getPluginManager();
2 $filters->addInvokableClass('mungify', 'My\Custom\Filter\Mungify');
```

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 fourth argument to the constructor:

```
1 // Via constructor:
2 $inflector = new Zend\Filter\Inflector('#foo/#bar.#sfx', array(), null, '#');
3
4 // Via accessor:
5 $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:

```
1 $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:

```
1 class Foo
2 {
3     /**
4      * @var string Inflector target
5      */
6     protected $_target = 'foo/:bar/:baz.:suffix';
```

```

7
8  /**
9   * Constructor
10  * @return void
11  */
12  public function __construct()
13  {
14      $this->_inflector = new Zend\Filter\Inflector();
15      $this->_inflector->setTargetReference($this->_target);
16  }
17
18  /**
19   * Set target; updates target in inflector
20   *
21   * @param string $target
22   * @return Foo
23   */
24  public function setTarget($target)
25  {
26      $this->_target = $target;
27      return $this;
28  }
29  }

```

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.

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:

```

1  $inflector = new Zend\Filter\Inflector(':script.:suffix');
2  $inflector->setStaticRule('suffix', 'phtml');
3
4  // change it later:
5  $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:

```

1 class Foo
2 {
3     /**
4      * @var string Suffix
5      */
6     protected $_suffix = 'phtml';
7
8     /**
9      * Constructor
10     * @return void
11     */
12     public function __construct()
13     {
14         $this->_inflector = new Zend\Filter\Inflector(':script.:suffix');
15         $this->_inflector->setStaticRuleReference('suffix', $this->_suffix);
16     }
17
18     /**
19      * Set suffix; updates suffix static rule in inflector
20      *
21      * @param string $suffix
22      * @return Foo
23      */
24     public function setSuffix($suffix)
25     {
26         $this->_suffix = $suffix;
27         return $this;
28     }
29 }

```

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.

```

1 $inflector = new Zend\Filter\Inflector(':script.:suffix');
2
3 // Set rule to use Zend\Filter\Word\CamelCaseToDash filter
4 $inflector->setFilterRule('script', 'Word\CamelCaseToDash');
5
6 // Add rule to lowercase string
7 $inflector->addFilterRule('script', new Zend\Filter\StringToLower());
8

```

```

9 // Set rules en-masse
10 $inflector->setFilterRule('script', array(
11     'Word\CamelCaseToDash',
12     new Zend\Filter\StringToLower()
13 ));

```

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

```

1 // Could also use setRules() with this notation:
2 $inflector->addRules(array(
3     // filter rules:
4     ':controller' => array('Word\CamelCaseToUnderscore', 'StringToLower'),
5     ':action'     => array('Word\CamelCaseToUnderscore', 'StringToLower'),
6
7     // Static rule:
8     'suffix'      => 'phtml'
9 ));

```

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.

Using a Traversable or an array

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()`.

Example

```
1 // With the constructor:
2 $options; // implements Traversable
3 $inflector = new Zend\Filter\Inflector($options);
4
5 // Or with setOptions():
6 $inflector = new Zend\Filter\Inflector();
7 $inflector->setOptions($options);
```

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 its 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.

```
1 echo StaticFilter::execute('&', 'HtmlEntities');
```

You can also pass an array of constructor arguments, if they are needed for the filter class.

```
1 echo StaticFilter::execute('"  
2     'HtmlEntities',  
3     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.

```
1 $pluginManager = StaticFilter::getPluginManager()->setInvokableClass(  
2     'myNewFilter', 'MyCustom\Filter\MyNewFilter'  
3 );  
4  
5 StaticFilter::setPluginManager(new MyFilterPluginManager());
```

This is useful when adding custom filters to be used by the `StaticFilter`.

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.

Example

The following example demonstrates how to write a custom filter:

```
1 namespace Application\Filter;
2
3 use Zend\Filter\FilterInterface;
4
5 class MyFilter implements FilterInterface
6 {
7     public function filter($value)
8     {
9         // perform some transformation upon $value to arrive on $valueFiltered
10
11         return $valueFiltered;
12     }
13 }
```

To attach an instance of the filter defined above to a filter chain:

```
1 $filterChain = new Zend\Filter\FilterChain();
2 $filterChain->attach(new Application\Filter\MyFilter());
```


`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 *ZendStdlib\Hydrator*.

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.

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.

Programmatic Form Creation

If nothing else, you can simply start creating elements, fieldsets, and forms and wiring them together.

```
1 use Zend\Captcha;
2 use Zend\Form\Element;
3 use Zend\Form\Fieldset;
4 use Zend\Form\Form;
5 use Zend\InputFilter\Input;
6 use Zend\InputFilter\InputFilter;
7
8 $name = new Element('name');
9 $name->setLabel('Your name');
10 $name->setAttributes(array(
11     'type' => 'text'
12 ));
13
14 $email = new Element\Email('email');
15 $email->setLabel('Your email address');
16
17 $subject = new Element('subject');
18 $subject->setLabel('Subject');
19 $subject->setAttributes(array(
20     'type' => 'text'
```

```
21 ));
22
23 $message = new Element\Textarea('message');
24 $message->setLabel('Message');
25
26 $captcha = new Element\Captcha('captcha');
27 $captcha->setCaptcha(new Captcha\Dumb());
28 $captcha->setLabel('Please verify you are human');
29
30 $csrf = new Element\Csrf('security');
31
32 $send = new Element('send');
33 $send->setValue('Submit');
34 $send->setAttributes(array(
35     'type' => 'submit'
36 ));
37
38
39 $form = new Form('contact');
40 $form->add($name);
41 $form->add($email);
42 $form->add($subject);
43 $form->add($message);
44 $form->add($captcha);
45 $form->add($csrf);
46 $form->add($send);
47
48 $nameInput = new Input('name');
49 // configure input... and all others
50 $inputFilter = new InputFilter();
51 // attach all inputs
52
53 $form->setInputFilter($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.

```
1 $sender = new Fieldset('sender');
2 $sender->add($name);
3 $sender->add($email);
4
5 $details = new Fieldset('details');
6 $details->add($subject);
7 $details->add($message);
8
9 $form = new Form('contact');
10 $form->add($sender);
11 $form->add($details);
12 $form->add($captcha);
13 $form->add($csrf);
14 $form->add($send);
```

Regardless of approach, as you can see, this can be tedious.

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.

```

1  use Zend\Form\Factory;
2
3  $factory = new Factory();
4  $form    = $factory->createForm(array(
5      'hydrator' => 'Zend\Stdlib\Hydrator\ArraySerializable',
6      'elements' => array(
7          array(
8              'spec' => array(
9                  'name' => 'name',
10                 'options' => array(
11                     'label' => 'Your name',
12                 ),
13                 'type' => 'Text',
14             ),
15         ),
16         array(
17             'spec' => array(
18                 'type' => 'Zend\Form\Element\Email',
19                 'name' => 'email',
20                 'options' => array(
21                     'label' => 'Your email address',
22                 ),
23             ),
24         ),
25         array(
26             'spec' => array(
27                 'name' => 'subject',
28                 'options' => array(
29                     'label' => 'Subject',
30                 ),
31                 'type' => 'Text',
32             ),
33         ),
34         array(
35             'spec' => array(
36                 'type' => 'Zend\Form\Element\Textarea',
37                 'name' => 'message',
38                 'options' => array(
39                     'label' => 'Message',
40                 ),
41             ),
42         ),
43         array(
44             'spec' => array(
45                 'type' => 'Zend\Form\Element\Captcha',
46                 'name' => 'captcha',
47                 'options' => array(
48                     'label' => 'Please verify you are human.',
49                     'captcha' => array(
50                         'class' => 'Dumb',
51                     ),
52                 ),
53         ),

```

```

54     ),
55     array(
56         'spec' => array(
57             'type' => 'Zend\Form\Element\Csrf',
58             'name' => 'security',
59         ),
60     ),
61     array(
62         'spec' => array(
63             'name' => 'send',
64             'type' => 'Submit',
65             'attributes' => array(
66                 'value' => 'Submit',
67             ),
68         ),
69     ),
70 ),
71 /* If we had fieldsets, they'd go here; fieldsets contain
72  * "elements" and "fieldsets" keys, and potentially a "type"
73  * key indicating the specific FieldsetInterface
74  * implementation to use.
75  * 'fieldsets' => array(
76  * ),
77  */
78
79 // Configuration to pass on to
80 // Zend\InputFilter\Factory::createInputFilter()
81 'input_filter' => array(
82     /* ... */
83 ),
84 );

```

If we wanted to use fieldsets, as we demonstrated in the previous example, we could do the following:

```

1  use Zend\Form\Factory;
2
3  $factory = new Factory();
4  $form    = $factory->createForm(array(
5      'hydrator' => 'Zend\Stdlib\Hydrator\ArraySerializable',
6      'fieldsets' => array(
7          array(
8              'spec' => array(
9                  'name' => 'sender',
10                 'elements' => array(
11                     array(
12                         'spec' => array(
13                             'name' => 'name',
14                             'options' => array(
15                                 'label' => 'Your name',
16                             ),
17                             'type' => 'Text'
18                         ),
19                     ),
20                 ),
21                 array(
22                     'spec' => array(
23                         'type' => 'Zend\Form\Element\Email',
24                         'name' => 'email',
25                         'options' => array(

```

```

25         'label' => 'Your email address',
26     ),
27 ),
28 ),
29 ),
30 ),
31 ),
32 array(
33     'spec' => array(
34         'name' => 'details',
35         'elements' => array(
36             array(
37                 'spec' => array(
38                     'name' => 'subject',
39                     'options' => array(
40                         'label' => 'Subject',
41                     ),
42                     'type' => 'Text',
43                 ),
44             ),
45             array(
46                 'spec' => array(
47                     'name' => 'message',
48                     'type' => 'Zend\Form\Element\Textarea',
49                     'options' => array(
50                         'label' => 'Message',
51                     ),
52                 ),
53             ),
54         ),
55     ),
56 ),
57 ),
58 'elements' => array(
59     array(
60         'spec' => array(
61             'type' => 'Zend\Form\Element\Captcha',
62             'name' => 'captcha',
63             'options' => array(
64                 'label' => 'Please verify you are human. ',
65                 'captcha' => array(
66                     'class' => 'Dumb',
67                 ),
68             ),
69         ),
70     ),
71     array(
72         'spec' => array(
73             'type' => 'Zend\Form\Element\Csrf',
74             'name' => 'security',
75         ),
76     ),
77 array(
78     'spec' => array(
79         'name' => 'send',
80         'type' => 'Submit',
81         'attributes' => array(
82             'value' => 'Submit',

```

```
83         ),
84     ),
85 ),
86 ),
87 // Configuration to pass on to
88 // Zend\InputFilter\Factory::createInputFilter()
89 'input_filter' => array(
90     /* ... */
91 ),
92 );
```

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.

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.

```
1 namespace Contact;
2
3 use Zend\Captcha\AdapterInterface as CaptchaAdapter;
4 use Zend\Form\Element;
5 use Zend\Form\Form;
6
7 class ContactForm extends Form
8 {
9     protected $captcha;
10
11     public function __construct(CaptchaAdapter $captcha)
12     {
13
14         parent::__construct();
15
16         $this->captcha = $captcha;
17
18         // add() can take either an Element/Fieldset instance,
19         // or a specification, from which the appropriate object
20         // will be built.
21
22         $this->add(array(
23             'name' => 'name',
24             'options' => array(
25                 'label' => 'Your name',
26             ),
27             'type' => 'Text',
28         ));
29         $this->add(array(
30             'type' => 'Zend\Form\Element\Email',
31             'name' => 'email',
32             'options' => array(
33                 'label' => 'Your email address',
34             ),
```

```

35         ));
36         $this->add(array(
37             'name' => 'subject',
38             'options' => array(
39                 'label' => 'Subject',
40             ),
41             'type' => 'Text',
42         ));
43         $this->add(array(
44             'type' => 'Zend\Form\Element\Textarea',
45             'name' => 'message',
46             'options' => array(
47                 'label' => 'Message',
48             ),
49         ));
50         $this->add(array(
51             'type' => 'Zend\Form\Element\Captcha',
52             'name' => 'captcha',
53             'options' => array(
54                 'label' => 'Please verify you are human.',
55                 'captcha' => $this->captcha,
56             ),
57         ));
58         $this->add(new Element\Csrf('security'));
59         $this->add(array(
60             'name' => 'send',
61             'type' => 'Submit',
62             'attributes' => array(
63                 'value' => 'Submit',
64             ),
65         ));
66
67         // We could also define the input filter here, or
68         // lazy-create it in the getInputFilter() method.
69     }
70 }

```

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.

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.

```

1  // assuming $captcha is an instance of some Zend\Captcha\AdapterInterface
2  $form = new Contact\ContactForm($captcha);
3
4  // If the form doesn't define an input filter by default, inject one.
5  $form->setInputFilter(new Contact\ContactFilter());
6
7  // Get the data. In an MVC application, you might try:
8  $data = $request->getPost(); // for POST data

```

```
9  $data = $request->getQuery(); // for GET (or query string) data
10
11  $form->setData($data);
12
13  // Validate the form
14  if ($form->isValid()) {
15      $validatedData = $form->getData();
16  } else {
17      $messages = $form->getMessages();
18  }
```

You can get the raw data if you want, by accessing the composed input filter.

```
1  $filter = $form->getInputFilter();
2
3  $rawValues = $filter->getRawValues();
4  $nameRawValue = $filter->getRawValue('name');
```

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:

```
1  namespace Zend\Form\Element;
2
3  use Zend\Form\Element;
4  use Zend\InputFilter\InputProviderInterface;
5  use Zend\Validator\Regex as RegexValidator;
6  use Zend\Validator\ValidatorInterface;
7
8  class Color extends Element implements InputProviderInterface
9  {
10     /**
11      * Seed attributes
12      *
13      * @var array
14      */
15     protected $attributes = array(
16         'type' => 'color',
17     );
18
19     /**
20      * @var ValidatorInterface
```

```

21     */
22     protected $validator;
23
24     /**
25      * Get validator
26      *
27      * @return ValidatorInterface
28      */
29     protected function getValidator()
30     {
31         if (null === $this->validator) {
32             $this->validator = new RegexValidator('/^#[0-9a-fA-F]{6}$/');
33         }
34         return $this->validator;
35     }
36
37     /**
38      * Provide default input rules for this element
39      *
40      * Attaches an email validator.
41      *
42      * @return array
43      */
44     public function getInputSpecification()
45     {
46         return array(
47             'name' => $this->getName(),
48             'required' => true,
49             'filters' => array(
50                 array('name' => 'Zend\Filter\StringTrim'),
51                 array('name' => 'Zend\Filter\StringToLower'),
52             ),
53             'validators' => array(
54                 $this->getValidator(),
55             ),
56         );
57     }
58 }

```

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.

```

1 namespace Contact\Form;
2
3 use Zend\Form\Fieldset;
4 use Zend\InputFilter\InputFilterProviderInterface;
5 use Zend\Validator;
6
7 class SenderFieldset extends Fieldset implements InputFilterProviderInterface
8 {
9     public function getInputFilterSpecification()
10     {
11         return array(
12             'name' => array(

```

```
13         'required' => true,
14         'filters' => array(
15             array('name' => 'Zend\Filter\StringTrim'),
16         ),
17         'validators' => array(
18             array(
19                 'name' => 'Zend\Validator\StringLength',
20                 'options' => array(
21                     'min' => 3,
22                     'max' => 256
23                 ),
24             ),
25         ),
26     ),
27     'email' => array(
28         'required' => true,
29         'filters' => array(
30             array('name' => 'Zend\Filter\StringTrim'),
31         ),
32         'validators' => array(
33             new Validator\EmailAddress(),
34         ),
35     ),
36 );
37 }
38 }
```

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!

Note: If you set custom input filter specification either in `getInputSpecification()` or in `getInputFilterSpecification()`, the `Zend\InputFilter\InputInterface` set for that specific field is reset to the default `Zend\InputFilter\Input`.

Some form elements may need a particular input filter, like `Zend\Form\Element\File`: in this case it's mandatory to specify the `type` key in your custom specification to match the original one (in ex. for the file element it's `Zend\InputFilter\FileInput`).

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.

```

1 $contact = new ArrayObject;
2 $contact['subject'] = '[Contact Form] ';
3 $contact['message'] = 'Type your message here';
4
5 $form     = new Contact\ContactForm;
6
7 $form->bind($contact); // form now has default values for
8                       // 'subject' and 'message'
9
10 $data = array(
11     'name'     => 'John Doe',
12     'email'    => 'j.doe@example.tld',
13     'subject' => '[Contact Form] \'sup?',
14 );
15 $form->setData($data);
16
17 if ($form->isValid()) {
18     // $contact now looks like:
19     // array(
20     //     'name'     => 'John Doe',
21     //     'email'    => 'j.doe@example.tld',
22     //     'subject' => '[Contact Form] \'sup?',
23     //     'message' => 'Type your message here',
24     // )
25     // only as an ArrayObject
26 }
```

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.

```

1 use Zend\Form\FormInterface;
2 $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:

```

1 namespace Zend\Stdlib\Hydrator;
2
3 interface HydratorInterface
4 {
5     /** @return array */
6     public function extract($object);
7     public function hydrate(array $data, $object);
8 }
```

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.

```
1 <?php
2 // within a view script
3 $form = $this->form;
4 $form->prepare();
5
6 // Assuming the "contact/process" route exists...
7 $form->setAttribute('action', $this->url('contact/process'));
8
9 // Set the method attribute for the form
10 $form->setAttribute('method', 'post');
11
12 // Get the form label plugin
13 $formLabel = $this->plugin('formLabel');
14
15 // Render the opening tag
16 echo $this->form()->openTag($form);
17 ?>
18 <div class="form_element">
19 <?php
20     $name = $form->get('name');
21     echo $formLabel->openTag() . $name->getOption('label');
22     echo $this->formInput($name);
23     echo $this->formElementErrors($name);
24     echo $formLabel->closeTag();
25 ?></div>
26
27 <div class="form_element">
28 <?php
29     $subject = $form->get('subject');
30     echo $formLabel->openTag() . $subject->getOption('label');
31     echo $this->formInput($subject);
32     echo $this->formElementErrors($subject);
33     echo $formLabel->closeTag();
34 ?></div>
35
36 <div class="form_element">
37 <?php
38     $message = $form->get('message');
39     echo $formLabel->openTag() . $message->getOption('label');
40     echo $this->formTextarea($message);
41     echo $this->formElementErrors($message);
42     echo $formLabel->closeTag();
43 ?></div>
44
45 <div class="form_element">
46 <?php
47     $captcha = $form->get('captcha');
48     echo $formLabel->openTag() . $captcha->getOption('label');
49     echo $this->formCaptcha($captcha);
50     echo $this->formElementErrors($captcha);
```

```

51     echo $formLabel->closeTag();
52 ?></div>
53
54 <?php echo $this->formElement($form->get('security')) ?>
55 <?php echo $this->formElement($form->get('send')) ?>
56
57 <?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 :

```

1  <?php
2  // within a view script
3  $form = $this->form;
4  $form->prepare();
5
6  // Assuming the "contact/process" route exists...
7  $form->setAttribute('action', $this->url('contact/process'));
8
9  // Set the method attribute for the form
10 $form->setAttribute('method', 'post');
11
12 // Render the opening tag
13 echo $this->form()->openTag($form);
14 ?>
15 <div class="form_element">
16 <?php
17     $name = $form->get('name');
18     echo $this->formRow($name);
19 ?></div>
20
21 <div class="form_element">
22 <?php
23     $subject = $form->get('subject');
24     echo $this->formRow($subject);
25 ?></div>
26
27 <div class="form_element">
28 <?php
29     $message = $form->get('message');
30     echo $this->formRow($message);
31 ?></div>
32
33 <div class="form_element">
34 <?php
35     $captcha = $form->get('captcha');
    
```

```
36     echo $this->formRow($captcha);
37 ?></div>
38
39 <?php echo $this->formElement($form->get('security')) ?>
40 <?php echo $this->formElement($form->get('send')) ?>
41
42 <?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 :

```
1 <div class="form_element">
2 <?php
3     $name = $form->get('name');
4     echo $this->formRow($name, 'append');
5 ?></div>
```

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.

```
1 $form->add(array(
2     'name' => 'phoneNumber',
3     'options' => array(
4         'label' => 'Your phone number'
5     ),
6     'attributes' => array(
7         'type' => 'tel'
8         'required' => 'required',
9         'pattern' => '^0[1-68]([- .]?[0-9]{2}){4}$'
10    )
11 ));
```

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.

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.

```
1 $form->setValidationGroup('name', 'email', 'subject', 'message');
2 $form->setData($data);
```

```

3  if ($form->isValid()) {
4      // Contains only the "name", "email", "subject", and "message" values
5      $data = $form->getData();
6  }

```

If you later want to reset the form to validate all, simply pass the `FormInterface::VALIDATE_ALL` flag to the `setValidationGroup()` method.

```

1  use Zend\Form\FormInterface;
2  $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 :

```

1  $form->setValidationGroup(array(
2      'profile' => array(
3          'firstname',
4          'lastname'
5      )
6  ));
7  $form->setData($data);
8  if ($form->isValid()) {
9      // Contains only the "firstname" and "lastname" values from the
10     // "profile" fieldset
11     $data = $form->getData();
12 }

```

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: `@ComposedObject("Namespace\Model\ComposedObject")` or an array to compose a collection: `@ComposedObject({"target_object":"Namespace\Model\ComposedCollection", "is_collection":"true", "options":{"count":2}})` `target_object` is the element to compose, `is_collection` flags this as a collection and `options` can take an array of options to pass into the collection.
- **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.
- *Instance*: specify an object class instance to bind to the form or fieldset.
- *Name*: specify the name of the current element, fieldset, or form. A string value is expected.
- *Object*: specify an object class instance to bind to the form or fieldset. (Note: this is deprecated in 2.4.0; use *Instance* instead.)
- *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 parsing 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.

```
1 use Zend\Form\Annotation;
2
3 /**
4  * @Annotation\Name("user")
5  * @Annotation\Hydrator("Zend\Stdlib\Hydrator\ObjectProperty")
6  */
7 class User
8 {
9     /**
10      * @Annotation\Exclude()
11      */
12     public $id;
```

```

13
14  /**
15   * @Annotation\Filter({"name":"StringTrim"})
16   * @Annotation\Validator({"name":"StringLength", "options":{"min":1, "max":25}})
17   * @Annotation\Validator({"name":"Regex", "options":{"pattern":"/^[a-zA-Z][a-zA-Z-
↪ 0-9_-]{0,24}$/"})
18   * @Annotation\Attributes({"type":"text"})
19   * @Annotation\Options({"label":"Username:"})
20   */
21   public $username;
22
23  /**
24   * @Annotation\Type("Zend\Form\Element\Email")
25   * @Annotation\Options({"label":"Your email address:"})
26   */
27   public $email;
28 }

```

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`:

```

1  use Zend\Form\Annotation\AnnotationBuilder;
2
3  $builder = new AnnotationBuilder();
4  $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.

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.

```
1  namespace Application\Entity;
2
3  class Product
4  {
5      /**
6       * @var string
7       */
8      protected $name;
9
10     /**
11      * @var int
12      */
13     protected $price;
14
15     /**
16      * @var Brand
17      */
18     protected $brand;
19
20     /**
21      * @var array
22      */
23     protected $categories;
24
25     /**
26      * @param string $name
27      * @return Product
28      */
```

```
29 public function setName($name)
30 {
31     $this->name = $name;
32     return $this;
33 }
34
35 /**
36  * @return string
37  */
38 public function getName()
39 {
40     return $this->name;
41 }
42
43 /**
44  * @param int $price
45  * @return Product
46  */
47 public function setPrice($price)
48 {
49     $this->price = $price;
50     return $this;
51 }
52
53 /**
54  * @return int
55  */
56 public function getPrice()
57 {
58     return $this->price;
59 }
60
61 /**
62  * @param Brand $brand
63  * @return Product
64  */
65 public function setBrand(Brand $brand)
66 {
67     $this->brand = $brand;
68     return $this;
69 }
70
71 /**
72  * @return Brand
73  */
74 public function getBrand()
75 {
76     return $this->brand;
77 }
78
79 /**
80  * @param array $categories
81  * @return Product
82  */
83 public function setCategories(array $categories)
84 {
85     $this->categories = $categories;
86     return $this;
```

```
87     }
88
89     /**
90      * @return array
91      */
92     public function getCategories()
93     {
94         return $this->categories;
95     }
96 }
97
98 class Brand
99 {
100     /**
101      * @var string
102      */
103     protected $name;
104
105     /**
106      * @var string
107      */
108     protected $url;
109
110     /**
111      * @param string $name
112      * @return Brand
113      */
114     public function setName($name)
115     {
116         $this->name = $name;
117         return $this;
118     }
119
120     /**
121      * @return string
122      */
123     public function getName()
124     {
125         return $this->name;
126     }
127
128     /**
129      * @param string $url
130      * @return Brand
131      */
132     public function setUrl($url)
133     {
134         $this->url = $url;
135         return $this;
136     }
137
138     /**
139      * @return string
140      */
141     public function getUrl()
142     {
143         return $this->url;
144     }
145 }
```

```
145 }
146
147 class Category
148 {
149     /**
150      * @var string
151      */
152     protected $name;
153
154     /**
155      * @param string $name
156      * @return Category
157      */
158     public function setName($name)
159     {
160         $this->name = $name;
161         return $this;
162     }
163
164     /**
165      * @return string
166      */
167     public function getName()
168     {
169         return $this->name;
170     }
171 }
```

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).

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:

```
1 namespace Application\Form;
2
3 use Application\Entity\Brand;
4 use Zend\Form\Fieldset;
5 use Zend\InputFilter\InputFilterProviderInterface;
6 use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;
7
8 class BrandFieldset extends Fieldset implements InputFilterProviderInterface
9 {
10     public function __construct()
11     {
12         parent::__construct('brand');
13
14         $this
15             ->setHydrator(new ClassMethodsHydrator(false))
16             ->setObject(new Brand());
17
18         ;
19
20         $this->add(array(
```

```

20         'name' => 'name',
21         'options' => array(
22             'label' => 'Name of the brand',
23         ),
24         'attributes' => array(
25             'required' => 'required',
26         ),
27     ));
28
29     $this->add(array(
30         'name' => 'url',
31         'type' => 'Zend\Form\Element\Url',
32         'options' => array(
33             'label' => 'Website of the brand',
34         ),
35         'attributes' => array(
36             'required' => 'required',
37         ),
38     ));
39 }
40
41 /**
42  * @return array
43  */
44 public function getInputFilterSpecification()
45 {
46     return array(
47         'name' => array(
48             'required' => true,
49         ),
50     );
51 }
52 }

```

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:

```

1  namespace Application\Form;
2
3  use Application\Entity\Category;
4  use Zend\Form\Fieldset;
5  use Zend\InputFilter\InputFilterProviderInterface;
6  use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;
7
8  class CategoryFieldset extends Fieldset implements InputFilterProviderInterface
9  {

```

```
10 public function __construct()
11 {
12     parent::__construct('category');
13
14     $this
15         ->setHydrator(new ClassMethodsHydrator(false))
16         ->setObject(new Category())
17     ;
18
19     $this->setLabel('Category');
20
21     $this->add(array(
22         'name' => 'name',
23         'options' => array(
24             'label' => 'Name of the category',
25         ),
26         'attributes' => array(
27             'required' => 'required',
28         ),
29     ));
30 }
31
32 /**
33  * @return array
34  */
35 public function getInputFilterSpecification()
36 {
37     return array(
38         'name' => array(
39             'required' => true,
40         ),
41     );
42 }
43 }
```

Nothing new here.

And finally the Product fieldset:

```
1 namespace Application\Form;
2
3 use Application\Entity\Product;
4 use Zend\Form\Fieldset;
5 use Zend\InputFilter\InputFilterProviderInterface;
6 use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;
7
8 class ProductFieldset extends Fieldset implements InputFilterProviderInterface
9 {
10     public function __construct()
11     {
12         parent::__construct('product');
13
14         $this
15             ->setHydrator(new ClassMethodsHydrator(false))
16             ->setObject(new Product())
17         ;
18
19         $this->add(array(
```

```

20         'name' => 'name',
21         'options' => array(
22             'label' => 'Name of the product',
23         ),
24         'attributes' => array(
25             'required' => 'required',
26         ),
27     ));
28
29     $this->add(array(
30         'name' => 'price',
31         'options' => array(
32             'label' => 'Price of the product',
33         ),
34         'attributes' => array(
35             'required' => 'required',
36         ),
37     ));
38
39     $this->add(array(
40         'type' => 'Application\Form\BrandFieldset',
41         'name' => 'brand',
42         'options' => array(
43             'label' => 'Brand of the product',
44         ),
45     ));
46
47     $this->add(array(
48         'type' => 'Zend\Form\Element\Collection',
49         'name' => 'categories',
50         'options' => array(
51             'label' => 'Please choose categories for this product',
52             'count' => 2,
53             'should_create_template' => true,
54             'allow_add' => true,
55             'target_element' => array(
56                 'type' => 'Application\Form\CategoryFieldset',
57             ),
58         ),
59     ));
60 }
61
62 /**
63  * Should return an array specification compatible with
64  * {@link Zend\InputFilter\Factory::createInputFilter()}.
65  *
66  * @return array
67  */
68 public function getInputFilterSpecification()
69 {
70     return array(
71         'name' => array(
72             'required' => true,
73         ),
74         'price' => array(
75             'required' => true,
76             'validators' => array(
77                 array(

```

```
78         'name' => 'Float',
79     ),
80 ),
81 ),
82 );
83 }
84 }
```

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 `` 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.

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 :

```
1  namespace Application\Form;
2
3  use Zend\Form\Form;
4  use Zend\InputFilter\InputFilter;
5  use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;
6
7  class CreateProduct extends Form
8  {
9      public function __construct()
10     {
11         parent::__construct('create_product');
12
13         $this
14             ->setAttribute('method', 'post')
15             ->setHydrator(new ClassMethodsHydrator(false))
16             ->setInputFilter(new InputFilter());
17     }
18 }
```



```

18
19     $this->add(array(
20         'type' => 'Application\Form\ProductFieldset',
21         'options' => array(
22             'use_as_base_fieldset' => true,
23         ),
24     ));
25
26     $this->add(array(
27         'type' => 'Zend\Form\Element\Csrf',
28         'name' => 'csrf',
29     ));
30
31     $this->add(array(
32         'name' => 'submit',
33         'attributes' => array(
34             'type' => 'submit',
35             'value' => 'Send',
36         ),
37     ));
38 }
39 }

```

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!

The Controller

Now, let’s create the action in the controller:

```

1  /**
2   * @return array
3   */
4  public function indexAction()
5  {
6      $form = new CreateProduct();
7      $product = new Product();
8      $form->bind($product);
9
10     $request = $this->getRequest();
11     if ($request->isPost()) {
12         $form->setData($request->getPost());
13
14         if ($form->isValid()) {
15             var_dump($product);
16         }
17     }
18
19     return array(

```

```
20         'form' => $form,  
21     );  
22 }
```

This is super easy. Nothing to do in the controllers. All the magic is done behind the scene.

The View

And finally, the view:

```
1 <?php  
2 $form->setAttribute('action', $this->url('home'))  
3     ->prepare();  
4  
5 echo $this->form()->openTag($form);  
6  
7 $product = $form->get('product');  
8  
9 echo $this->formRow($product->get('name'));  
10 echo $this->formRow($product->get('price'));  
11 echo $this->formCollection($product->get('categories'));  
12  
13 $brand = $product->get('brand');  
14  
15 echo $this->formRow($brand->get('name'));  
16 echo $this->formRow($brand->get('url'));  
17  
18 echo $this->formHidden($form->get('csrf'));  
19 echo $this->formElement($form->get('submit'));  
20  
21 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:

Name of the product

Price of the product

Please choose categories for this product

Category

Name of the category

Category

Name of the category

Name of the brand

Website of the brand

Envoyer

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 :

```
object(Application\Entity\Product)[ 622]
  protected 'name' => string 'Chair' (length=5)
  protected 'price' => string '25' (length=2)
  protected 'brand' =>
    object(Application\Entity\Brand)[ 597]
      protected 'name' => string 'Office Depot' (length=12)
      protected 'url' => string 'http://www.officedepot.fr' (length=25)
  protected 'categories' =>
    array (size=2)
      0 =>
        object(Application\Entity\Category)[ 615]
          protected 'name' => string 'Armchair' (length=8)
      1 =>
        object(Application\Entity\Category)[ 621]
          protected 'name' => string 'Office' (length=6)
```

As you can see, the bound object is completely filled, not with arrays, but with objects!

But that's not all.

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:

```
▼<fieldset>
  <legend>Please choose categories for this product</legend>
  ►<fieldset>...</fieldset>
  ►<fieldset>...</fieldset>
  <span data-template="<fieldset><legend>Category</legend><label>Name of the category<input name="
    product[categories][__index__][name]" required="required" type="text"></label></fieldset>"></span>
</fieldset>
```

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:

```
1 $this->add(array(
2     'type' => 'Zend\Form\Element\Collection',
3     'name' => 'categories',
4     'options' => array(
5         'label' => 'Please choose categories for this product',
6         'count' => 2,
7         'should_create_template' => true,
8         'template_placeholder' => '__placeholder__',
9         'target_element' => array(
10            'type' => 'Application\Form\CategoryFieldset',
11        ),
12    ),
13 ));
```

First, let's add a small button “Add new category” anywhere in the form:

```
1 <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:

```
1 <script>
2     function add_category() {
3         var currentCount = $('form > fieldset > fieldset').length;
4         var template = $('form > fieldset > span').data('template');
5         template = template.replace(/__index__/g, currentCount);
6
7         $('form > fieldset').append(template);
8
9         return false;
10    }
11 </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:

```
object(Application\Entity\Product)[ 622]
  protected 'name' => string 'Zend Framework' (length=14)
  protected 'price' => string '0' (length=1)
  protected 'brand' =>
    object(Application\Entity\Brand)[ 597]
      protected 'name' => string 'Zend' (length=4)
      protected 'url' => string 'http://www.zend.com' (length=19)
  protected 'categories' =>
    array (size=3)
      0 =>
        object(Application\Entity\Category)[ 615]
          protected 'name' => string 'Awesome' (length=7)
      1 =>
        object(Application\Entity\Category)[ 621]
          protected 'name' => string 'PHP' (length=3)
      2 =>
        object(Application\Entity\Category)[ 628]
          protected 'name' => string 'Framework' (length=9)
```

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:

```
1  $this->add(array(
2      'type' => 'Zend\Form\Element\Collection',
3      'name' => 'categories',
4      'options' => array(
5          'label' => 'Please choose categories for this product',
6          'count' => 2,
7          'should_create_template' => false,
8          'allow_add' => false,
9          'target_element' => array(
10             'type' => 'Application\Form\CategoryFieldset',
11         ),
12     ),
13 ));
```

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.

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:

```

1  <?php
2
3  $form
4      ->setAttribute('action', $this->url('home'))
5      ->prepare()
6  ;
7
8  echo $this->form()->openTag($form);
9
10 $product = $form->get('product');
11
12 echo $this->formRow($product->get('name'));
13 echo $this->formRow($product->get('price'));
14 echo $this->formCollection($product->get('categories'));
15
16 $brand = $product->get('brand');
17
18 echo $this->formRow($brand->get('name'));
19
20 echo $this->formHidden($form->get('csrf'));
21 echo $this->formElement($form->get('submit'));
22
23 echo $this->form()->closeTag();

```

This is what we get:

Name of the product

Price of the product

Please choose categories for this product

Category

Name of the category

Category

Name of the category

Name of the brand

Envoyer

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:

```

1  namespace Application\Form;
2
3  use Zend\Form\Form;
4  use Zend\InputFilter\InputFilter;
5  use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;
6
7  class CreateProduct extends Form
8  {
9      public function __construct()
10     {
11         parent::__construct('create_product');
12
13         $this
14             ->setAttribute('method', 'post')
15             ->setHydrator(new ClassMethodsHydrator())
16             ->setInputFilter(new InputFilter())
17         ;
18
19         $this->add(array(
20             'type' => 'Application\Form\ProductFieldset',
21             'options' => array(
22                 'use_as_base_fieldset' => true,
23             ),
24         ));
25
26         $this->add(array(
27             'type' => 'Zend\Form\Element\Csrf',
28             'name' => 'csrf',
29         ));
30
31         $this->add(array(
32             'name' => 'submit',
33             'attributes' => array(
34                 'type' => 'submit',
35                 'value' => 'Send',
36             ),
37         ));
38
39         $this->setValidationGroup(array(
40             'csrf',
41             'product' => array(
42                 'name',
43                 'price',
44                 'brand' => array(
45                     'name',
46                 ),
47             'categories' => array(
48                 'name',

```

```
49         ),
50     ),
51 );
52 }
53 }
```

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).

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.

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**.

The Form and InputFilter

Here we define a `Zend\Form\Element\File` input in a Form class named `UploadForm`.

```
1 // File: UploadForm.php
2
3 use Zend\Form\Element;
4 use Zend\Form\Form;
5
6 class UploadForm extends Form
7 {
8     public function __construct($name = null, $options = array())
9     {
10         parent::__construct($name, $options);
11         $this->addElements();
12     }
13
14     public function addElements()
15     {
16         // File Input
17         $file = new Element\File('image-file');
18         $file->setLabel('Avatar Image Upload')
19             ->setAttribute('id', 'image-file');
20         $this->add($file);
21     }
22 }
```

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.

The View Template

In the view template we render the <form>, a file input (with label and errors), and a submit button.

```
1 // File: upload-form.phtml
2 <?php $form->prepare(); // The correct enctype is set here ?>
3 <?php echo $this->form()->openTag($form); ?>
4
5 <div class="form-element">
6     <?php $fileElement = $form->get('image-file'); ?>
7     <?php echo $this->formLabel($fileElement); ?>
8     <?php echo $this->formFile($fileElement); ?>
9     <?php echo $this->formElementErrors($fileElement); ?>
10 </div>
11
12 <button>Submit</button>
13
14 <?php echo $this->form()->closeTag(); ?>
```

When rendered, the HTML should look similar to:

```
1 <form name="upload-form" id="upload-form" method="post" enctype="multipart/form-data">
2   <div class="form-element">
3     <label for="image-file">Avatar Image Upload</label>
```

```

4      <input type="file" name="image-file" id="image-file">
5      </div>
6
7      <button>Submit</button>
8  </form>

```

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.

```

1  // File: MyController.php
2
3  public function uploadFormAction()
4  {
5      $form = new UploadForm('upload-form');
6
7      $request = $this->getRequest();
8      if ($request->isPost()) {
9          // Make certain to merge the files info!
10         $post = array_merge_recursive(
11             $request->getPost()->toArray(),
12             $request->getFiles()->toArray()
13         );
14
15         $form->setData($post);
16         if ($form->isValid()) {
17             $data = $form->getData();
18             // Form is valid, save the form!
19             return $this->redirect()->toRoute('upload-form/success');
20         }
21     }
22
23     return array('form' => $form);
24 }

```

Upon a successful file upload, `$form->getData()` would return:

```

1  array(1) {
2      ["image-file"] => array(5) {
3          ["name"] => string(11) "myimage.png"
4          ["type"] => string(9) "image/png"
5          ["tmp_name"] => string(22) "/private/tmp/phpgRXd58"
6          ["error"] => int(0)
7          ["size"] => int(14908679)
8      }
9  }

```

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:

```
1 // A $_FILES array with single input and multiple files:
2 array(1) {
3     ["image-file"]=>array(2) {
4         ["name"]=>array(2) {
5             [0]=>string(9) "file0.txt"
6             [1]=>string(9) "file1.txt"
7         }
8         ["type"]=>array(2) {
9             [0]=>string(10) "text/plain"
10            [1]=>string(10) "text/html"
11        }
12    }
13 }
14
15 // How Zend\Http\PhpEnvironment\Request remaps the $_FILES array:
16 array(1) {
17     ["image-file"]=>array(2) {
18         [0]=>array(2) {
19             ["name"]=>string(9) "file0.txt"
20             ["type"]=>string(10) "text/plain"
21         },
22         [1]=>array(2) {
23             ["name"]=>string(9) "file1.txt"
24             ["type"]=>string(10) "text/html"
25         }
26     }
27 }
```

Zend\InputFilter\FileInput expects the file data be in this re-mapped array format.

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:

```
1 // File: UploadForm.php
2
3 use Zend\InputFilter;
4 use Zend\Form\Element;
5 use Zend\Form\Form;
6
7 class UploadForm extends Form
8 {
9     public function __construct($name = null, $options = array())
```

```

10 {
11     parent::__construct($name, $options);
12     $this->addElements();
13     $this->addInputFilter();
14 }
15
16 public function addElements()
17 {
18     // File Input
19     $file = new Element\File('image-file');
20     $file->setLabel('Avatar Image Upload')
21         ->setAttribute('id', 'image-file');
22     $this->add($file);
23 }
24
25 public function addInputFilter()
26 {
27     $inputFilter = new InputFilter\InputFilter();
28
29     // File Input
30     $fileInput = new InputFilter\FileInput('image-file');
31     $fileInput->setRequired(true);
32     $fileInput->getFilterChain()->attachByName(
33         'filerenameupload',
34         array(
35             'target' => './data/tmpuploads/avatar.png',
36             'randomize' => true,
37         )
38     );
39     $inputFilter->add($fileInput);
40
41     $this->setInputFilter($inputFilter);
42 }
43 }

```

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:

```

1 // File: MyController.php
2
3 public function uploadFormAction()
4 {
5     $form      = new UploadForm('upload-form');
6     $tempFile  = null;
7
8     $prg = $this->fileprg($form);
9     if ($prg instanceof \Zend\Http\PhpEnvironment\Response) {
10         return $prg; // Return PRG redirect response
11     } elseif (is_array($prg)) {
12         if ($form->isValid()) {
13             $data = $form->getData();
14             // Form is valid, save the form!
15             return $this->redirect()->toRoute('upload-form/success');
16         } else {
17             // Form not valid, but file uploads might be valid...

```

```
18         // Get the temporary file information to show the user in the view
19         $fileErrors = $form->get('image-file')->getMessages();
20         if (empty($fileErrors)) {
21             $tempFile = $form->get('image-file')->getValue();
22         }
23     }
24 }
25
26 return array(
27     'form' => $form,
28     'tempFile' => $tempFile,
29 );
30 }
```

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.

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`:

```
1 // File: UploadForm.php
2
3 use Zend\InputFilter;
4 use Zend\Form\Element;
5 use Zend\Form\Form;
6
7 class UploadForm extends Form
8 {
9     public function __construct($name = null, $options = array())
10     {
11         parent::__construct($name, $options);
12         $this->addElements();
13         $this->addInputFilter();
14     }
15
16     public function addElements()
17     {
18         // File Input
```

```

19     $file = new Element\File('image-file');
20     $file->setLabel('Avatar Image Upload')
21         ->setAttribute('id', 'image-file')
22         ->setAttribute('multiple', true);    // That's it
23     $this->add($file);
24 }
25
26 public function addInputFilter()
27 {
28     $inputFilter = new InputFilter\InputFilter();
29
30     // File Input
31     $fileInput = new InputFilter\FileInput('image-file');
32     $fileInput->setRequired(true);
33
34     // You only need to define validators and filters
35     // as if only one file was being uploaded. All files
36     // will be run through the same validators and filters
37     // automatically.
38     $fileInput->getValidatorChain()
39         ->attachByName('filesize', array('max' => 204800))
40         ->attachByName('fileimimetype', array('mimeType' => 'image/png,image/x-png
41         ->attachByName('fileimagesize', array('maxWidth' => 100, 'maxHeight' =>
42         ->100));
43
44     // All files will be renamed, i.e.:
45     // ./data/tmpuploads/avatar_4b3403665fea6.png,
46     // ./data/tmpuploads/avatar_5c45147660fb7.png
47     $fileInput->getFilterChain()->attachByName(
48         'filerenameupload',
49         array(
50             'target' => './data/tmpuploads/avatar.png',
51             'randomize' => true,
52         )
53     );
54     $inputFilter->add($fileInput);
55
56     $this->setInputFilter($inputFilter);
57 }

```

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.

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.

```
1 // File: upload-form.phtml
2 <?php $form->prepare(); ?>
3 <?php echo $this->form()->openTag($form); ?>
4     <?php echo $this->formFileSessionProgress(); // Must come before the file input! ?
5     ↪>
6     <div class="form-element">
7         <?php $fileElement = $form->get('image-file'); ?>
8         <?php echo $this->formLabel($fileElement); ?>
9         <?php echo $this->formFile($fileElement); ?>
10        <?php echo $this->formElementErrors($fileElement); ?>
11    </div>
12
13    <button>Submit</button>
14
15 <?php echo $this->form()->closeTag(); ?>
```

When rendered, the HTML should look similar to:

```
1 <form name="upload-form" id="upload-form" method="post" enctype="multipart/form-data">
2     <input type="hidden" id="progress_key" name="PHP_SESSION_UPLOAD_PROGRESS" value=
3     ↪ "12345abcde">
4
5     <div class="form-element">
6         <label for="image-file">Avatar Image Upload</label>
7         <input type="file" name="image-file" id="image-file">
8     </div>
9
10    <button>Submit</button>
11</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:

```
1 // File: MyController.php
2
3 public function uploadProgressAction()
```



```

4 {
5     $id = $this->params()->fromQuery('id', null);
6     $progress = new \Zend\ProgressBar\Upload\SessionProgress();
7     return new \Zend\View\Model\JsonModel($progress->getProgress($id));
8 }
9
10 // Returns JSON
11 //{
12 //     "total"      : 204800,
13 //     "current"    : 10240,
14 //     "rate"       : 1024,
15 //     "message"    : "10kB / 200kB",
16 //     "done"       : false
17 //}
    
```

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.

```

1 // File: upload-form.phtml
2 // ...after the form...
3
4 <!-- Twitter Bootstrap progress bar styles:
5     http://twitter.github.com/bootstrap/components.html#progress -->
6 <div id="progress" class="help-block">
7     <div class="progress progress-info progress-striped">
8         <div class="bar"></div>
9     </div>
10    <p></p>
11 </div>
12
13 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js"></
14 <script>
15 <script>
16 var progressInterval;
17
18 function getProgress() {
19     // Poll our controller action with the progress id
20     var url = '/upload-form/upload-progress?id=' + $('#progress_key').val();
21     $.getJSON(url, function(data) {
22         if (data.status && !data.status.done) {
23             var value = Math.floor((data.status.current / data.status.total) * 100);
24             showProgress(value, 'Uploading...');
25         } else {
26             showProgress(100, 'Complete!');
27             clearInterval(progressInterval);
28         }
29     });
30 }
    
```

```

31
32 function startProgress() {
33     showProgress(0, 'Starting upload...');
34     progressInterval = setInterval(getProgress, 900);
35 }
36
37 function showProgress(amount, message) {
38     $('#progress').show();
39     $('#progress .bar').width(amount + '%');
40     $('#progress > p').html(message);
41     if (amount < 100) {
42         $('#progress .progress')
43             .addClass('progress-info active')
44             .removeClass('progress-success');
45     } else {
46         $('#progress .progress')
47             .removeClass('progress-info active')
48             .addClass('progress-success');
49     }
50 }
51
52 $(function() {
53     // Register a 'submit' event listener on the form to perform the AJAX POST
54     $('#upload-form').on('submit', function(e) {
55         e.preventDefault();
56
57         if ($('#image-file').val() == '') {
58             // No files selected, abort
59             return;
60         }
61
62         // Perform the submit
63         //$fn.ajaxSubmit.debug = true;
64         $(this).ajaxSubmit({
65             beforeSend: function(arr, $form, options) {
66                 // Notify backend that submit is via ajax
67                 arr.push({ name: "isAjax", value: "1" });
68             },
69             success: function (response, statusText, xhr, $form) {
70                 clearInterval(progressInterval);
71                 showProgress(100, 'Complete!');
72
73                 // TODO: You'll need to do some custom logic here to handle a
74                 ↪ successful
75                 // form post, and when the form is invalid with validation errors.
76                 if (response.status) {
77                     // TODO: Do something with a successful form post, like redirect
78                     // window.location.replace(response.redirect);
79                 } else {
80                     // Clear the file input, otherwise the same file gets re-uploaded
81                     // http://stackoverflow.com/a/1043969
82                     var fileInput = $('#image-file');
83                     fileInput.replaceWith( fileInput.val('').clone( true ) );
84
85                     // TODO: Do something with these errors
86                     // showErrors(response.formErrors);
87                 }
88             }
89         });
90     });
91 }

```

```

88         error: function(a, b, c) {
89             // NOTE: This callback is *not* called when the form is invalid.
90             // It is called when the browser is unable to initiate or complete
91             ↪the ajax submit.
92             // You will need to handle validation errors in the 'success'
93             ↪callback.
94             console.log(a, b, c);
95         }
96     });
97     // Start the progress polling
98     startProgress();
99 }
</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):

```

1 // File: MyController.php
2
3 public function uploadFormAction()
4 {
5     $form = new UploadForm('upload-form');
6
7     $request = $this->getRequest();
8     if ($request->isPost()) {
9         // Make certain to merge the files info!
10        $post = array_merge_recursive(
11            $request->getPost()->toArray(),
12            $request->getFiles()->toArray()
13        );
14
15        $form->setData($post);
16        if ($form->isValid()) {
17            $data = $form->getData();
18            // Form is valid, save the form!
19            if (!empty($post['isAjax'])) {
20                return new JsonModel(array(
21                    'status' => true,
22                    'redirect' => $this->url()->fromRoute('upload-form/success'),
23                    'formData' => $data,
24                ));
25            } else {
26                // Fallback for non-JS clients
27                return $this->redirect()->toRoute('upload-form/success');
28            }
29        } else {
30            if (!empty($post['isAjax'])) {
31                // Send back failure information via JSON
32                return new JsonModel(array(
33                    'status' => false,
34                    'formErrors' => $form->getMessages(),
35                    'formData' => $form->getData(),
36                ));
37            }
38        }
39    }
40 }

```

```
41     return array('form' => $form);  
42 }
```

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.

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.

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:

```
1 $form->add(array(  
2     'type' => 'Zend\Form\Element\Email',  
3     'name' => 'email'  
4 ));
```

can now be replaced by:

```
1 $form->add(array(  
2     'type' => 'Email',  
3     'name' => 'email'  
4 ));
```

Each element provided out-of-the-box by Zend Framework 2 support this natively, so you can now make your initialization code more compact.

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:

```
1 namespace Application\Form\Element;
2
3 use Zend\Form\Element;
4 use Zend\InputFilter\InputProviderInterface;
5 use Zend\Validator\Regex as RegexValidator;
6
7 class Phone extends Element implements InputProviderInterface
8 {
9     /**
10      * @var ValidatorInterface
11      */
12     protected $validator;
13
14     /**
15      * Get a validator if none has been set.
16      *
17      * @return ValidatorInterface
18      */
19     public function getValidator()
20     {
21         if (null === $this->validator) {
22             $validator = new RegexValidator('/^\+?\d{11,12}$/');
23             $validator->setMessage('Please enter 11 or 12 digits only!',
24                                   RegexValidator::NOT_MATCH);
25
26             $this->validator = $validator;
27         }
28
29         return $this->validator;
30     }
31
32     /**
33      * Sets the validator to use for this element
34      *
35      * @param ValidatorInterface $validator
36      * @return Application\Form\Element\Phone
37      */
38     public function setValidator(ValidatorInterface $validator)
39     {
40         $this->validator = $validator;
41         return $this;
42     }
43
44     /**
45      * Provide default input rules for this element
46      *
47      * Attaches a phone number validator.
48      *
49      * @return array
50      */
51     public function getInputSpecification()
52     {
53         return array(
54             'name' => $this->getName(),
```

```

55         'required' => true,
56         'filters' => array(
57             array('name' => 'Zend\Filter\StringTrim'),
58         ),
59         'validators' => array(
60             $this->getValidator(),
61         ),
62     );
63 }
64 }
    
```

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 to start using your new custom element in your forms is to use the custom element's FQCN:

```

1  $form = new Zend\Form\Form();
2  $form->add(array(
3      'name' => 'phone',
4      'type' => 'Application\Form\Element\Phone',
5  ));
    
```

Or, if you are extending `Zend\Form\Form`:

```

1  namespace Application\Form;
2
3  use Zend\Form\Form;
4
5  class MyForm extends Form
6  {
7      public function __construct($name = null)
8      {
9          parent::__construct($name);
10
11          $this->add(array(
12              'name' => 'phone',
13              'type' => 'Application\Form\Element\Phone',
14          ))
15      }
16  }
    
```

If you don't want to use the custom element's FQCN, 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:

```

1  namespace Application;
2
3  use Zend\ModuleManager\Feature\FormElementProviderInterface;
4
    
```

```
5 class Module implements FormElementProviderInterface
6 {
7     public function getFormElementConfig()
8     {
9         return array(
10             'invokables' => array(
11                 'phone' => 'Application\Form\Element\Phone'
12             )
13         );
14     }
15 }
```

Or, you can do the same in your `module.config.php` file:

```
1 return array(
2     'form_elements' => array(
3         'invokables' => array(
4             'phone' => 'Application\Form\Element\Phone'
5         )
6     )
7 );
```

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 FQCN), but rather in the `init()` method:

```
1 namespace Application\Form;
2
3 use Zend\Form\Form;
4
5 class MyForm extends Form
6 {
7     public function init()
8     {
9         $this->add(array(
10             'name' => 'phone',
11             'type' => 'Phone',
12         ));
13     }
14 }
```

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`:

```
1 namespace Application\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4
5 class IndexController extends AbstractActionController
6 {
7     public function indexAction()
8     {
9         $sl = $this->getServiceLocator();
10         $form = $sl->get('FormElementManager')->get('Application\Form\MyForm');
```



```

11     return array('form' => $form);
12 }
13 }

```

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:

```

1 namespace Application;
2
3 use Zend\ModuleManager\Feature\FormElementProviderInterface;
4
5 class Module implements FormElementProviderInterface
6 {
7     public function getFormElementConfig()
8     {
9         return array(
10             'invokables' => array(
11                 'Email' => 'Application\Form\Element\MyEmail'
12             )
13         );
14     }
15 }

```

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 `Application\Form\MyForm` 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()`, but rather in its `init()` method, or after getting an instance of the form.
4. Create your form through the form element manager instead of directly instantiating it.

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`:

```
1 namespace Application\Form;
2
3 use Zend\Form\Form;
4
5 class CreateAlbum extends Form
6 {
7     public function init()
8     {
9         $this->add(array(
10             'name' => 'album',
11             'type' => 'AlbumFieldset'
12         ));
13     }
14 }
```

Let's now create the `AlbumFieldset` that depends on an `AlbumTable` object that allows you to fetch albums from the database.

```
1 namespace Application\Form;
2
3 use Album\Model\AlbumTable;
4 use Zend\Form\Fieldset;
5
6 class AlbumFieldset extends Fieldset
7 {
8     public function __construct(AlbumTable $albumTable)
9     {
10         // Add any elements that need to fetch data from database
11         // using the album table !
12     }
13 }
```

For this to work, you need to add a line to the form element manager by adding an element to your `Module.php` class:

```
1 namespace Application;
2
3 use Application\Form\AlbumFieldset;
4 use Zend\ModuleManager\Feature\FormElementProviderInterface;
5
6 class Module implements FormElementProviderInterface
7 {
8     public function getFormElementConfig()
9     {
10         return array(
11             'factories' => array(
12                 'AlbumFieldset' => function($sm) {
13                     // I assume here that the Album\Model\AlbumTable
14                     // dependency have been defined too
15
16                     $serviceLocator = $sm->getServiceLocator();
17                     $albumTable = $serviceLocator->get('Album\Model\AlbumTable');
18                     $fieldset = new AlbumFieldset($albumTable);
19                     return $fieldset;
20                 }
21             )
22         );
23     }
24 }
```

```

21         )
22     );
23 }
24 }

```

Create your form using the form element manager instead of directly instantiating it:

```

1 public function testAction()
2 {
3     $formManager = $this->serviceLocator->get('FormElementManager');
4     $form = $formManager->get('Application\Form\CreateAlbum');
5 }

```

Finally, to use your fieldset in a view you need to use the formCollection function.

```

1 echo $this->form()->openTag($form);
2 echo $this->formCollection($form->get('album'));
3 echo $this->form()->closeTag();

```

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.

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**:

```

1 namespace Application\Form;
2
3 use Album\Model;
4 use Zend\Form\Fieldset;
5 use Zend\ServiceManager\ServiceLocatorAwareInterface;
6 use Zend\ServiceManager\ServiceLocatorInterface;
7
8 class AlbumFieldset extends Fieldset implements ServiceLocatorAwareInterface
9 {
10     protected $serviceLocator;
11
12     public function __construct()
13     {
14         // Here, $this->serviceLocator is null because it has not been
15         // injected yet, as initializers are run after __construct
16     }
17
18     public function setServiceLocator(ServiceLocatorInterface $sl)
19     {
20         $this->serviceLocator = $sl;
21     }
22
23     public function getServiceLocator()
24     {

```

```
25     return $this->serviceLocator;
26 }
27 }
```

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:

```
1 namespace Application\Form;
2
3 use Album\Model;
4 use Zend\Form\Fieldset;
5 use Zend\ServiceManager\ServiceLocatorAwareInterface;
6 use Zend\ServiceManager\ServiceLocatorInterface;
7
8 class AlbumFieldset extends Fieldset implements ServiceLocatorAwareInterface
9 {
10     protected $serviceLocator;
11
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 }
```

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 *ZendForm Quick Start* for more information.

orphan

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.

```
1 use Zend\Form\Element;  
2 use Zend\Form\Form;  
3  
4 $username = new Element\Text('username');  
5 $username  
6     ->setLabel('Username')  
7     ->setAttributes(array(  
8         'class' => 'username',  
9         'size'  => '30',  
10    ));
```

```
11
12 $password = new Element\Password('password');
13 $password
14     ->setLabel('Password')
15     ->setAttributes(array(
16         'size' => '30',
17     ));
18
19 $form = new Form('my-form');
20 $form
21     ->add($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

setLabelOptions (*array \$labelOptions*)

Set label specific options.

getLabelOptions ()

Return the label specific options.

Return type array

setOptions (*array \$options*)

Set options for an element. Accepted options are: label, label_attributes", label_options, which call setLabel, setLabelAttributes and setLabelOptions, 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

Standard Elements

orphan

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`.

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $button = new Element\Button('my-button');
5 $button->setLabel('My Button')
6         ->setValue('foo');
7
8 $form = new Form('my-form');
9 $form->add($button);
```

orphan

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.

```
1 use Zend\Captcha;
2 use Zend\Form\Element;
3 use Zend\Form\Form;
4
5 $captcha = new Element\Captcha('captcha');
6 $captcha
7     ->setCaptcha(new Captcha\Dumb())
8     ->setLabel('Please verify you are human');
9
10 $form = new Form('my-form');
11 $form->add($captcha);
```

Here is with the array notation:

```
1 use Zend\Captcha;
2 use Zend\Form\Form;
3
4 $form = new Form('my-form');
5 $form->add(array(
6     'type' => 'Zend\Form\Element\Captcha',
7     'name' => 'captcha',
8     'options' => array(
9         'label' => 'Please verify you are human',
10        'captcha' => new Captcha\Dumb(),
11    ),
12 ));
```


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

orphan

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".

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $checkbox = new Element\Checkbox('checkbox');
5 $checkbox->setLabel('A checkbox');
6 $checkbox->setUseHiddenElement(true);
7 $checkbox->setCheckedValue('good');
8 $checkbox->setUncheckedValue('bad');
9
10 $form = new Form('my-form');
11 $form->add($checkbox);

```

Using the array notation:

```

1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Checkbox',
6     'name' => 'checkbox',
7     'options' => array(
8         'label' => 'A checkbox',
9         'use_hidden_element' => true,
10        'checked_value' => 'good',
11        'unchecked_value' => 'bad'

```

```
12     )
13  ));
```

When creating a checkbox element, setting an attribute of checked will result in the checkbox always being checked regardless of any data object which might subsequently be bound to the form. The correct way to set the default value of a checkbox is to set the value attribute as for any other element. To have a checkbox checked by default make the value equal to the checked_value eg:

```
1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\Checkbox',
6      'name' => 'checkbox',
7      'options' => array(
8          'label' => 'A checkbox',
9          'use_hidden_element' => true,
10         'checked_value' => 'yes',
11         'unchecked_value' => 'no'
12     ),
13     'attributes' => array(
14         'value' => 'yes'
15     )
16 ));
```

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.

orphan

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

```

1  use Zend\Form\Element;
2  use Zend\Form\Form;
3
4  $colors = new Element\Collection('collection');
5  $colors->setLabel('Colors');
6  $colors->setCount(2);
7  $colors->setTargetElement(new Element\Color());
8  $colors->setShouldCreateTemplate(true);
9
10 $form = new Form('my-form');
11 $form->add($colors);

```

Using the array notation:

```

1  use Zend\Form\Element;
2  use Zend\Form\Form;
3
4  $form = new Form('my-form');
5  $form->add(array(
6      'type' => 'Zend\Form\Element\Collection',
7      'options' => array(
8          'label' => 'Colors',
9          'count' => 2,
10         'should_create_template' => true,
11         'target_element' => new Element\Color()
12     )
13 ));

```

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 *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

orphan

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"`.

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $csrf = new Element\Csrf('csrf');
5
6 $form = new Form('my-form');
7 $form->add($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:

```
1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
```

```
5     'type' => 'Zend\Form\Element\Csrf',
6     'name' => 'csrf',
7     'options' => array(
8         'csrf_options' => array(
9             'timeout' => 600
10        )
11    )
12 );
```

Note: If you are using more than one form on a page, and each contains its own CSRF element, you will need to make sure that each form uniquely names its element; if you do not, it's possible for the value of one to override the other within the server-side session storage, leading to the inability to validate one or more of the forms on your page. We suggest prefixing the element name with the form's name or function: "login_csrf", "registration_csrf", etc.

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(Zend\Validator\Csrf \$validator)

Override the default CSRF validator by setting another one.

getCsrfValidator()

Get the CSRF validator.

Return type `Zend\Validator\Csrf`

orphan

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()`.

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 // Single file upload
5 $file = new Element\File('file');
6 $file->setLabel('Single file input');
7
8 // HTML5 multiple file upload
9 $multiFile = new Element\File('multi-file');
10 $multiFile->setLabel('Multi file input')
11     ->setAttribute('multiple', true);
12
13 $form = new Form('my-file');
14 $form->add($file)
15     ->add($multiFile);

```

orphan

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 *ZendForm\Element*.

Basic Usage

This element automatically adds a "type" attribute of value "hidden".

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $hidden = new Element\Hidden('my-hidden');
5 $hidden->setValue('foo');
6
7 $form = new Form('my-form');
8 $form->add($hidden);

```

Here is with the array notation:

```

1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Hidden',
6     'name' => 'my-hidden',
7     'attributes' => array(
8         'value' => 'foo'
9     )
10 ));

```

orphan

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".

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $image = new Element\Image('my-image');
5 $image->setAttribute('src', 'http://my.image.url'); // Src attribute is required
6
7 $form = new Form('my-form');
8 $form->add($image);
```

orphan

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

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $monthYear = new Element\MonthSelect('monthyear');
5 $monthYear->setLabel('Select a month and a year');
6 $monthYear->setMinYear(1986);
7
8 $form = new Form('dateselect');
9 $form->add($monthYear);
```

Using the array notation:

```
1 use Zend\Form\Form;
2
3 $form = new Form('dateselect');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\MonthSelect',
6     'name' => 'monthyear',
7     'options' => array(
8         'label' => 'Select a month and a year',
9         'min_year' => 1986,
10    )
11 ));
```


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.

orphan

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.

```
1  use Zend\Form\Element;
2  use Zend\Form\Form;
3
4  $multiCheckbox = new Element\MultiCheckbox('multi-checkbox');
5  $multiCheckbox->setLabel('What do you like ?');
6  $multiCheckbox->setValueOptions(array(
7      '0' => 'Apple',
8      '1' => 'Orange',
9      '2' => 'Lemon'
10 ));
11
12 $form = new Form('my-form');
13 $form->add($multiCheckbox);
```

Using the array notation:

```
1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\MultiCheckbox',
6      'name' => 'multi-checkbox',
7      'options' => array(
8          'label' => 'What do you like ?',
9          'value_options' => array(
10             '0' => 'Apple',
11             '1' => 'Orange',
12             '2' => 'Lemon',
13         ),
14     )
15 ));
```

Advanced Usage

In order to set attributes or customize the option elements, an array can be used instead of a string. The following keys are supported:

- "label" - The string displayed for the option.
- "value" - The form value associated with the option.
- "selected" - Boolean that sets whether the option is marked as selected.
- "disabled" - Boolean that sets whether the option will be disabled
- "attributes" - Array of html attributes that will be set on this option. Merged with the attributes set on the element.
- "label_attributes" - Array of html attributes that will be set on the label. Merged with the attributes set on the element's label.

```
1  $form = new Form('my-form');
2  $form->add(array(
3      'type' => 'Zend\Form\Element\MultiCheckbox',
```

```

4      'name' => 'multi-checkbox',
5      'options' => array(
6          'label' => 'What do you like ?',
7          'value_options' => array(
8              array(
9                  'value' => '0',
10                 'label' => 'Apple',
11                 'selected' => false,
12                 'disabled' => false,
13                 'attributes' => array(
14                     'id' => 'apple_option',
15                     'data-fruit' => 'apple',
16                 ),
17                 'label_attributes' => array(
18                     'id' => 'apple_label',
19                 ),
20             ),
21             array(
22                 'value' => '1',
23                 'label' => 'Orange',
24                 'selected' => true,
25             ),
26             array(
27                 'value' => '2',
28                 'label' => 'Lemon',
29             ),
30         ),
31     ),
32 );

```

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

unsetValueOption (\$key)

Unset the value option from the multi-checkbox.

orphan

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".

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $password = new Element\Password('my-password');
5 $password->setLabel('Enter your password');
6
7 $form = new Form('my-form');
8 $form->add($password);
```

orphan

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.

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $radio = new Element\Radio('gender');
5 $radio->setLabel('What is your gender ?');
6 $radio->setValueOptions(array(
7     '0' => 'Female',
8     '1' => 'Male',
9 ));
10
11 $form = new Form('my-form');
12 $form->add($radio);
```

Using the array notation:

```
1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Radio',
6     'name' => 'gender',
7     'options' => array(
8         'label' => 'What is your gender ?',
9         'value_options' => array(
10             '0' => 'Female',
11             '1' => 'Male',
12         ),
13     ),
14 );
```

```

13         ),
14     ));

```

Advanced Usage

See *MultiCheckbox* for examples of how to apply attributes and options to each radio button.

Public Methods

All the methods from the inherited *methods of Zend\Form\Element\MultiCheckbox* are also available for this element.

orphan

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".

```

1     use Zend\Form\Element;
2     use Zend\Form\Form;
3
4     $select = new Element\Select('language');
5     $select->setLabel('Which is your mother tongue?');
6     $select->setValueOptions(array(
7         '0' => 'French',
8         '1' => 'English',
9         '2' => 'Japanese',
10        '3' => 'Chinese',
11    ));
12
13    $form = new Form('language');
14    $form->add($select);

```

Using the array notation:

```

1     use Zend\Form\Form;
2
3     $form = new Form('my-form');
4     $form->add(array(
5         'type' => 'Zend\Form\Element\Select',
6         'name' => 'language',
7         'options' => array(
8             'label' => 'Which is your mother tongue?',
9             'value_options' => array(
10                '0' => 'French',
11                '1' => 'English',

```

```

12         '2' => 'Japanese',
13         '3' => 'Chinese',
14     ),
15     )
16 );
    
```

You can add an empty option (option with no value) using the "empty_option" option:

```

1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\Select',
6      'name' => 'language',
7      'options' => array(
8          'label' => 'Which is your mother tongue?',
9          'empty_option' => 'Please choose your language',
10         'value_options' => array(
11             '0' => 'French',
12             '1' => 'English',
13             '2' => 'Japanese',
14             '3' => 'Chinese',
15         ),
16     ),
17 );
    
```

Option groups are also supported. You just need to add an 'options' key to the value options.

```

1  use Zend\Form\Element;
2  use Zend\Form\Form;
3
4  $select = new Element\Select('language');
5  $select->setLabel('Which is your mother tongue?');
6  $select->setValueOptions(array(
7      'european' => array(
8          'label' => 'European languages',
9          'options' => array(
10             '0' => 'French',
11             '1' => 'Italian',
12         ),
13     ),
14     'asian' => array(
15         'label' => 'Asian languages',
16         'options' => array(
17             '2' => 'Japanese',
18             '3' => 'Chinese',
19         ),
20     ),
21 ));
22
23 $form = new Form('language');
24 $form->add($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. Accepted options, in addition to the inherited *options of Zend\Form\Element*, are: "value_options", "empty_option" and "disable_inarray_validator", which call *setValueOptions*, *setEmptyOption* and *setDisableInArrayValidator*, 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

unsetValueOption (*\$key*)

Unset the value option from the select element.

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

orphan

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 *ZendForm\Element*.

Basic Usage

This element automatically adds a "type" attribute of value "submit".

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $submit = new Element\Submit('my-submit');
5 $submit->setValue('Submit Form');
6
7 $form = new Form('my-form');
8 $form->add($submit);

```

orphan

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 *ZendForm\Element*.

Basic Usage

This element automatically adds a "type" attribute of value "text".

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $text = new Element\Text('my-text');
5 $text->setLabel('Enter your name');
6
7 $form = new Form('my-form');
8 $form->add($text);
```

orphan

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 *ZendFormElement*.

Basic Usage

This element automatically adds a "type" attribute of value "textarea".

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $textarea = new Element\Textarea('my-textarea');
5 $textarea->setLabel('Enter a description');
6
7 $form = new Form('my-form');
8 $form->add($textarea);
```

HTML5 Elements

orphan

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 its 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".


```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $color = new Element\Color('color');
5 $color->setLabel('Background color');
6
7 $form = new Form('my-form');
8 $form->add($color);

```

Here is the same example using the array notation:

```

1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Color',
6     'name' => 'color',
7     'options' => array(
8         'label' => 'Background color'
9     )
10 ));

```

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

orphan

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".

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $date = new Element\Date('appointment-date');
5 $date
6     ->setLabel('Appointment Date')
7     ->setAttributes(array(
8         'min' => '2012-01-01',
9         'max' => '2020-01-01',

```

```
10         'step' => '1', // days; default step interval is 1 day
11     ))
12     ->setOptions(array(
13         'format' => 'Y-m-d'
14     ));
15
16 $form = new Form('my-form');
17 $form->add($date);
```

Here is with the array notation:

```
1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\Date',
6      'name' => 'appointment-date',
7      'options' => array(
8          'label' => 'Appointment Date',
9          'format' => 'Y-m-d'
10     ),
11     'attributes' => array(
12         'min' => '2012-01-01',
13         'max' => '2020-01-01',
14         'step' => '1', // days; default step interval is 1 day
15     )
16 ));
```

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

orphan

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 its 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".

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $dateTime = new Element\DateTime('appointment-date-time');
5 $dateTime
6     ->setLabel('Appointment Date/Time')
7     ->setAttributes(array(
8         'min' => '2010-01-01T00:00:00Z',
9         'max' => '2020-01-01T00:00:00Z',
10        'step' => '1', // minutes; default step interval is 1 min
11    ))
12    ->setOptions(array(
13        'format' => 'Y-m-d\TH:iP'
14    ));
15
16 $form = new Form('my-form');
17 $form->add($dateTime);

```

Here is with the array notation:

```

1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\DateTime',
6     'name' => 'appointment-date-time',
7     'options' => array(
8         'label' => 'Appointment Date/Time',
9         'format' => 'Y-m-d\TH:iP'
10    ),
11     'attributes' => array(
12         'min' => '2010-01-01T00:00:00Z',
13         'max' => '2020-01-01T00:00:00Z',
14         'step' => '1', // minutes; default step interval is 1 min
15    )
16 ));

```

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

orphan

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 its 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”.

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $dateTimeLocal = new Element\DateTimeLocal('appointment-date-time');
5 $dateTimeLocal
6     ->setLabel('Appointment Date')
7     ->setAttributes(array(
8         'min' => '2010-01-01T00:00:00',
9         'max' => '2020-01-01T00:00:00',
10        'step' => '1', // minutes; default step interval is 1 min
11    ))
12    ->setOptions(array(
13        'format' => 'Y-m-d\TH:i'
14    ));
15
16 $form = new Form('my-form');
17 $form->add($dateTimeLocal);
```

Here is with the array notation:

```

1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\DateTimeLocal',
6      'name' => 'appointment-date-time',
7      'options' => array(
8          'label' => 'Appointment Date',
9          'format' => 'Y-m-d\TH:i'
10     ),
11     'attributes' => array(
12         'min' => '2010-01-01T00:00:00',
13         'max' => '2020-01-01T00:00:00',
14         'step' => '1', // minutes; default step interval is 1 min
15     )
16 ));

```

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

orphan

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"`.

```

1  use Zend\Form\Element;
2  use Zend\Form\Form;
3
4  $form = new Form('my-form');
5
6  // Single email address
7  $email = new Element\Email('email');
8  $email->setLabel('Email Address');
9  $form->add($email);

```

```
10 // Comma separated list of emails
11
12 $emails = new Element\Email('emails');
13 $emails
14     ->setLabel('Email Addresses')
15     ->setAttribute('multiple', true);
16 $form->add($emails);
```

Here is with the array notation:

```
1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Email',
6     'name' => 'email',
7     'options' => array(
8         'label' => 'Email Address'
9     ),
10 ));
11
12 $form->add(array(
13     'type' => 'Zend\Form\Element\Email',
14     'name' => 'emails',
15     'options' => array(
16         'label' => 'Email Addresses'
17     ),
18     'attributes' => array(
19         'multiple' => true
20     )
21 ));
```

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.

orphan

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 its 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".

```

1  use Zend\Form\Element;
2  use Zend\Form\Form;
3
4  $month = new Element\Month('month');
5  $month
6      ->setLabel('Month')
7      ->setAttributes(array(
8          'min' => '2012-01',
9          'max' => '2020-01',
10         'step' => '1', // months; default step interval is 1 month
11     ));
12
13 $form = new Form('my-form');
14 $form->add($month);
    
```

Here is with the array notation:

```

1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\Month',
6      'name' => 'month',
7      'options' => array(
8          'label' => 'Month'
9      ),
10     'attributes' => array(
11         'min' => '2012-12',
12         'max' => '2020-01',
13         'step' => '1', // months; default step interval is 1 month
14     )
15 ));
    
```

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 an 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

orphan

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 its 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"`.

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $number = new Element\Number('quantity');
5 $number
6     ->setLabel('Quantity')
7     ->setAttributes(array(
8         'min' => '0',
9         'max' => '10',
10        'step' => '1', // default step interval is 1
11    ));
12
13 $form = new Form('my-form');
14 $form->add($number);
```

Here is with the array notation:

```
1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Number',
6     'name' => 'quantity',
7     'options' => array(
```



```

8         'label' => 'Quantity'
9     ),
10    'attributes' => array(
11        'min' => '0',
12        'max' => '10',
13        'step' => '1', // default step interval is 1
14    )
15 );

```

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

orphan

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”.

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $range = new Element\Range('range');
5 $range
6     ->setLabel('Minimum and Maximum Amount')
7     ->setAttributes(array(
8         'min' => '0', // default minimum is 0
9         'max' => '100', // default maximum is 100

```

```

10         'step' => '1',    // default interval is 1
11     ));
12
13 $form = new Form('my-form');
14 $form->add($range);

```

Here is with the array notation:

```

1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\Range',
6      'name' => 'range',
7      'options' => array(
8          'label' => 'Minimum and Maximum Amount'
9      ),
10     'attributes' => array(
11         'min' => 0, // default minimum is 0
12         'max' => 100, // default maximum is 100
13         'step' => 1 // default interval is 1
14     )
15 ));

```

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

orphan

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".

```

1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $time = new Element\Time('time');
5 $time
6     ->setLabel('Time')
7     ->setAttributes(array(
8         'min' => '00:00:00',
9         'max' => '23:59:59',
10        'step' => '60', // seconds; default step interval is 60 seconds
11    ))
12    ->setOptions(array(
13        'format' => 'H:i:s'
14    ));
15
16 $form = new Form('my-form');
17 $form->add($time);

```

Here is the same example using the array notation:

```

1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Time',
6     'name' => 'time',
7     'options' => array(
8         'label' => 'Time',
9         'format' => 'H:i:s'
10    ),
11     'attributes' => array(
12         'min' => '00:00:00',
13         'max' => '23:59:59',
14         'step' => '60', // seconds; default step interval is 60 seconds
15    )
16 ));

```

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

orphan

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 its input filter specification for validating HTML5 URL input values on the server.

Basic Usage

This element automatically adds a "type" attribute of value "url".

```
1 use Zend\Form\Element;
2 use Zend\Form\Form;
3
4 $url = new Element\Url('webpage-url');
5 $url->setLabel('Webpage URL');
6
7 $form = new Form('my-form');
8 $form->add($url);
```

Here is the same example using the array notation:

```
1 use Zend\Form\Form;
2
3 $form = new Form('my-form');
4 $form->add(array(
5     'type' => 'Zend\Form\Element\Url',
6     'name' => 'webpage-url',
7     'options' => array(
8         'label' => 'Webpage URL'
9     )
10 ));
```

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

orphan

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".

```

1  use Zend\Form\Element;
2  use Zend\Form\Form;
3
4  $week = new Element\Week('week');
5  $week
6      ->setLabel('Week')
7      ->setAttributes(array(
8          'min' => '2012-W01',
9          'max' => '2020-W01',
10         'step' => '1', // weeks; default step interval is 1 week
11     ));
12
13 $form = new Form('my-form');
14 $form->add($week);

```

Here is the same example using the array notation:

```

1  use Zend\Form\Form;
2
3  $form = new Form('my-form');
4  $form->add(array(
5      'type' => 'Zend\Form\Element\Week',
6      'name' => 'week',
7      'options' => array(
8          'label' => 'Week'
9      ),
10     'attributes' => array(
11         'min' => '2012-W01',
12         'max' => '2020-W01',
13         'step' => '1', // weeks; default step interval is 1 week
14     )
15 ));

```

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 an input filter specification, which includes Zend\Filter\StringTrim and will add the appro-

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

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.

Standard Helpers

orphan

Form

The `Form` view helper is used to render a `<form>` HTML element and its attributes.

It iterates through all its elements and relies on the `FormCollection` and `FormRow` view helpers to render them appropriately.

You can also use `Zend\Form\View\Helper\FormRow` in conjunction with `Form::openTag()` and `Form::closeTag()` to have a more fine grained control over the output.

Basic usage:

```
1  /**
2   * inside view template
3   *
4   * @var \Zend\View\Renderer\PhpRenderer $this
5   * @var \Zend\Form\Form $form
6   */
7
8  echo $this->form($form);
```

```

9 // i.e.
10 // <form action="" method="POST">
11 //     <label>
12 //         <span>Some Label</span>
13 //         <input type="text" name="some_element" value="">
14 //     </label>
15 // </form>

```

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

__invoke (*FormInterface* *\$form* = null)

Prepares and renders the whole form.

Parameters *\$form* – A Form object.

Return type string

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

orphan

FormButton

The FormButton view helper is used to render a <button> HTML element and its attributes.

Basic usage:

```

1 use Zend\Form\Element;
2
3 $element = new Element\Button('my-button');
4 $element->setLabel("Reset");
5
6 // Within your view...
7
8 /**
9  * Example #1: Render entire button in one shot...
10  */
11 echo $this->formButton($element);
12 // <button name="my-button" type="button">Reset</button>
13
14 /**
15  * Example #2: Render button in 3 steps
16  */
17 // Render the opening tag
18 echo $this->formButton()->openTag($element);
19 // <button name="my-button" type="button">
20
21 echo '<span class="inner">' . $element->getLabel() . '</span>';
22
23 // Render the closing tag
24 echo $this->formButton()->closeTag();
25 // </button>

```



```

26
27 /**
28  * Example #3: Override the element label
29  */
30 echo $this->formButton()->render($element, 'My Content');
31 // <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

orphan

FormCaptcha

TODO Basic usage:

```

1 use Zend\Captcha;
2 use Zend\Form\Element;
3
4 $captcha = new Element\Captcha('captcha');
5 $captcha
6     ->setCaptcha(new Captcha\Dumb())
7     ->setLabel('Please verify you are human');
8
9 // Within your view...
10
11 echo $this->formCaptcha($captcha);
12
13 // TODO

```

orphan

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:

```

1  use Zend\Form\Element;
2
3  $element = new Element\Checkbox('my-checkbox');
4
5  // Within your view...
6
7  /**
8   * Example #1: Default options
9   */
10 echo $this->formCheckbox($element);
11 // <input type="hidden" name="my-checkbox" value="0">
12 // <input type="checkbox" name="my-checkbox" value="1">
13
14 /**
15  * Example #2: Disable hidden element
16  */
17 $element->setUseHiddenElement(false);
18 echo $this->formCheckbox($element);
19 // <input type="checkbox" name="my-checkbox" value="1">
20
21 /**
22  * Example #3: Change checked/unchecked values
23  */
24 $element->setUseHiddenElement(true)
25         ->setUncheckedValue('no')
26         ->setCheckedValue('yes');
27 echo $this->formCheckbox($element);
28 // <input type="hidden" name="my-checkbox" value="no">
29 // <input type="checkbox" name="my-checkbox" value="yes">

```

orphan

FormCollection

TODO Basic usage:

TODO

orphan

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:

```

1  use Zend\Form\Form;
2  use Zend\Form\Element;
3
4  // Within your view...
5
6  /**
7   * Example #1: Render different types of form elements

```

```

8  */
9  $textElement = new Element\Text('my-text');
10 $checkboxElement = new Element\Checkbox('my-checkbox');
11
12 echo $this->formElement($textElement);
13 // <input type="text" name="my-text" value="">
14
15 echo $this->formElement($checkboxElement);
16 // <input type="hidden" name="my-checkbox" value="0">
17 // <input type="checkbox" name="my-checkbox" value="1">
18
19 /**
20  * Example #2: Loop through form elements and render them
21  */
22 $form = new Form();
23 // ...add elements and input filter to form...
24 $form->prepare();
25
26 // Render the opening tag
27 echo $this->form()->openTag($form);
28
29 // ...loop through and render the form elements...
30 foreach ($form as $element) {
31     echo $this->formElement($element); // <-- Magic!
32     echo $this->formElementErrors($element);
33 }
34
35 // Render the closing tag
36 echo $this->form()->closeTag();

```

orphan

FormElementErrors

The `FormElementErrors` view helper is used to render the validation error messages of an element.

Basic usage:

```

1  use Zend\Form\Form;
2  use Zend\Form\Element;
3  use Zend\InputFilter\InputFilter;
4  use Zend\InputFilter\Input;
5
6  // Create a form
7  $form = new Form();
8  $element = new Element\Text('my-text');
9  $form->add($element);
10
11 // Create a input
12 $input = new Input('my-text');
13 $input->setRequired(true);
14
15 $inputFilter = new InputFilter();
16 $inputFilter->add($input);
17 $form->setInputFilter($inputFilter);
18
19 // Force a failure

```

```

20 $form->setData(array()); // Empty data
21 $form->isValid();        // Not valid
22
23 // Within your view...
24
25 /**
26  * Example #1: Default options
27  */
28 echo $this->formElementErrors($element);
29 // <ul><li>Value is required and can't be empty</li></ul>
30
31 /**
32  * Example #2: Add attributes to open format
33  */
34 echo $this->formElementErrors($element, array('class' => 'help-inline'));
35 // <ul class="help-inline"><li>Value is required and can't be empty</li></ul>
36
37 /**
38  * Example #3: Custom format
39  */
40 echo $this->formElementErrors()
41     ->setMessageOpenFormat('<div class="help-inline">')
42     ->setMessageSeparatorString('</div><div class="help-inline">')
43     ->setMessageCloseString('</div>')
44     ->render($element);
45 // <div class="help-inline">Value is required and can'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*' 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 '**' 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 '**' 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

orphan

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:

```
1 use Zend\Form\Element;
2
3 $element = new Element\File('my-file');
4
5 // Within your view...
6
7 echo $this->formFile($element);
8 // <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.

```
1 use Zend\Form\Element;
2
3 $element = new Element\File('my-file');
4 $element->setAttribute('multiple', true);
5
6 // Within your view...
7
8 echo $this->formFile($element);
9 // <input type="file" name="my-file" multiple="multiple">
```

orphan

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:

```

1 use Zend\Form\Element;
2
3 $element = new Element\Hidden('my-hidden');
4 $element->setValue('foo');
5
6 // Within your view...
7
8 echo $this->formHidden($element);
9 // <input type="hidden" name="my-hidden" value="foo">

```

orphan

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:

```

1 use Zend\Form\Element;
2
3 $element = new Element\Image('my-image');
4 $element->setAttribute('src', '/img/my-pic.png');
5
6 // Within your view...
7
8 echo $this->formImage($element);
9 // <input type="image" name="my-image" src="/img/my-pic.png">

```

orphan

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 its 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

orphan

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 its rendering.

Basic usage:

```

1 use Zend\Form\Element;
2
3 $element = new Element\Text('my-text');
4 $element->setLabel('Label')
5     ->setAttribute('id', 'text-id')
6     ->setLabelAttributes(array('class' => 'control-label'));
7
8 // Within your view...
9
10 /**
11  * Example #1: Render label in one shot
12  */
13 echo $this->formLabel($element);
14 // <label class="control-label" for="text-id">Label</label>
15
16 echo $this->formLabel($element, $this->formText($element));
17 // <label class="control-label" for="text-id">Label<input type="text" name="my-text">
18     ↪</label>
19
20 echo $this->formLabel($element, $this->formText($element), 'append');
21 // <label class="control-label" for="text-id"><input type="text" name="my-text">Label
22     ↪</label>
23
24 /**
25  * Example #2: Render label in separate steps
26  */
27 // Render the opening tag
28 echo $this->formLabel()->openTag($element);
29 // <label class="control-label" for="text-id">
30
31 // Render the closing tag
32 echo $this->formLabel()->closeTag();
33 // </label>
34
35 /**
36  * Example #3: Render html label after toggling off escape
37  */
38 $element->setLabel('<abbr title="Completely Automated Public Turing test to tell
39     ↪Computers and Humans Apart">CAPTCHA</abbr>');
40 $element->setLabelOptions(array('disable_html_escape' => true));
41 echo $this->formLabel($element);
42 // <label class="control-label" for="text-id">
43 //     <abbr title="Completely Automated Public Turing test to tell Computers and
44     ↪Humans Apart">CAPTCHA</abbr>
45 // </label>

```

Note: HTML escape only applies to the `Element::setLabel` property, not to the helper `$labelContent` parameter.

Attaching a translator and setting a text domain:

```

1 // Setting a translator
2 $this->formLabel()->setTranslator($translator);
3
4 // Setting a text domain

```

```

5 $this->formLabel()->setTranslatorTextDomain('my-text-domain');
6
7 // Setting both
8 $this->formLabel()->setTranslator($translator, 'my-text-domain');
```

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

orphan

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:

```

1 use Zend\Form\Element;
2
3 $element = new Element\MultiCheckbox('my-multicheckbox');
4 $element->setValueOptions(array(
5     '0' => 'Apple',
6     '1' => 'Orange',
7     '2' => 'Lemon',
8 ));
```



```

9
10 // Within your view...
11
12 /**
13  * Example #1: using the default label placement
14  */
15 echo $this->formMultiCheckbox($element);
16 // <label><input type="checkbox" name="my-multicheckbox[]" value="0">Apple</label>
17 // <label><input type="checkbox" name="my-multicheckbox[]" value="1">Orange</label>
18 // <label><input type="checkbox" name="my-multicheckbox[]" value="2">Lemon</label>
19
20 /**
21  * Example #2: using the prepend label placement
22  */
23 echo $this->formMultiCheckbox($element, 'prepend');
24 // <label>Apple<input type="checkbox" name="my-multicheckbox[]" value="0"></label>
25 // <label>Orange<input type="checkbox" name="my-multicheckbox[]" value="1"></label>
26 // <label>Lemon<input type="checkbox" name="my-multicheckbox[]" value="2"></label>

```

orphan

FormPassword

The FormPassword view helper can be used to render a `<input type="password">` HTML form input. It is meant to work with the *Zend\Form\Element\Password* element.

FormPassword extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```

1 use Zend\Form\Element;
2
3 $element = new Element\Password('my-password');
4
5 // Within your view...
6 echo $this->formPassword($element);

```

Output:

```

1 <input type="password" name="my-password" value="">

```

orphan

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:

```

1 use Zend\Form\Element;
2
3 $element = new Element\Radio('gender');
4 $element->setValueOptions(array(
5     '0' => 'Male',
6     '1' => 'Female',
7 ));

```

```

8
9 // Within your view...
10
11 /**
12  * Example #1: using the default label placement
13  */
14 echo $this->formRadio($element);
15 // <label><input type="radio" name="gender[]" value="0">Male</label>
16 // <label><input type="radio" name="gender[]" value="1">Female</label>
17
18 /**
19  * Example #2: using the prepend label placement
20  */
21 echo $this->formRadio($element, 'prepend');
22 // <label>Male<input type="checkbox" name="gender[]" value="0"></label>
23 // <label>Female<input type="checkbox" name="gender[]" value="1"></label>

```

orphan

FormReset

The `FormReset` view helper can be used to render a `<input type="reset">` HTML form input.

`FormText` extends from `Zend\Form\View\Helper\FormInput`. Basic usage:

```

1 use Zend\Form\Element;
2
3 $element = new Element('my-reset');
4 $element->setAttribute('value', 'Reset');
5
6 // Within your view...
7 echo $this->formReset($element);

```

Output:

```

1 <input type="reset" name="my-reset" value="Reset">

```

orphan

FormRow

The `FormRow` view helper is in turn used by `Form` view helper to render each row of a form, nevertheless it can be use stand-alone. A form row usually consists of the output produced by the helper specific to an input, plus its label and errors, if any.

`FormRow` handles different rendering options, having elements wrapped by the `<label>` HTML block by default, but also allowing to render them in separate blocks when the element has an `id` attribute specified, thus preserving browser usability features in any case.

Other options involve label positioning, escaping, toggling errors and using custom partial templates. Please check out `Zend\Form\View\Helper\FormRow` method API for more details. Usage:

```

1 /**
2  * inside view template
3  *
4  * @var \Zend\View\Renderer\PhpRenderer $this

```

```

5  * @var \Zend\Form\Form $form
6  */
7
8  // Prepare the form
9  $form->prepare();
10
11 // Render the opening tag
12 echo $this->form()->openTag($form);
13
14 /** @var \Zend\Form\Element\Text $element */
15 $element = $form->get('some_element');
16 $element->setLabel('Some Label');
17
18 // Render 'some_element' label, input, and errors if any
19 echo $this->formRow($element);
20 // i.e. <label><span>Some Label</span><input type="text" name="some_element" value="">
21 //      </label>
22
23 // Altering label position
24 echo $this->formRow($element, 'append');
25 // i.e. <label><input type="text" name="some_element" value=""><span>Some Label</span>
26 //      </label>
27
28 // Setting the 'id' attribute will result in a separated label rather than a wrapping
29 //      one
30 $element->setAttribute('id', 'element_id');
31 echo $this->formRow($element);
32 // i.e. <label for="element_id">Some Label</label><input type="text" name="some_
33 //      element" id="element_id" value="">
34
35 // Turn off escaping for HTML labels
36 $element->setLabel('<abbr title="Completely Automated Public Turing test to tell
37 //      Computers and Humans Apart">CAPTCHA</abbr>');
38 $element->setLabelOptions(array('disable_html_escape' => true));
39 // i.e.
40 // <label>
41 //     <span>
42 //         <abbr title="Completely Automated Public Turing test to tell Computers and
43 //         Humans Apart">CAPTCHA</abbr>
44 //     </span>
45 //     <input type="text" name="some_element" value="">
46 // </label>
47
48 // Render the closing tag
49 echo $this->form()->closeTag();

```

Note: Label content is escaped by default

orphan

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:

```

1  use Zend\Form\Element;
2
3  $element = new Element\Select('language');
4  $element->setValueOptions(array(
5      '0' => 'French',
6      '1' => 'English',
7      '2' => 'Japanese',
8      '3' => 'Chinese'
9  ));
10
11 // Within your view...
12
13 /**
14  * Example
15  */
16 echo $this->formSelect($element);

```

orphan

FormSubmit

The FormSubmit view helper can be used to render a `<input type="submit">` HTML form input. It is meant to work with the *Zend\Form\Element\Submit* element.

FormSubmit extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```

1  use Zend\Form\Element;
2
3  $element = new Element\Submit('my-submit');
4
5  // Within your view...
6  echo $this->formSubmit($element);

```

Output:

```

1  <input type="submit" name="my-submit" value="">

```

orphan

FormText

The FormText view helper can be used to render a `<input type="text">` HTML form input. It is meant to work with the *Zend\Form\Element\Text* element.

FormText extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```

1  use Zend\Form\Element;
2
3  $element = new Element\Text('my-text');
4
5  // Within your view...
6  echo $this->formText($element);

```

Output:

```
1 <input type="text" name="my-text" value="">
```

orphan

FormTextarea

The `FormTextarea` view helper can be used to render a `<textarea></textarea>` HTML form input. It is meant to work with the *Zend\Form\Element\Textarea* element. Basic usage:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Textarea('my-textarea');
4
5 // Within your view...
6 echo $this->formTextarea($element);
```

Output:

```
1 <textarea name="my-textarea"></textarea>
```

orphan

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

HTML5 Helpers

orphan

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.

`FormColor` extends from *`Zend\Form\View\Helper\FormInput`*. Basic usage:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Color('my-color');
4
5 // Within your view...
6 echo $this->formColor($element);
```

Output:

```
1 <input type="color" name="my-color" value="">
```

orphan

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:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Date('my-date');
4
5 // Within your view...
6
7 echo $this->formDate($element);
8 // <input type="date" name="my-date" value="">
```

orphan

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:

```
1 use Zend\Form\Element;
2
3 $element = new Element\DateTime('my-datetime');
```

```

4 // Within your view...
5
6 echo $this->formDateTime($element);
7 // <input type="datetime" name="my-datetime" value="">
8

```

orphan

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:

```

1 use Zend\Form\Element;
2
3 $element = new Element\DateTimeLocal('my-datetime');
4
5 // Within your view...
6
7 echo $this->formDateTimeLocal($element);
8 // <input type="datetime-local" name="my-datetime" value="">
9

```

orphan

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:

```

1 use Zend\Form\Element;
2
3 $element = new Element\Email('my-email');
4
5 // Within your view...
6
7 echo $this->formEmail($element);
8 // <input type="email" name="my-email" value="">
9

```

orphan

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:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Month('my-month');
4
5 // Within your view...
6
7 echo $this->formMonth($element);
8 // <input type="month" name="my-month" value="">
```

orphan

FormNumber

The FormNumber view helper can be used to render a `<input type="number">` HTML form input. It is meant to work with the *Zend\Form\Element\Number* element, which provides a default input specification for validating numerical values.

FormNumber extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Number('my-number');
4
5 // Within your view...
6 echo $this->formNumber($element);
```

Output:

```
1 <input type="number" name="my-number" value="">
```

Usage of min, max and step attributes:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Number('my-number');
4 $element->setAttributes(
5     array(
6         'min' => 5,
7         'max' => 20,
8         'step' => 0.5,
9     )
10 );
11 $element->setValue(12);
12
13 // Within your view...
14 echo $this->formNumber($element);
```

Output:

```
1 <input type="number" name="my-number" min="5" max="20" step="0.5" value="12">
```

orphan

FormRange

The `FormRange` view helper can be used to render a `<input type="range">` HTML form input. It is meant to work with the `Zend\Form\Element\Range` element, which provides a default input specification for validating numerical values.

`FormRange` extends from `Zend\Form\View\Helper\FormInput`. Basic usage:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Range('my-range');
4
5 // Within your view...
6 echo $this->formRange($element);
```

Output:

```
1 <input type="range" name="my-range" value="">
```

Usage of `min`, `max` and `step` attributes:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Range('my-range');
4 $element->setAttributes(
5     array(
6         'min' => 0,
7         'max' => 100,
8         'step' => 5,
9     )
10 );
11 $element->setValue(20);
12
13 // Within your view...
14 echo $this->formRange($element);
```

Output:

```
1 <input type="range" name="my-range" min="0" max="100" step="5" value="20">
```

orphan

FormSearch

The `FormSearch` view helper can be used to render a `<input type="search">` HTML5 form input.

`FormSearch` extends from `Zend\Form\View\Helper\FormText`. Basic usage:

```
1 use Zend\Form\Element;
2
3 $element = new Element('my-search');
4
5 // Within your view...
6 echo $this->formSearch($element);
```

Output:

```
1 <input type="search" name="my-search" value="">
```

orphan

FormTel

The FormTel view helper can be used to render a `<input type="tel">` HTML5 form input.

FormTel extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
1 use Zend\Form\Element;
2
3 $element = new Element('my-tel');
4
5 // Within your view...
6 echo $this->formTel($element);
```

Output:

```
1 <input type="tel" name="my-tel" value="">
```

orphan

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:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Time('my-time');
4
5 // Within your view...
6
7 echo $this->formTime($element);
8 // <input type="time" name="my-time" value="">
```

orphan

FormUrl

The FormUrl view helper can be used to render a `<input type="url">` HTML form input. It is meant to work with the *Zend\Form\Element\Url* element, which provides a default input specification with an URL validator.

FormUrl extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Url('my-url');
4
5 // Within your view...
6 echo $this->formUrl($element);
```

Output:

```
1 <input type="url" name="my-url" value="">
```

Usage of custom regular expression pattern:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Url('my-url');
4 $element->setAttribute('pattern', 'https://.+');
5
6 // Within your view...
7 echo $this->formUrl($element);
```

Output:

```
1 <input type="url" name="my-url" pattern="https://.+" value="">
```

orphan

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:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Week('my-week');
4
5 // Within your view...
6 echo $this->formWeek($element);
```

Output:

```
1 <input type="week" name="my-week" value="">
```

Usage of min, max and step attributes:

```
1 use Zend\Form\Element;
2
3 $element = new Element\Week('my-week');
4 $element->setAttributes(
5     array(
6         'min' => '2012-W01',
7         'max' => '2020-W01',
8         'step' => 2, // weeks; default step interval is 1 week
9     )
10 );
11 $element->setValue('2014-W10');
12
13 // Within your view...
14 echo $this->formWeek($element);
```

Output:

```
1 <input type="week" name="my-week" min="2012-W01" max="2020-W01" step="2" value="2014-
  ↪ W10">
```

File Upload Progress Helpers

orphan

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:

```
1 // Within your view...
2
3 echo $this->formFileApcProgress();
4 // <input type="hidden" id="progress_key" name="APC_UPLOAD_PROGRESS" value="12345abcde
  ↪ ">
```

orphan

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:

```

1 // Within your view...
2
3 echo $this->formFileSessionProgress();
4 // <input type="hidden" id="progress_key" name="PHP_SESSION_UPLOAD_PROGRESS" value=
   ↪ "12345abcde">

```

orphan

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:

```

1 // Within your view...
2
3 echo $this->formFileSessionProgress();
4 // <input type="hidden" id="progress_key" name="UPLOAD_IDENTIFIER" value="12345abcde">

```


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.

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 145

The Request Class

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](#).

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.

```
1 use Zend\Http\Request;
2
3 $request = Request::fromString(<<<EOS
4 POST /foo HTTP/1.1
5 \r\n
```

```
6 HeaderField1: header-field-value1
7 HeaderField2: header-field-value2
8 \r\n\r\n
9 foo=bar&
10 EOS
11 );
12
13 // OR, the completely equivalent
14
15 $request = new Request();
16 $request->setMethod(Request::METHOD_POST);
17 $request->setUri('/foo');
18 $request->getHeaders()->addHeaders(array(
19     'HeaderField1' => 'header-field-value1',
20     'HeaderField2' => 'header-field-value2',
21 ));
22 $request->getPost()->set('foo', 'bar');
```

Configuration Options

No configuration options are available.

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

Examples

Generating a Request object from a string

```

1 use Zend\Http\Request;
2
3 $string = "GET /foo HTTP/1.1\r\n\r\nSome Content";
4 $request = Request::fromString($string);
5
6 $request->getMethod(); // returns Request::METHOD_GET
7 $request->getUri(); // returns Zend\Uri\Http object
8 $request->getUriString(); // returns '/foo'
9 $request->getVersion(); // returns Request::VERSION_11 or '1.1'
10 $request->getContent(); // returns 'Some Content'

```

Retrieving and setting headers

```
1 use Zend\Http\Request;
2 use Zend\Http\Header\Cookie;
3
4 $request = new Request();
5 $request->getHeaders()->get('Content-Type'); // return content type
6 $request->getHeaders()->addHeader(new Cookie(array('foo' => 'bar')));
7 foreach ($request->getHeaders() as $header) {
8     echo $header->getFieldName() . ' with value ' . $header->getFieldValue();
9 }
```

Retrieving and setting GET and POST values

```
1 use Zend\Http\Request;
2
3 $request = new Request();
4
5 // getPost() and getQuery() both return, by default, a Parameters object, which
6 // extends ArrayObject
7 $request->getPost()->foo = 'Foo value';
8 $request->getQuery()->bar = 'Bar value';
9 $request->getPost('foo'); // returns 'Foo value'
10 $request->getQuery()->offsetGet('bar'); // returns 'Bar value'
```

Generating a formatted HTTP Request from a Request object

```
1 use Zend\Http\Request;
2
3 $request = new Request();
4 $request->setMethod(Request::METHOD_POST);
5 $request->setUri('/foo');
6 $request->getHeaders()->addHeaders(array(
7     'HeaderField1' => 'header-field-value1',
8     'HeaderField2' => 'header-field-value2',
9 ));
10 $request->getPost()->set('foo', 'bar');
11 $request->setContent($request->getPost()->toString());
12 echo $request->toString();
13
14 /** Will produce:
15 POST /foo HTTP/1.1
16 HeaderField1: header-field-value1
17 HeaderField2: header-field-value2
18
19 foo=bar
20 */
```

CHAPTER 146

The Response Class

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](#).

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.

```
1 use Zend\Http\Response;
2 $response = Response::fromString(<<<EOS
3 HTTP/1.0 200 OK
4 HeaderField1: header-field-value
5 HeaderField2: header-field-value2
```

```
6
7 <html>
8 <body>
9     Hello World
10 </body>
11 </html>
12 EOS);
13
14 // OR
15
16 $response = new Response();
17 $response->setStatusCode(Response::STATUS_CODE_200);
18 $response->getHeaders()->addHeaders(array(
19     'HeaderField1' => 'header-field-value',
20     'HeaderField2' => 'header-field-value2',
21 ));
22 $response->setContent(<<<EOS
23 <html>
24 <body>
25     Hello World
26 </body>
27 </html>
28 EOS
29 );
```

Configuration Options

No configuration options are available.

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 raw message content

Returns mixed

getBody `getBody()`

Get decoded message content

Returns mixed

toString toString()

Returns string

Examples

Generating a Response object from a string

```
1 use Zend\Http\Response;
2 $request = Response::fromString(<<<EOS
3 HTTP/1.0 200 OK
4 HeaderField1: header-field-value
5 HeaderField2: header-field-value2
6
7 <html>
8 <body>
9     Hello World
10 </body>
11 </html>
12 EOS);
```

Generating a formatted HTTP Response from a Response object

```
1 use Zend\Http\Response;
2 $response = new Response();
3 $response->setStatusCode(Response::STATUS_CODE_200);
4 $response->getHeaders()->addHeaders(array(
5     'HeaderField1' => 'header-field-value',
6     'HeaderField2' => 'header-field-value2',
7 ));
8 $response->setContent(<<<EOS
9 <html>
10 <body>
11     Hello World
12 </body>
13 </html>
14 EOS);
```

The Headers Class

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.

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.

```
1 // $client is an instance of Zend\Http\Client
2
3 // You can retrieve the request headers by first retrieving
4 // the Request object and then calling getHeaders on it
5 $requestHeaders = $client->getRequest()->getHeaders();
6
7 // The same method also works for retrieving Response headers
8 $responseHeaders = $client->getResponse()->getHeaders();
```

`Zend\Http\Headers` can also extract headers from a string:

```
1 $headerString = <<<EOB
2 Host: www.example.com
3 Content-Type: text/html
4 Content-Length: 1337
```

```

5  EOB;
6
7  $headers = Zend\Http\Headers::fromString($headerString);
8  // $headers is now populated with three objects
9  //   (1) Zend\Http\Header\Host
10 //   (2) Zend\Http\Header\ContentType
11 //   (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.

Configuration Options

No configuration options are available.

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

Zend\Http\Header\HeaderInterface 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"

Returns string

Zend\Http\Header\AbstractAccept Methods

parseHeaderLine `parseHeaderLine(string $headerLine)`

Parse the given header line and add the values

Returns void

getFieldValuePartsFromHeaderLine `getFieldValuePartsFromHeaderLine(string $headerLine)`

Parse the Field Value Parts represented by a header line

Throws `Zend\Http\Header\Exception\InvalidArgumentException` if the header is invalid

Returns array

getFieldValue `getFieldValue(array|null $values = null)`

Get field value

Returns string

match `match(array|string $matchAgainst)`
 Match a media string against this header. Returns the matched value or false
 Returns `Accept\FieldValuePart\AcceptFieldValuePart` or `bool`

getPrioritized `getPrioritized()`
 Returns all the keys, values and parameters this header represents
 Returns array

Zend\Http\Header\AbstractDate Methods

setDateFormat `static setDateFormat(int $format)`
 Set date output format.
 Returns void

getDateFormat `static getDateFormat()`
 Return current date output format
 Returns string

setDate `setDate(string|DateTime $date)`
 Set the date for this header, this can be a string or an instance of `DateTime`
 Throws `Zend\Http\Header\Exception\InvalidArgumentException` if the date is neither a valid string nor an instance of `\DateTime`.
 Returns self

getDate `getDate()`
 Return date for this header
 Returns self

compareTo `compareTo(string|DateTime $date)`
 Compare provided date to date for this header. Returns `< 0` if date in header is less than `$date`; `> 0` if it's greater, and `0` if they are equal. See [strcmp](#).
 Returns int

date `date()`
 Return date for this header as an instance of `\DateTime`
 Returns `\DateTime`

fromTimestamp `fromTimestamp(int $time)`
 Create date-based header from Unix timestamp
 Returns self

fromTimeString `fromTimeString(string $time)`
 Create date-based header from `strtotime()`-compatible string
 Returns self

Zend\Http\Header\AbstractLocation Methods

setUri `setUri(string|Zend\Uri\UriInterface $uri)`

Set the URI/URL for this header, this can be a string or an instance of `Zend\Uri\Http`

Throws `Zend\Http\Header\Exception\InvalidArgumentException` if `$uri` is neither a valid URL nor an instance of `Zend\Uri\UriInterface`.

Returns self

getUri `getUri()`

Return the URI for this header

Returns string

uri `uri()`

Return the URI for this header as an instance of `Zend\Uri\Http`

Returns `Zend\Uri\UriInterface`

List of HTTP Header Types

Some header classes expose methods for manipulating their value. The following list contains all of the classes available in the `Zend\Http\Header*` namespace, as well as any specific methods they contain. All these classes implement `Zend\Http\Header\HeaderInterface` and its *methods*.

Accept See `Zend\Http\Header\AbstractAccept` *methods*.

addMediaType(string \$type, int|float \$priority = 1) Add a media type, with the given priority

Returns self

hasMediaType(string \$type) Does the header have the requested media type?

Returns bool

AcceptCharset See `Zend\Http\Header\AbstractAccept` *methods*.

addCharset(string \$type, int|float \$priority = 1) Add a charset, with the given priority

Returns self

hasCharset(string \$type) Does the header have the requested charset?

Returns bool

AcceptEncoding See `Zend\Http\Header\AbstractAccept` *methods*.

addEncoding(string \$type, int|float \$priority = 1) Add an encoding, with the given priority

Returns self

hasEncoding(string \$type) Does the header have the requested encoding?

Returns bool

AcceptLanguage See `Zend\Http\Header\AbstractAccept` *methods*.

addLanguage(string \$type, int|float \$priority = 1) Add a language, with the given priority

Returns self

hasLanguage(string \$type) Does the header have the requested language?

Returns bool

AcceptRanges `getRangeUnit()`

`setRangeUnit($rangeUnit)`

Age

getDeltaSeconds() Get number of seconds

Returns int

setDeltaSeconds() Set number of seconds

Returns self

Allow

getAllMethods() Get list of all defined methods

Returns array

getAllowedMethods() Get list of allowed methods

Returns array

allowMethods(array|string \$allowedMethods) Allow methods or list of methods

Returns self

disallowMethods(array|string \$allowedMethods) Disallow methods or list of methods

Returns self

denyMethods(array|string \$allowedMethods) Convenience alias for `disallowMethods()`

Returns self

isAllowedMethod(string \$method) Check whether method is allowed

Returns bool

AuthenticationInfo No additional methods

Authorization No additional methods

CacheControl

isEmpty() Checks if the internal directives array is empty

Returns bool

addDirective(string \$key, string|bool \$value) Add a directive

For directives like 'max-age=60', \$value = '60'

For directives like 'private', use the default \$value = true

Returns self

hasDirective(string \$key) Check the internal directives array for a directive

Returns bool

getDirective(string \$key) Fetch the value of a directive from the internal directive array

Returns string|null

removeDirective(string \$key) Remove a directive

Returns self

Connection

setValue(\$value) Set arbitrary header value

RFC allows any token as value, 'close' and 'keep-alive' are commonly used

Returns self

isPersistent() Whether the connection is persistent

Returns bool

setPersistent(bool \$flag) Set Connection header to define persistent connection

Returns self

ContentDisposition No additional methods

ContentEncoding No additional methods

ContentLanguage No additional methods

ContentLength No additional methods

ContentLocation See `Zend\Http\Header\AbstractLocation` [methods](#).

ContentMD5 No additional methods

ContentRange No additional methods

ContentSecurityPolicy

getDirectives() Retrieve the defined directives for the policy

Returns an array

setDirective(string \$name, array \$sources) Set the directive with the given name to include the sources

As an example: an auction site wishes to load images from any URI, plugin content from a list of trusted media providers (including a content distribution network), and scripts only from a server under its control hosting sanitized ECMAScript:

```
// http://www.w3.org/TR/2012/CR-CSP-20121115/#sample-policy-definitions
// Example #2
$csp = new ContentSecurityPolicy();
$csp->setDirective('default-src', array()) // No sources
    ->setDirective('img-src', array('*'))
    ->setDirective('object-src', array('media1.example.com', 'media2.example.
com', '*.cdn.example.com'))
    ->setDirective('script-src', array('trustedscripts.example.com'));
```

Returns self

ContentTransferEncoding No additional methods

ContentType

match(array|string \$matchAgainst) Determine if the mediatype value in this header matches the provided criteria

Returns bool|string

getMediaType() Get the media type

Returns string

setMediaType(string \$mediaType) Set the media type

Returns self

getParameters() Get any additional content-type parameters currently set

Returns array

setParameters(array \$parameters) Set additional content-type parameters

Returns self

getCharset() Get the content-type character set encoding, if any

Returns string|null

setCharset(string \$charset) Set the content-type character set encoding

Returns self

Cookie Extends `ArrayObject`

`static fromSetCookieArray(array $setCookies)`

`setEncodeValue()`

`getEncodeValue()`

Date See `Zend\Http\Header\AbstractDate` [methods](#).

Etag No additional methods

Expect No additional methods

Expires See `Zend\Http\Header\AbstractDate` [methods](#).

From No additional methods

Host No additional methods

IfMatch No additional methods

IfModifiedSince See `Zend\Http\Header\AbstractDate` [methods](#).

IfNoneMatch No additional methods

IfRange No additional methods

IfUnmodifiedSince See `Zend\Http\Header\AbstractDate` [methods](#).

KeepAlive No additional methods

LastModified See `Zend\Http\Header\AbstractDate` [methods](#).

Location See `Zend\Http\Header\AbstractLocation` [methods](#).

MaxForwards No additional methods

Origin No additional methods

Pragma No additional methods

ProxyAuthenticate `toStringMultipleHeaders(array $headers)`

ProxyAuthorization No additional methods

Range No additional methods

Referer See `Zend\Http\Header\AbstractLocation` [methods](#).

Refresh No additional methods

RetryAfter See `Zend\Http\Header\AbstractDate` [methods](#).

setDeltaSeconds(int \$delta) Set number of seconds

Returns self

getDeltaSeconds() Get number of seconds

Returns int

Server No additional methods

SetCookie

getName() / setName(string \$name) The cookie name

getValue() / setValue(string \$value) The cookie value

getExpires() / setExpires(int|string \$expires) The time frame the cookie is valid for, null is a session cookie

getPath() / setPath(string \$path) The URI path the cookie is bound to

getDomain() / setDomain(string \$domain) The domain the cookie applies to

getMaxAge() / setMaxAge(int \$maxAge) The maximum age of the cookie

getVersion() / setVersion(int \$version) The cookie version

isSecure() Whether the cookies contains the Secure flag

Returns bool

setSecure(bool \$secure) Set whether the cookies contains the Secure flag

Returns self

isHttponly() Whether the cookies can be accessed via HTTP only

Returns bool

setHttponly(bool \$httponly) Set whether the cookies can be accessed via HTTP only

Returns self

isExpired() Whether the cookie is expired

Returns bool

isSessionCookie() Whether the cookie is a session cookie

Returns bool

setQuoteFieldValue(bool \$quotedValue) Set whether the value for this cookie should be quoted

Returns self

hasQuoteFieldValue() Check whether the value for this cookie should be quoted

Returns bool

isValidForRequest() Whether the cookie is valid for a given request domain, path and isSecure

Returns bool

match(string \$uri, bool \$matchSessionCookies, int \$now) Checks whether the cookie should be sent or not in a specific scenario

Returns bool

static matchCookieDomain(string \$cookieDomain, string \$host) Check if a cookie's domain matches a host name.

Returns bool

static matchCookiePath(string \$cookiePath, string \$path) Check if a cookie's path matches a URL path

Returns bool

toStringMultipleHeaders(array \$headers) Returns string

TE No additional methods

Trailer No additional methods

TransferEncoding No additional methods

Upgrade No additional methods

UserAgent No additional methods

Vary No additional methods

Via No additional methods

Warning No additional methods

WWWAuthenticate toStringMultipleHeaders(array \$headers)

Examples

Retrieving headers from a Zend\Http\Headers object

```

1 // $client is an instance of Zend\Http\Client
2 $response = $client->send();
3 $headers = $response->getHeaders();
4
5 // We can check if the Request contains a specific header by
6 // using the ``has`` method. Returns boolean ``TRUE`` if at least
7 // one matching header found, and ``FALSE`` otherwise
8 $headers->has('Content-Type');
9
10 // We can retrieve all instances of a specific header by using
11 // the ``get`` method:
12 $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

```
1 $headers = new Zend\Http\Headers();
2
3 // We can directly add any object that implements Zend\Http\Header\HeaderInterface
4 $typeHeader = Zend\Http\Header\ContentType::fromString('Content-Type: text/html');
5 $headers->addHeader($typeHeader);
6
7 // We can add headers using the raw string representation, either
8 // passing the header name and value as separate arguments...
9 $headers->addHeaderLine('Content-Type', 'text/html');
10
11 // .. or we can pass the entire header as the only argument
12 $headers->addHeaderLine('Content-Type: text/html');
13
14 // We can also add headers in bulk using addHeaders, which accepts
15 // an array of individual header definitions that can be in any of
16 // the accepted formats outlined below:
17 $headers->addHeaders(array(
18
19     // An object implementing Zend\Http\Header\HeaderInterface
20     Zend\Http\Header\ContentType::fromString('Content-Type: text/html'),
21
22     // A raw header string
23     'Content-Type: text/html',
24
25     // We can also pass the header name as the array key and the
26     // header content as that array key's value
27     'Content-Type' => 'text/html');
28
29 );
```

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`

```
1 // $headers is a pre-configured instance of Zend\Http\Headers
2
3 // We can also delete individual headers or groups of headers
4 $matches = $headers->get('Content-Type');
5
6 // If more than one header was found, iterate over the collection
7 // and remove each one individually
8 if ($matches instanceof ArrayIterator) {
9     foreach ($headers as $header) {
10         $headers->removeHeader($header);
11     }
12 // If only a single header was found, remove it directly
13 } elseif ($matches instanceof Zend\Http\Header\HeaderInterface) {
14     $headers->removeHeader($header);
15 }
```



```
16
17 // In addition to this, we can clear all the headers currently stored in
18 // the container by calling the clearHeaders() method
19 $matches->clearHeaders();
```


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](#)).

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. The `send()` method is used to submit the request to the remote server, and a `Zend\Http\Response` object is returned:

```
1 use Zend\Http\Client;
2
3 $client = new Client('http://example.org', array(
4     'maxredirects' => 0,
5     'timeout'      => 30
6 ));
7 $response = $client->send();
```

Both constructor parameters can be left out, and set later using the `setUri()` and `setConfig()` methods:

```
1 use Zend\Http\Client;
2
3 $client = new Client();
4 $client->setUri('http://example.org');
5 $client->setOptions(array(
6     'maxredirects' => 0,
7     'timeout'      => 30
```

```
8  ));  
9  $response = $client->send();
```

Zend\Http\Client can also dispatch requests using a separately configured request object (see the [ZendHttpRequest manual page](#) for full details of the methods available):

```
1  use Zend\Http\Client;  
2  use Zend\Http\Request;  
3  
4  $request = new Request();  
5  $request->setUri('http://example.org');  
6  
7  $client = new Client();  
8  
9  $response = $client->send($request);
```

Note: Zend\Http\Client uses Zend\Uri\Http to validate URLs. See the [ZendUri manual page](#) for more information on the validation process.

Configuration

The constructor and setOptions() method accepts an associative array of configuration parameters, or a Zend\Config\Config object. Setting these parameters is optional, as they all have default values.

Table 148.1: 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
user-agent	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
store-response	Whether to store last response for later retrieval with getLastResponse(). If set to FALSE, getLastResponse() will return NULL.	boolean	TRUE
encode-cookies	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
output-stream	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

The options are also passed to the adapter class upon instantiation, so the same array or Zend\Config\Config object) can be used for adapter configuration. See the [Zend Http Client adapter section](#) for more information on the adapter-specific options available.

Examples

Performing a Simple GET Request

Performing simple *HTTP* requests is very easily done:

```

1 use Zend\Http\Client;
2
3 $client = new Client('http://example.org');
4 $response = $client->send();

```

Using Request Methods Other Than GET

The request method can be set using `setMethod()`. 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.

```
1 use Zend\Http\Client;
2
3 $client = new Client('http://example.org');
4
5 // Performing a POST request
6 $client->setMethod('POST');
7 $response = $client->send();
```

For convenience, `Zend\Http\Request` defines all the request methods as class constants, `Zend\Http\Request::METHOD_GET`, `Zend\Http\Request::METHOD_POST` and so on:

```
1 use Zend\Http\Client;
2 use Zend\Http\Request;
3
4 $client = new Client('http://example.org');
5
6 // Performing a POST request
7 $client->setMethod(Request::METHOD_POST);
8 $response = $client->send();
```

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 use Zend\Http\Client;
2 $client = new Client();
3
4 // This is equivalent to setting a URL in the Client's constructor:
5 $client->setUri('http://example.com/index.php?knight=lancelot');
6
7 // Adding several parameters with one call
8 $client->setParameterGet(array(
9     'first_name' => 'Bender',
10    'middle_name' => 'Bending',
11    'last_name'   => 'Rodríguez',
12    'made_in'     => 'Mexico',
13 ));
```

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.

```
1 use Zend\Http\Client;
2
3 $client = new Client();
```

```

4
5 // Setting several POST parameters, one of them with several values
6 $client->setParameterPost(array(
7     'language' => 'es',
8     'country'   => 'ar',
9     'selection' => array(45, 32, 80)
10 ));

```

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.

Connecting to SSL URLs

If you are trying to connect to an SSL (https) URL and are using the default (Zend\Http\Client\Adapter\Socket) adapter, you may need to set the `sslcapath` configuration option in order to allow PHP to validate the SSL certificate:

```

1 use Zend\Http\Client;
2
3 $client = new Client('https://example.org', array(
4     'sslcapath' => '/etc/ssl/certs'
5 ));
6 $response = $client->send();

```

The exact path to use will vary depending on your Operating System. Without this you'll get the exception “Unable to enable crypto on TCP connection” when trying to connect.

Alternatively, you could switch to the curl adapter, which negotiates SSL connections more transparently:

```

1 use Zend\Http\Client;
2
3 $client = new Client('https://example.org', array(
4     'adapter' => 'Zend\Http\Client\Adapter\Curl'
5 ));
6 $response = $client->send();

```

A Complete Example

```

1 use Zend\Http\Client;
2
3 $client = new Client();
4 $client->setUri('http://www.example.com');
5 $client->setMethod('POST');
6 $client->setParameterPost(array(
7     'foo' => 'bar'
8 ));
9
10 $response = $client->send();
11
12 if ($response->isSuccess()) {
13     // the POST was successful
14 }

```

or the same thing, using a request object:

```
1  use Zend\Http\Client;
2  use Zend\Http\Request;
3
4  $request = new Request();
5  $request->setUri('http://www.example.com');
6  $request->setMethod('POST');
7  $request->getPost()->set('foo', 'bar');
8
9  $client = new Client();
10 $response = $client->send($request);
11
12 if ($response->isSuccess()) {
13     // the POST was successful
14 }
```

HTTP Client - Connection Adapters

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.

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 149.1: Zend\Http\Client\Adapter\Socket configuration parameters

Parameter	Description	Expected Type	Default Value
<code>persistent</code>	Whether to use persistent TCP connections	boolean	FALSE
<code>ssltransport</code>	SSL transport layer (eg. 'ssl', 'tls')	string	ssl
<code>sslcert</code>	Path to a PEM encoded SSL certificate	string	NULL
<code>sslpassphrase</code>	Passphrase for the SSL certificate file	string	NULL
<code>sslverifypeer</code>	Whether to verify the SSL peer	string	TRUE
<code>sslcapath</code>	Path to SSL certificate directory	string	NULL
<code>sslallow-selfsigned</code>	Whether to allow self-signed certificates	string	FALSE
<code>sslusecontext</code>	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

```

1 // Set the configuration parameters
2 $config = array(
3     'adapter'      => 'Zend\Http\Client\Adapter\Socket',
4     'ssltransport' => 'tls'
5 );
6
7 // Instantiate a client object
8 $client = new Zend\Http\Client('https://www.example.com', $config);
9
10 // The following request will be sent over a TLS secure connection.
11 $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)
```

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

```

1  // Array of options
2  $options = array(
3      'socket' => array(
4          // Bind local socket side to a specific interface
5          'bindto' => '10.1.2.3:50505'
6      ),
7      'ssl' => array(
8          // Verify server side certificate,
9          // do not accept invalid or self-signed SSL certificates
10         'verify_peer' => true,
11         'allow_self_signed' => false,
12
13         // Capture the peer's certificate
14         'capture_peer_cert' => true
15     )
16 );
17
18 // Create an adapter object and attach it to the HTTP client
19 $adapter = new Zend\Http\Client\Adapter\Socket();
20 $client = new Zend\Http\Client();
21 $client->setAdapter($adapter);
22
23 // Method 1: pass the options array to setStreamContext()
24 $adapter->setStreamContext($options);
25
26 // Method 2: create a stream context and pass it to setStreamContext()
27 $context = stream_context_create($options);
28 $adapter->setStreamContext($context);
29
30 // Method 3: get the default stream context and set the options on it
31 $context = $adapter->getStreamContext();
32 stream_context_set_option($context, $options);
33

```

```

34 // Now, perform the request
35 $response = $client->send();
36
37 // If everything went well, you can now access the context again
38 $opts = stream_context_get_options($adapter->getStreamContext());
39 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.

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:

Table 149.2: `Zend\Http\Client` configuration parameters

Parameter	Description	Expected Type	Example Value
<code>proxy_host</code>	Proxy server address	string	‘proxy.myhost.com’ or ‘10.1.2.3’
<code>proxy_port</code>	Proxy server TCP port	integer	8080 (default) or 81
<code>proxy_user</code>	Proxy user name, if required	string	‘shahar’ or ‘’ for none (default)
<code>proxy_pass</code>	Proxy password, if required	string	‘secret’ or ‘’ for none (default)
<code>proxy_auth</code>	Proxy HTTP authentication type	string	<code>Zend\Http\Client::AUTH_BASIC</code> (default)

`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

```

1 // Set the configuration parameters
2 $config = array(
3     'adapter' => 'Zend\Http\Client\Adapter\Proxy',
4     'proxy_host' => 'proxy.int.zend.com',
5     'proxy_port' => 8000,
6     'proxy_user' => 'shahar.e',

```

```

7     'proxy_pass' => 'bananashaped'
8 );
9
10 // Instantiate a client object
11 $client = new Zend\Http\Client('http://www.example.com', $config);
12
13 // 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.

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

```

1 $config = array(
2     'adapter' => 'Zend\Http\Client\Adapter\Curl',
3     'curloptions' => array(CURLOPT_FOLLOWLOCATION => true),
4 );
5 $client = new Zend\Http\Client($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()`;

Transferring Files by Handle

You can use cURL to transfer very large files over *HTTP* by filehandle.

```

1 $putFileSize = filesize("filepath");
2 $putFileHandle = fopen("filepath", "r");
3
4 $adapter = new Zend\Http\Client\Adapter\Curl();
5 $client = new Zend\Http\Client();
6 $client->setAdapter($adapter);
7 $client->setMethod('PUT');
8 $adapter->setOptions(array(
9     'curloptions' => array(
10         CURLOPT_INFILE => $putFileHandle,

```

```
11         CURLOPT_INFILESIZE => $putFileSize
12     )
13 );
14 $client->send();
```

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

```
1  // Instantiate a new adapter and client
2  $adapter = new Zend\Http\Client\Adapter\Test();
3  $client = new Zend\Http\Client('http://www.example.com', array(
4      'adapter' => $adapter
5  ));
6
7  // Set the expected response
8  $adapter->setResponse(
9      "HTTP/1.1 200 OK" . "\r\n" .
10     "Content-type: text/xml" . "\r\n" .
11     "\r\n" .
12     '<?xml version="1.0" encoding="UTF-8"?>' .
13     '<rss version="2.0" ' .
14     '    xmlns:content="http://purl.org/rss/1.0/modules/content/" ' .
15     '    xmlns:wfw="http://wellformedweb.org/CommentAPI/" ' .
16     '    xmlns:dc="http://purl.org/dc/elements/1.1/">' .
17     '    <channel>' .
18     '        <title>Premature Optimization</title>' .
19     // and so on...
20     '</rss>');
21
22 $response = $client->send();
23 // .. 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

```

1 // Instantiate a new adapter and client
2 $adapter = new Zend\Http\Client\Adapter\Test();
3 $client = new Zend\Http\Client('http://www.example.com', array(
4     'adapter' => $adapter
5 ));
6
7 // Set the first expected response
8 $adapter->setResponse(
9     "HTTP/1.1 302 Found" . "\r\n" .
10    "Location: /" . "\r\n" .
11    "Content-Type: text/html" . "\r\n" .
12    "\r\n" .
13    '<html>' .
14    ' <head><title>Moved</title></head>' .
15    ' <body><p>This page has moved.</p></body>' .
16    '</html>');
17
18 // Set the next successive response
19 $adapter->addResponse(
20    "HTTP/1.1 200 OK" . "\r\n" .
21    "Content-Type: text/html" . "\r\n" .
22    "\r\n" .
23    '<html>' .
24    ' <head><title>My Pet Store Home Page</title></head>' .
25    ' <body><p>...</p></body>' .
26    '</html>');
27
28 // inject the http client object ($client) into your object
29 // 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

```

1 // Instantiate a new adapter and client
2 $adapter = new Zend\Http\Client\Adapter\Test();
3 $client = new Zend\Http\Client('http://www.example.com', array(
4     'adapter' => $adapter

```

```
5    ));
6
7    // Force the next request to fail with an exception
8    $adapter->setNextRequestWillFail(true);
9
10   try {
11       // This call will result in a Zend\Http\Client\Adapter\Exception\RuntimeException
12       $client->send();
13   } catch (Zend\Http\Client\Adapter\Exception\RuntimeException $e) {
14       // ...
15   }
16
17   // Further requests will work as expected until
18   // you call setNextRequestWillFail(true) again
```

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

```
1  class MyApp\Http\Client\Adapter\BananaProtocol
2      implements Zend\Http\Client\Adapter\AdapterInterface
3  {
4      /**
5       * Set Adapter Options
6       *
7       * @param array $config
8       */
9      public function setOptions($config = array())
10     {
11         // This rarely changes - you should usually copy the
12         // implementation in Zend\Http\Client\Adapter\Socket.
13     }
14
15     /**
16      * Connect to the remote server
17      *
18      * @param string $host
19      * @param int $port
20      * @param boolean $secure
21      */
22     public function connect($host, $port = 80, $secure = false)
23     {
24         // Set up the connection to the remote server
25     }
26 }
```



```

27  /**
28   * Send request to the remote server
29   *
30   * @param string      $method
31   * @param Zend\Uri\Http $url
32   * @param string      $http_ver
33   * @param array       $headers
34   * @param string      $body
35   * @return string Request as text
36   */
37  public function write($method,
38                      $url,
39                      $http_ver = '1.1',
40                      $headers = array(),
41                      $body = '')
42  {
43      // Send request to the remote server.
44      // This function is expected to return the full request
45      // (headers and body) as a string
46  }
47
48  /**
49   * Read response from server
50   *
51   * @return string
52   */
53  public function read()
54  {
55      // Read response from remote server and return it as a string
56  }
57
58  /**
59   * Close the connection to the server
60   *
61   */
62  public function close()
63  {
64      // Close the connection to the remote server - called last.
65  }
66 }
67
68 // Then, you could use this adapter:
69 $client = new Zend\Http\Client(array(
70     'adapter' => 'MyApp\Http\Client\Adapter\BananaProtocol'
71 ));

```

HTTP Client - Advanced Usage

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 do 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

```
1 // Strict Redirections
2 $client->setOptions(array('strictredirects' => true));
3
4 // Non-strict Redirections
5 $client->setOptions(array('strictredirects' => false));
```

You can always get the number of redirections done after sending a request using the `getRedirectionsCount()` method.

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()

```
1 // Easy and simple: by providing a cookie name and cookie value
2 $client->addCookie('flavor', 'chocolate chips');
3
4 // By providing a Zend\Http\Header\SetCookie object
5 $cookie = Zend\Http\Header\SetCookie::fromString('Set-Cookie: flavor=chocolate
6 ↪%20chips');
7 $client->addCookie($cookie);
8
9 // Multiple cookies can be set at once by providing an
10 // array of Zend\Http\Header\SetCookie objects
11 $cookies = array(
12     Zend\Http\Header\SetCookie::fromString('Set-Cookie: flavorOne=chocolate%20chips
13 ↪'),
14     Zend\Http\Header\SetCookie::fromString('Set-Cookie: flavorTwo=vanilla'),
15 );
16 $client->addCookie($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()

```
1 // setCookies accepts an array of cookie values as $name => $value
2 $client->setCookies(array(
3     'flavor' => 'chocolate chips',
4     'amount' => 10,
5 ));
```

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\Cookies`. This is useful, for example when you need to log in to a remote site first and receive an authentication or session ID cookie before sending further requests.

Enabling Cookie Stickiness

```
1 $headers = $client->getRequest()->getHeaders();
2 $cookies = new Zend\Http\Cookies($headers);
3
4 // First request: log in and start a session
5 $client->setUri('http://example.com/login.php');
6 $client->setParameterPost(array('user' => 'h4x0r', 'password' => '133t'));
7 $client->setMethod('POST');
8
9 $response = $client->getResponse();
10 $cookies->addCookiesFromResponse($response, $client->getUri());
11
12 // Now we can send our next request
13 $client->setUri('http://example.com/read_member_news.php');
14 $client->setCookies($cookies->getMatchingCookies($client->getUri()));
15 $client->setMethod('GET');
```

For more information about the `Zend\Http\Cookies` class, see [this section](#).

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

```

1 // Fetch the container
2 $headers = $client->getRequest()->getHeaders();
3
4 // Setting a single header. Will not overwrite any
5 // previously-added headers of the same name.
6 $headers->addHeaderLine('Host', 'www.example.com');
7
8 // Another way of doing the exact same thing
9 $headers->addHeaderLine('Host: www.example.com');
10
11 // Another way of doing the exact same thing using
12 // the provided Zend\Http\Header class
13 $headers->addHeader(Zend\Http\Header\Host::fromString('Host: www.example.com'));
14
15 // You can also add multiple headers at once by passing an
16 // array to addHeaders using any of the formats below:
17 $headers->addHeaders(array(
18     // Zend\Http\Header\* object
19     Zend\Http\Header\Host::fromString('Host: www.example.com'),
20
21     // Header name as array key, header value as array key value
22     'Cookie' => 'PHPSESSID=1234567890abcdef1234567890abcdef',
23
24     // Raw header string
25     'Cookie: language=he',
26 ));

```

`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

```

1 // Setting multiple headers. Will remove all existing
2 // headers and add new ones to the Request header container
3 $client->setHeaders(array(
4     Zend\Http\Header\Host::fromString('Host: www.example.com'),
5     'Accept-Encoding' => 'gzip,deflate',
6     'X-Powered-By: Zend Framework',
7 ));

```

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

```
1 // Uploading arbitrary data as a file
2 $text = 'this is some plain text';
3 $client->setFileUpload('some_text.txt', 'upload', $text, 'text/plain');
4
5 // Uploading an existing file
6 $client->setFileUpload('/tmp/Backup.tar.gz', 'bufile');
7
8 // Send the files
9 $client->setMethod('POST');
10 $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.

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

```
1 $xml = '<book>' .
2       ' <title>Islands in the Stream</title>' .
3       ' <author>Ernest Hemingway</author>' .
4       ' <year>1970</year>' .
5       '</book>';
```

```
6 $client->setMethod('POST');
7 $client->setRawBody($xml);
8 $client->setEncType('text/xml');
9 $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.

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

```
1 // Using basic authentication
2 $client->setAuth('shahar', 'myPassword!', Zend\Http\Client::AUTH_BASIC);
3
4 // Since basic auth is default, you can just do this:
5 $client->setAuth('shahar', 'myPassword!');
6
7 // You can also specify username and password in the URI
8 $client->setUri('http://christer:secret@example.com');
```

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\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

```
1 // First, instantiate the client
2 $client = new Zend\Http\Client('http://www.example.com/fetchdata.php', array(
3     'keepalive' => true
4 ));
5
6 // Do we have the cookies stored in our session?
7 if (isset($_SESSION['cookiejar']) &&
8     $_SESSION['cookiejar'] instanceof Zend\Http\Cookies) {
9
10     $cookieJar = $_SESSION['cookiejar'];
11 } else {
12     // If we don't, authenticate and store cookies
13     $client->setUri('http://www.example.com/login.php');
14     $client->setParameterPost(array(
15         'user' => 'shahar',
16         'pass' => 'somesecret'
17     ));
18     $response = $client->setMethod('POST')->send();
19     $cookieJar = Zend\Http\Cookies::fromResponse($response);
20
21     // Now, clear parameters and set the URI to the original one
22     // (note that the cookies that were set by the server are now
23     // stored in the jar)
24     $client->resetParameters();
25     $client->setUri('http://www.example.com/fetchdata.php');
26 }
27
28 // Add the cookies to the new request
29 $client->setCookies($cookieJar->getMatchingCookies($client->getUri()));
30 $response = $client->setMethod('GET')->send();
31
32 // Store cookies in session, for next page
33 $_SESSION['cookiejar'] = $cookieJar;
```

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

```
1  $client->setStream(); // will use temp file
2  $response = $client->send();
3  // copy file
4  copy($response->getStreamName(), "my/downloads/file");
5  // use stream
6  $fp = fopen("my/downloads/file2", "w");
7  stream_copy_to_stream($response->getStream(), $fp);
8  // Also can write to known file
9  $client->setStream("my/downloads/myfile")->send();
```

HTTP Client - Static Usage

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:

Quick Start

```
1 use Zend\Http\ClientStatic;
2
3 // Simple GET request
4 $response = ClientStatic::get('http://example.org');
5
6 // More complex GET request, specifying query string 'foo=bar' and adding a
7 // custom header to request JSON data be returned (Accept: application/json)
8 $response = ClientStatic::get(
9     'http://example.org',
10     array('foo' => 'bar'),
11     array('Accept' => 'application/json')
12 );
13
14 // We can also do a POST request using the same format. Here we POST
15 // login credentials (username/password) to a login page:
16 $response = ClientStatic::post('https://example.org/login.php', array(
17     'username' => 'foo',
18     'password' => 'bar',
19 ));
```

Available Methods

get `get(string $url, array $query = array(), array $headers = array(), mixed $body = null, $clientOptions = null)`

Perform an HTTP GET request using the provided URL, query string variables, headers and request body. The fifth parameter can be used to pass configuration options to the HTTP Client instance.

Returns `Zend\Http\Response`

post `post(string $url, array $params, array $headers = array(), mixed $body = null, $clientOptions = null)`

Perform an HTTP POST request using the provided URL, parameters, headers and request body. The fifth parameter can be used to pass configuration options to the HTTP Client instance.

Returns `Zend\Http\Response`

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.

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:

```
1 use Zend\I18n\Translator\Translator;
2
3 $translator = new Translator();
4 $translator->addTranslationFile($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:

```
1 use Zend\I18n\Translator\Translator;
2
3 $translator = new Translator();
4 $translator->addTranslationFilePattern($type, $baseDir, $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.

Supported formats

The translator supports the following major translation formats:

- PHP arrays
- Gettext
- INI

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.

Translating messages

Translating messages can be accomplished by calling the `translate()` method of the translator:

```
1 $translator->translate($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:

```
1 $translator->translatePlural($singular, $plural, $number, $textDomain, $locale);
```

Plural translations are only available if the underlying format supports the transport of plural messages and plural rule definitions.

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.

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.

orphan

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

```
1 // Within your view
2
3 echo $this->currencyFormat(1234.56, 'USD', null, 'en_US');
4 // This returns: "$1,234.56"
5
6 echo $this->currencyFormat(1234.56, 'EUR', null, 'de_DE');
7 // This returns: "1.234,56 €"
8
9 echo $this->currencyFormat(1234.56, 'USD', true, 'en_US');
10 // This returns: "$1,234.56"
11
12 echo $this->currencyFormat(1234.56, 'USD', false, 'en_US');
13 // This returns: "$1,235"
```

```

14 echo $this->currencyFormat(12345678.90, 'EUR', true, 'de_DE', '#0.# kg');
15 // This returns: "12345678,90 kg"
16
17 echo $this->currencyFormat(12345678.90, 'EUR', false, 'de_DE', '#0.# kg');
18 // This returns: "12345679 kg"
19

```

currencyFormat (float \$number[, string \$currencyCode = null[, bool \$showDecimals = null[, string \$locale = null[, string \$pattern = null]]]])

Format a number

Parameters

- **\$number** – The numeric currency value.
- **\$currencyCode** – (Optional) The 3-letter ISO 4217 currency code indicating the currency to use. If unset, it will use the default value `null (getCurrencyCode())`.
- **\$showDecimals** – (Optional) Boolean false as third argument shows no decimals. If unset, it will use the default value `true (shouldShowDecimals())`.
- **\$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()`).
- **\$pattern** – (Optional) Pattern string that is used by the formatter. If unset, it will use the default value `null (getCurrencyPattern())`.

Return type string

Available Methods

Set the currency code and the locale

The `$currencyCode` and `$locale` options can be set prior to formatting and will be applied each time the helper is used:

```

1 // Within your view
2
3 $this->plugin('currencyformat')->setCurrencyCode('USD')->setLocale('en_US');
4
5 echo $this->currencyFormat(1234.56);
6 // This returns: "$1,234.56"
7
8 echo $this->currencyFormat(5678.90);
9 // This returns: "$5,678.90"

```

setCurrencyCode (string \$currencyCode)

The 3-letter ISO 4217 currency code indicating the currency to use

Parameters **\$currencyCode** – The 3-letter ISO 4217 currency code.

Return type `Zend\I18n\View\Helper\CurrencyFormat`

setLocale (string \$locale)

Set locale to use instead of the default

Parameters **\$locale** – Locale in which the number would be formatted.

Return type `Zend\I18n\View\Helper\CurrencyFormat`

Show decimals

```

1 // Within your view
2
3 $this->plugin('currencyformat')->setShouldShowDecimals(false);
4
5 echo $this->currencyFormat(1234.56, 'USD', null, 'en_US');
6 // This returns: "$1,235"

```

setShouldShowDecimals (*bool \$showDecimals*)

Set if the view helper should show two decimals

Parameters **\$showDecimals** – Whether or not to show the decimals.

Return type Zend\I18n\View\Helper\CurrencyFormat

Set currency pattern

```

1 // Within your view
2
3 $this->plugin('currencyformat')->setCurrencyPattern('#0.##kg');
4
5 echo $this->currencyFormat(12345678.90, 'EUR', null, 'de_DE');
6 // This returns: "12345678,90 kg"

```

setCurrencyPattern (*string \$currencyPattern*)

Set the currency pattern used by the formatter. (See the [NumberFormatter::setPattern](#) *PHP* method for more information.)

Parameters **\$currencyPattern** – Pattern in syntax described in [ICU DecimalFormat documentation](#)

Return type Zend\I18n\View\Helper\CurrencyFormat

orphan

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

```

1 // Within your view
2
3 // Date and Time
4 echo $this->dateFormat(
5     new DateTime(),
6     IntlDateFormatter::MEDIUM, // date
7     IntlDateFormatter::MEDIUM, // time
8     "en_US"
9 );
10 // This returns: "Jul 2, 2012 6:44:03 PM"

```

```
11 // Date Only
12 echo $this->dateFormat(
13     new DateTime(),
14     IntlDateFormatter::LONG, // date
15     IntlDateFormatter::NONE, // time
16     "en_US"
17 );
18 // This returns: "July 2, 2012"
19
20 // Time Only
21 echo $this->dateFormat(
22     new DateTime(),
23     IntlDateFormatter::NONE, // date
24     IntlDateFormatter::SHORT, // time
25     "en_US"
26 );
27 // This returns: "6:44 PM"
28
```

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.

```
1 // Within your view
2 $this->plugin("dateFormat")->setTimezone("America/New_York")->setLocale("en_US");
3
4 echo $this->dateFormat(new DateTime(), IntlDateFormatter::MEDIUM); // "Jul 2, 2012"
5 echo $this->dateFormat(new DateTime(), IntlDateFormatter::SHORT); // "7/2/12"
```

orphan

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

```

1 // Within your view
2
3 // Example of Decimal formatting:
4 echo $this->numberFormat (
5     1234567.891234567890000,
6     NumberFormatter::DECIMAL,
7     NumberFormatter::TYPE_DEFAULT,
8     "de_DE"
9 );
10 // This returns: "1.234.567,891"
11
12 // Example of Percent formatting:
13 echo $this->numberFormat (
14     0.80,
15     NumberFormatter::PERCENT,
16     NumberFormatter::TYPE_DEFAULT,
17     "en_US"
18 );
19 // This returns: "80%"
20
21 // Example of Scientific notation formatting:
22 echo $this->numberFormat (
23     0.00123456789,
24     NumberFormatter::SCIENTIFIC,
25     NumberFormatter::TYPE_DEFAULT,
26     "fr_FR"
27 );
28 // 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.

```

1 // Within your view
2 $this->plugin("numberformat")
3     ->setFormatStyle (NumberFormatter::PERCENT)
4     ->setFormatType (NumberFormatter::TYPE_DOUBLE)
5     ->setLocale ("en_US");
6

```

```
7 echo $this->numberFormat(0.56); // "56%"
8 echo $this->numberFormat(0.90); // "90%"
```

orphan

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:

```
1 // Get the ViewHelperPlugin Manager from Service manager, so we can fetch the
  ↳ ``Plural``
2 // helper and add the plural rule for the application's language
3 $viewHelperManager = $serviceManager->get('ViewHelperManager');
4 $pluralHelper      = $viewHelperManager->get('Plural');
5
6 // Here is the rule for French
7 $pluralHelper->setPluralRule('nplurals=2; plural=(n==0 || n==1 ? 0 : 1)');
```

The string reads like that:

1. First, we specify how many plurals forms 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:

```
1 // Here is the rule for English
2 $pluralHelper->setPluralRule('nplurals=2; plural=(n==1 ? 0 : 1)');
```

Now that we have defined the rule, we can use it in our views:

```

1 <?php
2     // If the rule defined in Module.php is the English one:
3
4     echo $this->plural(array('car', 'cars'), 0); // prints "cars"
5     echo $this->plural(array('car', 'cars'), 1); // prints "car"
6
7     // If the rule defined in Module.php is the French one:
8     echo $this->plural(array('voiture', 'voitures'), 0); // prints "voiture"
9     echo $this->plural(array('voiture', 'voitures'), 1); // prints "voiture"
10    echo $this->plural(array('voiture', 'voitures'), 2); // prints "voitures"
11 ?>

```

orphan

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

```

1 // Within your view
2
3 echo $this->translate("Some translated text.");
4
5 echo $this->translate("Translated text from a custom text domain.", "customDomain");
6
7 echo sprintf($this->translate("The current time is %s."), $currentTime);
8
9 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*.

orphan

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

```
1 // Within your view
2 echo $this->translatePlural("car", "cars", $num);
3
4 // Use a custom domain
5 echo $this->translatePlural("monitor", "monitors", $num, "customDomain");
6
7 // Change locale
8 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*.

orphan

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`

Zend Framework comes with a set of filters related to Internationalization.

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

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()`

Basic Usage

```
1 // Default settings, deny whitespace
2 $filter = new \Zend\I18n\Filter\Alnum();
3 echo $filter->filter("This is (my) content: 123");
4 // Returns "Thisismycontent123"
5
```

```
6 // First param in constructor is $allowWhiteSpace
7 $filter = new \Zend\I18n\Filter\Alnum(true);
8 echo $filter->filter("This is (my) content: 123");
9 // 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.

Alpha

The Alpha filter can be used to return only alphabetic characters in the unicode “letter” category. All other characters are suppressed.

Supported Options

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()`

Basic Usage

```
1 // Default settings, deny whitespace
2 $filter = new \Zend\I18n\Filter\Alpha();
3 echo $filter->filter("This is (my) content: 123");
4 // Returns "Thisismycontent"
5
6 // Allow whitespace
7 $filter = new \Zend\I18n\Filter\Alpha(true);
8 echo $filter->filter("This is (my) content: 123");
9 // 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.

NumberFormat

The `NumberFormat` filter can be used to return locale-specific number and percentage strings. It extends the `NumberParse` filter, which acts as wrapper for the `NumberFormatter` class within the Internationalization extension (Intl).

Supported Options

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()`

Basic Usage

```
1 $filter = new \Zend\I18n\Filter\NumberFormat("de_DE");
2 echo $filter->filter(1234567.8912346);
3 // Returns "1.234.567,891"
4
5 $filter = new \Zend\I18n\Filter\NumberFormat("en_US", NumberFormatter::PERCENT);
6 echo $filter->filter(0.80);
7 // Returns "80%"
8
9 $filter = new \Zend\I18n\Filter\NumberFormat("fr_FR", NumberFormatter::SCIENTIFIC);
10 echo $filter->filter(0.00123456789);
11 // Returns "1,23456789E-3"
```

NumberParse

The `NumberParse` filter can be used to parse a number from a string. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl).

Supported Options

The following options are supported for `NumberParse`:

```
NumberParse([ string $locale [, int $style [, int $type ]]])
```

- `$locale`: (Optional) Locale in which the number would be parsed (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 parsing, one of the [format style constants](#). If unset, it will use `NumberFormatter::DEFAULT_STYLE` as the default style.

Methods for getting/setting the parse style are also available: `getStyle()` and `setStyle()`

- `$type`: (Optional) The [parsing type](#) to use. If unset, it will use `NumberFormatter::TYPE_DOUBLE` as the default type.

Methods for getting/setting the parse type are also available: `getType()` and `setType()`

Basic Usage

```
1 $filter = new \Zend\I18n\Filter\NumberParse("de_DE");
2 echo $filter->filter("1.234.567,891");
3 // Returns 1234567.8912346
4
5 $filter = new \Zend\I18n\Filter\NumberParse("en_US", NumberFormatter::PERCENT);
6 echo $filter->filter("80%");
7 // Returns 0.80
8
9 $filter = new \Zend\I18n\Filter\NumberParse("fr_FR", NumberFormatter::SCIENTIFIC);
10 echo $filter->filter("1,23456789E-3");
11 // Returns 0.00123456789
```

CHAPTER 155

I18n Validators

Zend Framework comes with a set of validators related to Internationalization.

`Zend\I18n\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.

Supported Options

The following options are supported for `Zend\I18n\Validator\Alnum`:

- **`allowWhiteSpace`**: If whitespace characters are allowed. This option defaults to `FALSE`

Basic usage

A basic example is the following one:

```
1 $validator = new Zend\I18n\Validator\Alnum();
2 if ($validator->isValid('Abcd12')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

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()`.

```
1 $validator = new Zend\I18n\Validator\Alnum(array('allowWhiteSpace' => true));
2 if ($validator->isValid('Abcd and 12')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

Using different languages

There are actually 3 languages which are not accepted in their own script. These languages are **korean**, **japanese** and **chinese** because these languages are using an alphabet where a single character is built by using multiple characters.

In the case you are using these languages, the input will only be validated by using the english alphabet.

Alpha Validator

Zend\I18n\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\I18n\Validator\Alnum validator with the exception that it does not accept digits. The following options are supported for Zend\I18n\Validator\Alpha:

- **allowWhiteSpace**: If whitespace characters are allowed. This option defaults to FALSE

A basic example is the following one:

```
1 $validator = new Zend\I18n\Validator\Alpha();
2 if ($validator->isValid('Abcd')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

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()`.

```
1 $validator = new Zend\I18n\Validator\Alpha(array('allowWhiteSpace' => true));
2 if ($validator->isValid('Abcd and efg')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

When using Zend\I18n\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 these languages are using an alphabet where a single character is built by using multiple characters.

In the case you are using these languages, the input will only be validated by using the english alphabet.

`Zend\I18n\Validator\IsFloat` allows you to validate if a given value contains a floating-point value. This validator validates also localized input. The following options are supported for `Zend\I18n\Validator\IsFloat`:

- **locale**: Sets the locale which will be used to validate localized float values.

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:

```
1 $validator = new Zend\I18n\Validator\IsFloat();
2
3 $validator->isValid(1234.5); // returns true
4 $validator->isValid('10a01'); // returns false
5 $validator->isValid('1,234.5'); // returns true
```

In the above example we expected that our environment is set to “en” as locale. 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\IsFloat` is able to validate such notations. However, it is limited to the locale you set. See the following code:

```
1 $validator = new Zend\I18n\Validator\IsFloat(array('locale' => 'de'));
2
3 $validator->isValid(1234.5); // returns true
4 $validator->isValid("1 234,5"); // returns false
5 $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()`.

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `float` is a reserved keyword, which required renaming the `Float` validator. If you were using the `Float` validator directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `IsFloat` class instead.

Users pulling their `Float` validator instance from the validator plugin manager receive an `IsFloat` instance instead starting in 2.4.0.

IsInt

`Zend\I18n\Validator\IsInt` validates if a given value is an integer. Also localized integer values are recognised and can be validated.

Supported Options

The following options are supported for `Zend\I18n\Validator\IsInt`:

- **locale**: Sets the locale which will be used to validate localized integers.

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:

```
1 $validator = new Zend\I18n\Validator\IsInt();
2
3 $validator->isValid(1234); // returns true
4 $validator->isValid(1234.5); // returns false
5 $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.

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\IsInt` 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:

```
1 $validator = new Zend\I18n\Validator\IsInt(array('locale' => 'de'));
2
3 $validator->isValid(1234); // returns true
4 $validator->isValid("1,234"); // returns false
5 $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()`.

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `int` is a reserved keyword, which required renaming the `Int` validator. If you were using the `Int` validator directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `IsInt` class instead.

Users pulling their `Int` validator instance from the validator plugin manager receive an `IsInt` instance instead starting in 2.4.0.

Introduction

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.

```
1  use Zend\InputFilter\InputFilter;
2  use Zend\InputFilter\Input;
3  use Zend\Validator;
4
5  $email = new Input('email');
6  $email->getValidatorChain()
7      ->attach(new Validator\EmailAddress());
8
9  $password = new Input('password');
10 $password->getValidatorChain()
11     ->attach(new Validator\StringLength(8));
12
13 $inputFilter = new InputFilter();
14 $inputFilter->add($email)
15     ->add($password)
16     ->setData($_POST);
17
18 if ($inputFilter->isValid()) {
19     echo "The form is valid\n";
20 } else {
21     echo "The form is not valid\n";
22     foreach ($inputFilter->getInvalidInput() as $error) {
23         print_r($error->getMessages());
24     }
25 }
```

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 `attach()` 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.

```

1 // As individual parameters
2 $inputFilter->setValidationGroup('email', 'password');
3
4 // or as an array of names
5 $inputFilter->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.

```

1 use Zend\InputFilter\Input;
2 use Zend\InputFilter\InputFilter;
3
4 $input = new Input('foo');
5 $input->getFilterChain()
6     ->attachByName('stringtrim')
7     ->attachByName('alpha');
8
9 $inputFilter = new InputFilter();
10 $inputFilter->add($input)
11     ->setData(array(
12         'foo' => ' Bar3 ',
13     ));
14
15 echo "Before:\n";
16 echo $inputFilter->getRawValue('foo') . "\n"; // the output is ' Bar3 '
17 echo "After:\n";
18 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):

```

1 use Zend\InputFilter\Factory;
2
3 $factory = new Factory();
4 $inputFilter = $factory->createInputFilter(array(
5     'password' => array(
6         'name' => 'password',
7         'required' => true,
8         'validators' => array(
9             array(
10                 'name' => 'not_empty',
11             ),
12             array(
13                 'name' => 'string_length',
14                 'options' => array(
15                     'min' => 8
16                 ),
17             ),
18         ),
19     ),
20 );
```

```

18         ),
19     ),
20 );
21
22 $inputFilter->setData($_POST);
23 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.

```

1  use Zend\InputFilter\InputFilter;
2
3  $filter = new InputFilter();
4
5  // Adding a single input
6  $filter->add(array(
7      'name' => 'username',
8      'required' => true,
9      'validators' => array(
10         array(
11             'name' => 'not_empty',
12         ),
13         array(
14             'name' => 'string_length',
15             'options' => array(
16                 'min' => 5
17             ),
18         ),
19     ),
20 );
21
22 // Adding another input filter what also contains a single input. Merging both.
23 $filter->add(array(
24     'type' => 'Zend\InputFilter\InputFilter',
25     'password' => array(
26         'name' => 'password',
27         'required' => true,
28         'validators' => array(
29             array(
30                 'name' => 'not_empty',
31             ),
32             array(
33                 'name' => 'string_length',
34                 'options' => array(
35                     'min' => 8
36                 ),
37             ),
38         ),
39     ),
40 );

```

The `merge()` method may be used on an InputFilterInterface in order to add two or more filters to each other, effectively allowing you to create chains of filters. This is especially useful in object hierarchies whereby we may define a generic set of validation rules on the base object and build these up to more specific rules along the way.

In the example below an `InputFilter` is built up for the name property as well as for the email property allowing them to be re-used elsewhere. When the `isValid()` method is called on the object, all of the merged filters are run against the calling object in order to validate the internal properties based on our compound set of filters.

```
1      use Zend\InputFilter\InputFilter;
2
3      /**
4       * Filter to ensure a name property is set and > 8 characters
5       */
6      class NameInputFilter extends InputFilter
7      {
8          /** Filter body goes here */
9      }
10
11     /**
12      * Filter to ensure an email property is set and > 8 characters and is valid
13      */
14     class EmailInputFilter extends InputFilter
15     {
16         /** Filter body goes here */
17     }
18
19     class SimplePerson
20     {
21         /** Member variables omitted for brevity */
22
23         /** @var InputFilter */
24         protected $inputFilter;
25
26         /**
27          * Retrieve input filter
28          *
29          * @return InputFilter
30          */
31         public function getInputFilter()
32         {
33             if (!$this->inputFilter) {
34                 // Create a new input filter
35                 $this->inputFilter = new InputFilter();
36                 // Merge our inputFilter in for the email property
37                 $this->inputFilter->merge(new EmailInputFilter());
38                 // Merge our inputFilter in for the name property
39                 $this->inputFilter->merge(new NameInputFilter());
40             }
41             return $this->inputFilter;
42         }
43
44         /**
45          * Set input filter
46          *
47          * @param InputFilterInterface $inputFilter
48          * @return SimplePerson
49          */
50         public function setInputFilter(InputFilterInterface $inputFilter)
51         {
52             $this->inputFilter = $inputFilter;
53
54             return $this;
```

```
55         }  
56     }
```

Also see

- *Zend\Filter*
- *Zend\Validator*

Input filter specifications

Zend\InputFilter allows configuration-driven creation of input filters via Zend\InputFilter\InputFilterAbstractServiceFactory. This abstract factory is responsible for creating and returning an appropriate input filter given named configuration under the top-level configuration key `input_filter_specs`.

It is registered with Zend\InputFilter\InputFilterPluginManager, allowing you to pull the input filter via that plugin manager. A side effect is that forms pulled from Zend\Form\FormElementManager can use these named input filters.

Setup

This functionality is disabled by default.

To enable it, you must add the Zend\InputFilter\InputFilterAbstractServiceFactory abstract factory to the Zend\InputFilter\InputFilterPluginManager configuration, which is under the `input_filters` configuration key.

```
1 return array(  
2     'input_filters' => array(  
3         'abstract_factories' => array(  
4             'Zend\InputFilter\InputFilterAbstractServiceFactory'  
5         ),  
6     ),  
7 );
```

Example

In the following code, we define configuration for an input filter named `foobar`:

```
1 return array(  
2     'input_filter_specs' => array(  
3         'foobar' => array(  
4             0 => array(  
5                 'name' => 'name',  
6                 'required' => true,  
7                 'filters' => array(  
8                     0 => array(  
9                         'name' => 'Zend\Filter\StringTrim',  
10                        'options' => array(),  
11                    ),  
12                ),  
13                'validators' => array(),  
14                'description' => 'Hello to name',  
15                'allow_empty' => false,  
16                'continue_if_empty' => false,  
17            ),  
18        ),  
19    );
```

When creating a controller, we might then pull the `InputFilterManager`, and retrieve the `foobar` input filter we've defined in order to inject it:

```
1 use Zend\ServiceManager\FactoryInterface;  
2 use Zend\ServiceManager\ServiceLocatorInterface;  
3  
4 class MyValidatingControllerFactory implements FactoryInterface  
5 {  
6     public function createService(ServiceLocatorInterface $controllers)  
7     {  
8         // Retrieve the application service manager  
9         $services = $controllers->getServiceLocator();  
10  
11        // Retrieve the InputFilterManager  
12        $filters = $services->get('InputFilterManager');  
13  
14        // Instantiate the controller and pass it the foobar input filter  
15        return new MyValidatingController($filters->get('foobar'));  
16    }  
17 }
```

And you can use it, as you already did with other input filters:

```
1 $inputFilter->setData(array(  
2     'name' => 'test',  
3 ));  
4  
5 if (! $inputFilter->isValid()) {  
6     echo 'Data invalid';  
7 }
```


The `Zend\InputFilter\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.

Basic Usage

Usage of `FileInput` is essentially the same as `Input`:

```
1 use Zend\Http\PhpEnvironment\Request;
2 use Zend\Filter;
3 use Zend\InputFilter\InputFilter;
4 use Zend\InputFilter\Input;
5 use Zend\InputFilter\FileInput;
6 use Zend\Validator;
7
8 // Description text input
9 $description = new Input('description'); // Standard Input type
10 $description->getFilterChain()           // Filters are run first w/ Input
11     ->attach(new Filter\StringTrim());
12 $description->getValidatorChain()        // Validators are run second w/ Input
13     ->attach(new Validator\StringLength(array('max' => 140)));
14
```

```
15 // File upload input
16 $file = new FileInput('file');           // Special File Input type
17 $file->getValidatorChain()                // Validators are run first w/ FileInput
18     ->attach(new Validator\File\UploadFile());
19 $file->getFilterChain()                    // Filters are run second w/ FileInput
20     ->attach(new Filter\File\RenameUpload(array(
21         'target'    => './data/tmpuploads/file',
22         'randomize' => true,
23     )));
24
25 // Merge $_POST and $_FILES data together
26 $request = new Request();
27 $postData = array_merge_recursive($request->getPost()->toArray(), $request->
    ↪getFiles()->toArray());
28
29 $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 }
```

Also see

- *File filter classes*
- *File validator classes*

CHAPTER 162

Introduction

`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 163

Basic Usage

Usage of `Zend\Json\Json` involves using the two public static methods available: `Zend\Json\Json::encode()` and `Zend\Json\Json::decode()`.

```
1 // Retrieve a value:
2 $phpNative = Zend\Json\Json::decode($encodedValue);
3
4 // Encode it to return to the client:
5 $json = Zend\Json\Json::encode($phpNative);
```

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()`.

```
1 // Encode it to return to the client:
2 $json = Zend\Json\Json::encode($phpNative);
3 if ($debug) {
4     echo Zend\Json\Json::prettyPrint($json, array("indent" => " "));
5 }
```

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.

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 `stdClass` objects. However, if you desire an associative array returned, you can specify this:

```
1 // Decode JSON objects as PHP array
2 $phpNative = Zend\Json\Json::decode($encodedValue, Zend\Json\Json::TYPE_ARRAY);
```

Any objects thus decoded are returned as associative arrays with keys and values 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`.

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()`.

```
1 // Encode PHP object recursively
2 $jsonObject = Zend\Json\Json::encode($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:

```
1 $jsonObject = Zend\Json\Json::encode(
2     $data,
3     true,
4     array('silenceCyclicalExceptions' => true)
5 );
```

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:

```
1 Zend\Json\Json::$useBuiltinEncoderDecoder = true;
```

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:

```
1 $data = array(
2     'onClick' => new Zend\Json\Expr('function() {'
3         . 'alert("I am a valid JavaScript callback '
4         . 'created by Zend\Json"); }'),
5     'other' => 'no expression',
6 );
7 $jsonObjectWithExpression = Zend\Json\Json::encode(
8     $data,
9     false,
10    array('enableJsonExprFinder' => true)
11 );
```

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:

```
1 // fromXml function simply takes a String containing XML contents
2 // as input.
3 $jsonContents = Zend\Json\Json::fromXml($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.

Example

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:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <books>
3   <book id="1">
```

```

4      <title>Code Generation in Action</title>
5      <author><first>Jack</first><last>Herrington</last></author>
6      <publisher>Manning</publisher>
7  </book>
8
9  <book id="2">
10     <title>PHP Hacks</title>
11     <author><first>Jack</first><last>Herrington</last></author>
12     <publisher>O'Reilly</publisher>
13 </book>
14
15 <book id="3">
16     <title>Podcasting Hacks</title>
17     <author><first>Jack</first><last>Herrington</last></author>
18     <publisher>O'Reilly</publisher>
19 </book>
20 </books>

```

JSON output string returned from `Zend\Json\Json::fromXml()` function:

```

1  {
2      "books" : {
3          "book" : [ {
4              "@attributes" : {
5                  "id" : "1"
6              },
7              "title" : "Code Generation in Action",
8              "author" : {
9                  "first" : "Jack", "last" : "Herrington"
10             },
11             "publisher" : "Manning"
12         }, {
13             "@attributes" : {
14                 "id" : "2"
15             },
16             "title" : "PHP Hacks", "author" : {
17                 "first" : "Jack", "last" : "Herrington"
18             },
19             "publisher" : "O'Reilly"
20         }, {
21             "@attributes" : {
22                 "id" : "3"
23             },
24             "title" : "Podcasting Hacks", "author" : {
25                 "first" : "Jack", "last" : "Herrington"
26             },
27             "publisher" : "O'Reilly"
28         }
29     ]}
30 }

```

More details about this `xml2json` feature can be found in the original proposal itself. Take a look at the [Zend_xml2json proposal](#).

Zend\Json\Server - JSON-RPC server

Introduction

`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.

Basic 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':

```
1  /**
2   * Calculator - sample class to expose via JSON-RPC
3   */
4  class Calculator
5  {
6      /**
7       * Return sum of two variables
8       *
9       * @param int $x
10      * @param int $y
11      * @return int
12      */
13     public function add($x, $y)
14     {
15         return $x + $y;
16     }
17
18     /**
19      * Return difference of two variables
20      *
21      * @param int $x
22      * @param int $y
23      * @return int
24      */
25     public function subtract($x, $y)
26     {
27         return $x - $y;
28     }
29
30     /**
31      * Return product of two variables
32      *
33      * @param int $x
34      * @param int $y
35      * @return int
36      */
37     public function multiply($x, $y)
38     {
39         return $x * $y;
40     }
41
42     /**
43      * Return the division of two variables
44      *
45      * @param int $x
46      * @param int $y
47      * @return float
48      */
49     public function divide($x, $y)
50     {
51         return $x / $y;
52     }
53 }
```

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:

```

1  $server = new Zend\Json\Server\Server();
2
3  // Indicate what functionality is available:
4  $server->setClass('Calculator');
5
6  // Handle the request:
7  $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:

```

1  $server = new Zend\Json\Server\Server();
2  $server->setClass('Calculator');
3
4  if ('GET' == $_SERVER['REQUEST_METHOD']) {
5      // Indicate the URL endpoint, and the JSON-RPC version used:
6      $server->setTarget('/json-rpc.php')
7          ->setEnvelope(Zend\Json\Server\Smd::ENV_JSONRPC_2);
8
9      // Grab the SMD
10     $smd = $server->getServiceMap();
11
12     // Return the SMD to the client
13     header('Content-Type: application/json');
14     echo $smd;
15     return;
16 }
17
18 $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:

```

1  $server = new Zend\Json\Server\Server();
2  $server->setClass('Calculator');
3
4  if ('GET' == $_SERVER['REQUEST_METHOD']) {
5      $server->setTarget('/json-rpc.php')
6          ->setEnvelope(Zend\Json\Server\Smd::ENV_JSONRPC_2);
7      $smd = $server->getServiceMap();
8
9      // Set Dojo compatibility:
10     $smd->setDojoCompatible(true);
11
12     header('Content-Type: application/json');
13     echo $smd;
14     return;
15 }
16
17 $server->handle();
    
```

Advanced Details

While most functionality for `Zend\Json\Server` is spelled out in [this section](#), more advanced functionality is available.

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

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.

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*.

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.

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.

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.

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):

```
1 $options = array(  
2     'host'           => 's0.foo.net',  
3     'username'       => 'CN=user1,DC=foo,DC=net',  
4     'password'       => 'pass1',  
5     'bindRequiresDn' => true,  
6     'accountDomainName' => 'foo.net',  
7     'baseDn'         => 'OU=Sales,DC=foo,DC=net',  
8 );
```

```
9 $ldap = new Zend\Ldap\Ldap($options);
10 $acctname = $ldap->getCanonicalAccountName('abaker',
11                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
12 echo "$acctname\n";
```

If you are using Microsoft AD a simple example is:

```
1 $options = array(
2     'host' => 'dc1.w.net',
3     'useStartTls' => true,
4     'username' => 'user1@w.net',
5     'password' => 'pass1',
6     'accountDomainName' => 'w.net',
7     'accountDomainNameShort' => 'W',
8     'baseDn' => 'CN=Users,DC=w,DC=net',
9 );
10 $ldap = new Zend\Ldap\Ldap($options);
11 $acctname = $ldap->getCanonicalAccountName('bcarter',
12                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
13 echo "$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.

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()`:

```
1 $options = array(
2     'host' => 's0.foo.net',
3     'username' => 'CN=user1,DC=foo,DC=net',
4     'password' => 'pass1',
5     'bindRequiresDn' => true,
6     'accountDomainName' => 'foo.net',
7     'baseDn' => 'OU=Sales,DC=foo,DC=net',
8 );
9 $ldap = new Zend\Ldap\Ldap($options);
10 $ldap->bind('abaker', 'moonbike55');
11 $acctname = $ldap->getCanonicalAccountName('abaker',
12                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
13 echo "$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**’.

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 167.1: 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 **EXAMPLE\abaker**.

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:

```

1  $acctname = 'W\\user2';
2  $password = 'pass2';
3
4  $multiOptions = array(
5      'server1' => array(
6          'host' => 's0.foo.net',
7          'username' => 'CN=user1,DC=foo,DC=net',
8          'password' => 'pass1',
9          'bindRequiresDn' => true,
10         'accountDomainName' => 'foo.net',
11         'accountDomainNameShort' => 'FOO',
12         'accountCanonicalForm' => 4, // ACCT_FORM_PRINCIPAL
13         'baseDn' => 'OU=Sales,DC=foo,DC=net',
14     ),
15     'server2' => array(
16         'host' => 'dc1.w.net',
17         'useSsl' => true,
18         'username' => 'user1@w.net',
19         'password' => 'pass1',
20         'accountDomainName' => 'w.net',
21         'accountDomainNameShort' => 'W',
22         'accountCanonicalForm' => 4, // ACCT_FORM_PRINCIPAL

```

```
23     'baseDn'                => 'CN=Users,DC=w,DC=net',
24     ),
25 );
26
27 $ldap = new Zend\Ldap\Ldap();
28
29 foreach ($multiOptions as $name => $options) {
30
31     echo "Trying to bind using server options for '$name'\n";
32
33     $ldap->setOptions($options);
34     try {
35         $ldap->bind($acctname, $password);
36         $acctname = $ldap->getCanonicalAccountName($acctname);
37         echo "SUCCESS: authenticated $acctname\n";
38         return;
39     } catch (Zend\Ldap\Exception\LdapException $zle) {
40         echo ' ' . $zle->getMessage() . "\n";
41         if ($zle->getCode() === Zend\Ldap\Exception\LdapException::LDAP_X_DOMAIN_
↪MISMATCH) {
42             continue;
43         }
44     }
45 }
```

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).

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 168.1: 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.

API Reference

Note: Method names in **italics** are static methods.

orphan

CHAPTER 169

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 \$attributes)
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, string \$sort)
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 \$alwaysEmulate)
Zend\Ldap\Ldap move(string Zend\Ldap\Dn \$from, string Zend\Ldap\Dn \$to, boolean \$recursively, boolean \$alwaysEmulate)

Method
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()

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 169.2: Zend\Ldap\Collection API

Method	Description
<code>__construct(Zend\Ldap\Collection\IteratorInterface \$iterator)</code>	Constructor. The constructor must be provided by a <code>Zend\Ldap\Collection\Iterator\Interface</code> which does the real result iteration. <code>Zend\Ldap\Collection\Iterator\Default</code> is the default implementation for iterating ext/ldap results.
<code>boolean close()</code>	Closes the internal iterator. This is also called in the destructor.
<code>array toArray()</code>	Returns all entries as an array.
<code>array getFirst()</code>	Returns the first entry in the collection or NULL if the collection is empty.

orphan

CHAPTER 170

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 170.1: 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.

orphan

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 171.1: 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 toLdapDateTime(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 toLdapBoolean(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().

orphan

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 172.1: 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 \$parentDn, string Zend\Ldap\Dn \$childDn)	Checks if given \$childDn is beneath \$parentDn subtree.

orphan

Table 173.1: 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 in the form: (attr {filtertype} {prepend} . \$value . \$append. Normally this constructor is not needed as all filters can be created by using the appropriate factory methods.
toString()	Returns a string representation of the filter.

orphan

CHAPTER 174

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
<code>Zend\Ldap\Ldap getLdap()</code>	Returns the current LDAP connection
<code>Zend\Ldap\Node attachLdap(Zend\Ldap\Ldap \$ldap)</code>	Attach the current LDAP connection
<code>Zend\Ldap\Node detachLdap()</code>	Detach node from LDAP connection
<code>boolean isAttached()</code>	Checks if the node is attached to a LDAP connection
<code>Zend\Ldap\Node create(string array Zend\Ldap\Dn \$dn, array \$objectClass)</code>	Factory method to create a new node
<code>Zend\Ldap\Node fromLdap(string array Zend\Ldap\Dn \$dn, Zend\Ldap\Ldap \$ldap)</code>	Factory method to create a new node from LDAP
<code>Zend\Ldap\Node fromArray(array \$data, boolean \$fromDataSource)</code>	Factory method to create a new node from array
<code>boolean isNew()</code>	Tells if the node is new
<code>boolean willBeDeleted()</code>	Tells if this node will be deleted
<code>Zend\Ldap\Node delete()</code>	Marks this node for deletion
<code>boolean willBeMoved()</code>	Tells if this node will be moved
<code>Zend\Ldap\Node update(Zend\Ldap\Ldap \$ldap)</code>	Sends all pending changes to LDAP
<code>Zend\Ldap\Dn getCurrentDn()</code>	Gets the current DN
<code>Zend\Ldap\Dn getDn()</code>	Gets the original DN
<code>string getDnString(string \$caseFold)</code>	Gets the original DN string
<code>array getDnArray(string \$caseFold)</code>	Gets the original DN array
<code>string getRdnString(string \$caseFold)</code>	Gets the RDN string
<code>array getRdnArray(string \$caseFold)</code>	Gets the RDN array
<code>Zend\Ldap\Node setDn(Zend\Ldap\Dn string array \$newDn)</code>	Sets the new DN
<code>Zend\Ldap\Node move(Zend\Ldap\Dn string array \$newDn)</code>	This is an alias for <code>setDn()</code>
<code>Zend\Ldap\Node rename(Zend\Ldap\Dn string array \$newDn)</code>	This is an alias for <code>setDn()</code>

Method	Description
array getObjectClass()	Returns the object class
Zend\Ldap\Node setObjectClass(array string \$value)	Sets the object class
Zend\Ldap\Node appendObjectClass(array string \$value)	Appends to the object class
string toLdif(array \$options)	Returns a LDAP entry in LDIF format
array getChangedData()	Gets changed data
array getChanges()	Returns all changes
string toString()	Returns the LDAP entry as a string
string __toString()	Cast to string
array toArray(boolean \$includeSystemAttributes)	Returns an array of LDAP entry data
string toJson(boolean \$includeSystemAttributes)	Returns a JSON representation of the LDAP entry
array getData(boolean \$includeSystemAttributes)	Returns the LDAP entry data
boolean existsAttribute(string \$name, boolean \$emptyExists)	Checks whether an attribute exists
boolean attributeHasValue(string \$name, mixed array \$value)	Checks if the attribute has a value
integer count()	Returns the number of attributes
mixed getAttribute(string \$name, integer null \$index)	Gets a LDAP attribute
array getAttributes(boolean \$includeSystemAttributes)	Gets all attributes
Zend\Ldap\Node setAttribute(string \$name, mixed \$value)	Sets a LDAP attribute
Zend\Ldap\Node appendToAttribute(string \$name, mixed \$value)	Appends to a LDAP attribute
array integer getDateAttribute(string \$name, integer null \$index)	Gets a LDAP date attribute
Zend\Ldap\Node setDateTimeAttribute(string \$name, integer array \$value, boolean \$utc)	Sets a LDAP date attribute
Zend\Ldap\Node appendToDateTimeAttribute(string \$name, integer array \$value, boolean \$utc)	Appends to a LDAP date attribute
Zend\Ldap\Node setPasswordAttribute(string \$password, string \$hashType, string \$attribName)	Sets a LDAP password attribute
Zend\Ldap\Node deleteAttribute(string \$name)	Deletes a LDAP attribute
void removeDuplicatesFromAttribute(string \$name)	Removes duplicate values from an attribute
void removeFromAttribute(string \$attribName, mixed array \$value)	Removes the value from an attribute
boolean exists(Zend\Ldap\Ldap \$ldap)	Checks if the LDAP entry exists
Zend\Ldap\Node reload(Zend\Ldap\Ldap \$ldap)	Reloads the LDAP entry
Zend\Ldap\Node\Collection searchSubtree(string Zend\Ldap\Filter\AbstractFilter \$filter, integer \$scope, string \$sort)	Searches the LDAP subtree
integer countSubtree(string Zend\Ldap\Filter\AbstractFilter \$filter, integer \$scope)	Count the number of entries in the subtree
integer countChildren()	Count the number of children
Zend\Ldap\Node\Collection searchChildren(string Zend\Ldap\Filter\AbstractFilter \$filter, string \$sort)	Searches the LDAP children
boolean hasChildren()	Returns whether the entry has children
Zend\Ldap\Node\ChildrenIterator getChildren()	Returns all children
Zend\Ldap\Node getParent(Zend\Ldap\Ldap \$ldap)	Returns the parent LDAP entry

orphan

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 to obvious reasons.

Table 175.1: 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, mixedarray \$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 getDateAttribute(string \$name, integer null \$index)	Gets a LDAP date/time attribute. Data conversion is applied using Zend\Ldap\Attribute::getDateAttribute().
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
930	Zend\Ldap\Node\RootDse::SERVER_TYPE_GENERIC for unknown LDAP servers Zend\Ldap\Node\RootDse::SERVER_TYPE_OPENLDAP for OpenLDAP server- s Zend\Ldap\Node\RootDse::SERVER_TYPE_ACTIVEDIRECTORY for Microsoft ActiveDirectory

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 175.2: `Zend\Ldap\Node\RootDse\OpenLdap` API

Method	Description
<code>integer getServerType()</code>	Gets the server type. Returns <code>Zend\Ldap\Node\RootDse::SERVER_TYPE_OPENLDAP</code>
<code>string null getConfigContext()</code>	Gets the <code>configContext</code> .
<code>string null getMonitorContext()</code>	Gets the <code>monitorContext</code> .
<code>boolean supportsControl(string[] \$oids)</code>	Determines if the control is supported.
<code>boolean supportsExtension(string[] \$oids)</code>	Determines if the extension is supported.
<code>boolean supportsFeature(string[] \$oids)</code>	Determines if the feature is supported.

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 175.3: Zend\Ldap\Node\RootDse\ActiveDirectory API

Method	Description
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_ACTIVEDIRECTORY
string null getConfigNamingContext()	Gets the configurationNamingContext.
string null getCurrentTime()	Gets the currentTime.
string null getDefaultNamingContext()	Gets the defaultNamingContext.
string null getDnsHostName()	Gets the dnsHostName.
string null getDomainControllerFunctionality()	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[] \$oids)	Determines if the capability is supported.
boolean supportsControl(string[] \$oids)	Determines if the control is supported.
boolean supportsPolicy(string[] \$policies)	Determines if the version is supported.

eDirectory

Additionally the common methods above apply to instances of *Zend\Ldap\Node\RootDse\Directory*.

Note: Refer to [Getting Information about the LDAP Server](#) for information on the attributes of Novell eDirectory RootDSE.

Table 175.4: Zend\Ldap\Node\RootDse\Directory 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”.

orphan

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 176.1: 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, mixedarray \$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 getDateAttribute(string \$name, integer null \$index)	Gets a LDAP date/time attribute. Data conversion is applied using Zend\Ldap\Attribute::getDateAttribute().
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 176.2: 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 176.3: 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 types Zend\Ldap\Node\Schema::OBJECTCLASS_TYPE_STRUCTURALfor structural classes Zend\Ldap\Node\Schema::OBJECTCLASS_TYPE_ABSTRACTfor abstract classes Zend\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 176.4: 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.

OpenLDAP

Additionally the common methods above apply to instances of *ZendLdapNodeSchemaOpenLDAP*.

Table 176.5: 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 176.6: 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 176.7: Zend\Ldap\Node\Schema\ObjectClass\OpenLDAP API

Method	Description
array get-Parents()	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.

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 176.8: Zend\Ldap\Node\Schema\AttributeType\ActiveDirectory API

Zend\Ldap\Node\Schema\AttributeType\ActiveDirectory does not provide any additional methods.
--

Table 176.9: Zend\Ldap\Node\Schema\ObjectClass\ActiveDirectory API

Zend\Ldap\Node\Schema\ObjectClass\ActiveDirectory does not provide any additional methods.
--

orphan

Zend\Ldap\Ldif\Encoder

Table 177.1: 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.

Authentication scenarios

OpenLDAP

ActiveDirectory

Basic CRUD operations

Retrieving data from the LDAP

Getting an entry by its DN

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
5 /*
6  $hm is an array of the following structure
7  array(
8      'dn'          => 'cn=Hugo Müller,ou=People,dc=my,dc=local',
9      'cn'          => array('Hugo Müller'),
10     'sn'          => array('Müller'),
11     'objectclass' => array('inetOrgPerson', 'top'),
12     ...
13 )
14 */
```

Check for the existence of a given DN

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $isThere = $ldap->exists('cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Count children of a given DN

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $childrenCount = $ldap->countChildren(
5     'cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Searching the LDAP tree

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $result = $ldap->search('(objectclass=*)',
5     'ou=People,dc=my,dc=local',
6     Zend\Ldap\Ldap::SEARCH_SCOPE_ONE);
7 foreach ($result as $item) {
8     echo $item["dn"] . ': ' . $item['cn'][0] . PHP_EOL;
9 }
```

Adding data to the LDAP

Add a new entry to the LDAP

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $entry = array();
5 Zend\Ldap\Attribute::setAttribute($entry, 'cn', 'Hans Meier');
6 Zend\Ldap\Attribute::setAttribute($entry, 'sn', 'Meier');
7 Zend\Ldap\Attribute::setAttribute($entry, 'objectClass', 'inetOrgPerson');
8 $ldap->add('cn=Hans Meier,ou=People,dc=my,dc=local', $entry);
```

Deleting from the LDAP

Delete an existing entry from the LDAP

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $ldap->delete('cn=Hans Meier,ou=People,dc=my,dc=local');
```


Updating the LDAP

Update an existing entry on the LDAP

```

1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
5 Zend\Ldap\Attribute::setAttribute($hm, 'mail', 'mueller@my.local');
6 Zend\Ldap\Attribute::setPassword($hm,
7                                 'newPa$$w0rd',
8                                 Zend\Ldap\Attribute::PASSWORD_HASH_SHA1);
9 $ldap->update('cn=Hugo Müller,ou=People,dc=my,dc=local', $hm);

```

Extended operations

Copy and move entries in the LDAP

Copy a LDAP entry recursively with all its descendants

```

1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $ldap->copy('cn=Hugo Müller,ou=People,dc=my,dc=local',
5           'cn=Hans Meier,ou=People,dc=my,dc=local',
6           true);

```

Move a LDAP entry recursively with all its descendants to a different subtree

```

1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $ldap->bind();
4 $ldap->moveToSubtree('cn=Hugo Müller,ou=People,dc=my,dc=local',
5                    'ou=Dismissed,dc=my,dc=local',
6                    true);

```


Creation and modification of DN strings

Using the filter API to create search filters

Create simple LDAP filters

```
1 $f1 = Zend\Ldap\Filter::equals('name', 'value');           // (name=value)
2 $f2 = Zend\Ldap\Filter::begins('name', 'value');          // (name=value*)
3 $f3 = Zend\Ldap\Filter::ends('name', 'value');            // (name=*value)
4 $f4 = Zend\Ldap\Filter::contains('name', 'value');         // (name=*value*)
5 $f5 = Zend\Ldap\Filter::greater('name', 'value');          // (name>value)
6 $f6 = Zend\Ldap\Filter::greaterOrEqual('name', 'value');   // (name>=value)
7 $f7 = Zend\Ldap\Filter::less('name', 'value');             // (name<value)
8 $f8 = Zend\Ldap\Filter::lessOrEqual('name', 'value');       // (name<=value)
9 $f9 = Zend\Ldap\Filter::approx('name', 'value');           // (name~value)
10 $f10 = Zend\Ldap\Filter::any('name');                      // (name=*)
```

Create more complex LDAP filters

```
1 $f1 = Zend\Ldap\Filter::ends('name', 'value')->negate(); // (!(name=*value))
2
3 $f2 = Zend\Ldap\Filter::equals('name', 'value');
4 $f3 = Zend\Ldap\Filter::begins('name', 'value');
5 $f4 = Zend\Ldap\Filter::ends('name', 'value');
6
7 // (&(name=value)(name=value*)(name=*value))
8 $f5 = Zend\Ldap\Filter::andFilter($f2, $f3, $f4);
9
```

```
10 // (| (name=value) (name=value*) (name=*value))  
11 $f6 = Zend\Ldap\Filter::orFilter($f2, $f3, $f4);
```

Modify LDAP entries using the Attribute API

Object-oriented access to the LDAP tree using Zend\Ldap\Node

Basic CRUD operations

Retrieving data from the LDAP

Getting a node by its DN

Searching a node's subtree

Adding a new node to the LDAP

Deleting a node from the LDAP

Updating a node on the LDAP

Extended operations

Copy and move nodes in the LDAP

Tree traversal

Traverse LDAP tree recursively

```
1 $options = array(/* ... */);  
2 $ldap = new Zend\Ldap\Ldap($options);  
3 $ldap->bind();  
4 $ri = new RecursiveIteratorIterator($ldap->getBaseNode(),  
5                                     RecursiveIteratorIterator::SELF_FIRST);
```

```
6 foreach ($ri as $rdn => $n) {  
7     var_dump($n);  
8 }
```

Getting information from the LDAP server

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

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $rootdse = $ldap->getRootDse();
4 $serverType = $rootdse->getServerType();
```

Schema Browsing

Getting hands on the server schema

```
1 $options = array(/* ... */);
2 $ldap = new Zend\Ldap\Ldap($options);
3 $schema = $ldap->getSchema();
4 $classes = $schema->getObjectClasses();
```

OpenLDAP

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.

Serializing LDAP data to and from LDIF

Serialize a LDAP entry to LDIF

```

1  $data = array(
2      'dn' => 'uid=rogasawara,ou=,o=Airius',
3      'objectclass' => array('top',
4                              'person',
5                              'organizationalPerson',
6                              'inetOrgPerson'),
7      'uid' => array('rogasawara'),
8      'mail' => array('rogasawara@airius.co.jp'),
9      'givenname;lang-ja' => array(''),
10     'sn;lang-ja' => array(''),
11     'cn;lang-ja' => array(' '),
12     'title;lang-ja' => array(' '),
13     'preferredlanguage' => array('ja'),
14     'givenname' => array(''),
15     'sn' => array(''),
16     'cn' => array(' '),
17     'title' => array(' '),
18     'givenname;lang-ja;phonetic' => array(''),
19     'sn;lang-ja;phonetic' => array(''),
20     'cn;lang-ja;phonetic' => array(' '),
21     'title;lang-ja;phonetic' => array(' '),
22     'givenname;lang-en' => array('Rodney'),
23     'sn;lang-en' => array('Ogasawara'),
24     'cn;lang-en' => array('Rodney Ogasawara'),
25     'title;lang-en' => array('Sales, Director'),
26 );
27 $ldif = Zend\Ldap\Ldif\Encoder::encode($data, array('sort' => false,
28                                                    'version' => null));
29 /*
30 $ldif contains:
31 dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6Y0oLG89QWl1yaXVz

```

```
32 objectclass: top
33 objectclass: person
34 objectclass: organizationalPerson
35 objectclass: inetOrgPerson
36 uid: rogasawara
37 mail: rogasawara@airius.co.jp
38 givenname;lang-ja:: 44Ot44OJ44OL44O8
39 sn;lang-ja:: 5bCP56yg5Y6f
40 cn;lang-ja:: 5bCP56yg5Y6fI0ODreODieODi+ODvA==
41 title;lang-ja:: 5Za25qWt6Y0oI0mDqOmVtw==
42 preferredlanguage: ja
43 givenname:: 44Ot44OJ44OL44O8
44 sn:: 5bCP56yg5Y6f
45 cn:: 5bCP56yg5Y6fI0ODreODieODi+ODvA==
46 title:: 5Za25qWt6Y0oI0mDqOmVtw==
47 givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
48 sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
49 cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
50 title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
51 givenname;lang-en: Rodney
52 sn;lang-en: Ogasawara
53 cn;lang-en: Rodney Ogasawara
54 title;lang-en: Sales, Director
55 */
```

Deserialize a LDIF string into a LDAP entry

```
1 $ldif = "dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6Y0oLG89QWlyXVz
2 objectclass: top
3 objectclass: person
4 objectclass: organizationalPerson
5 objectclass: inetOrgPerson
6 uid: rogasawara
7 mail: rogasawara@airius.co.jp
8 givenname;lang-ja:: 44Ot44OJ44OL44O8
9 sn;lang-ja:: 5bCP56yg5Y6f
10 cn;lang-ja:: 5bCP56yg5Y6fI0ODreODieODi+ODvA==
11 title;lang-ja:: 5Za25qWt6Y0oI0mDqOmVtw==
12 preferredlanguage: ja
13 givenname:: 44Ot44OJ44OL44O8
14 sn:: 5bCP56yg5Y6f
15 cn:: 5bCP56yg5Y6fI0ODreODieODi+ODvA==
16 title:: 5Za25qWt6Y0oI0mDqOmVtw==
17 givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
18 sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
19 cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
20 title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
21 givenname;lang-en: Rodney
22 sn;lang-en: Ogasawara
23 cn;lang-en: Rodney Ogasawara
24 title;lang-en: Sales, Director";
25 $data = Zend\Ldap\Ldif\Encoder::decode($ldif);
26 /*
27 $data = array(
28     'dn'                                     => 'uid=rogasawara,ou=,o=Airius',
```

```

29     'objectclass'                => array('top',
30                                         'person',
31                                         'organizationalPerson',
32                                         'inetOrgPerson'),
33     'uid'                        => array('rogasawara'),
34     'mail'                      => array('rogasawara@airius.co.jp'),
35     'givenname;lang-ja'        => array(''),
36     'sn;lang-ja'               => array(''),
37     'cn;lang-ja'               => array(' '),
38     'title;lang-ja'            => array(' '),
39     'preferredlanguage'        => array('ja'),
40     'givenname'                => array(''),
41     'sn'                       => array(''),
42     'cn'                       => array(' '),
43     'title'                    => array(' '),
44     'givenname;lang-ja;phonetic' => array(''),
45     'sn;lang-ja;phonetic'      => array(''),
46     'cn;lang-ja;phonetic'      => array(' '),
47     'title;lang-ja;phonetic'   => array(' '),
48     'givenname;lang-en'        => array('Rodney'),
49     'sn;lang-en'               => array('Ogasawara'),
50     'cn;lang-en'               => array('Rodney Ogasawara'),
51     'title;lang-en'            => array('Sales, Director'),
52 );
53 */

```

The AutoloaderFactory

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.

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:

```
1 $config = array(  
2     'Zend\Loader\ClassMapAutoloader' => array(  
3         'application' => APPLICATION_PATH . '/.classmap.php',  
4         'zf'          => APPLICATION_PATH . '/../library/Zend/.classmap.php',  
5     ),  
6     'Zend\Loader\StandardAutoloader' => array(  
7         'namespaces' => array(  
8             'Phly\Mustache' => APPLICATION_PATH . '/../library/Phly/Mustache',  
9             'Doctrine'     => APPLICATION_PATH . '/../library/Doctrine',  
10        ),  
11    ),  
12 );
```

An equivalent INI-style configuration might look like the following:

```
1 Zend\Loader\ClassMapAutoloader.application = APPLICATION_PATH ".classmap.php"  
2 Zend\Loader\ClassMapAutoloader.zf         = APPLICATION_PATH "../library/Zend/.  
↪ classmap.php"
```

```

3 Zend\Loader\StandardAutoloader.namespaces.Phly\Mustache = APPLICATION_PATH "../
  ↳library/Phly/Mustache"
4 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`.

```

1 // This example assumes ZF is on your include_path.
2 // You could also load the factory class from a path relative to the
3 // current script, or via an absolute path.
4 require_once 'Zend/Loader/AutoloaderFactory.php';
5 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.

Configuration 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.

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`.

Examples

Please see the *Quick Start* for a detailed example.

The StandardAutoloader

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:

```
1 $filename = str_replace(array('_', '\\'), DIRECTORY_SEPARATOR, $classname)
2     . '.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 autoloading, 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.

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

```
1 // This example assumes ZF is on your include_path.
2 // You could also load the autoloader class from a path relative to the
3 // current script, or via an absolute path.
4 require_once 'Zend/Loader/StandardAutoloader.php';
5 $loader = new Zend\Loader\StandardAutoloader(array('autoregister_zf' => true));
6
7 // Register the "Phly" namespace:
8 $loader->registerNamespace('Phly', APPLICATION_PATH . '/../library/Phly');
9
10 // Register the "Scapi" vendor prefix:
11 $loader->registerPrefix('Scapi', APPLICATION_PATH . '/../library/Scapi');
12
13 // Optionally, specify the autoloader as a "fallback" autoloader;
14 // this is not recommended.
15 $loader->setFallbackAutoloader(true);
16
17 // Register with spl_autoload:
18 $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.

```
1 require_once 'Zend/Loader/StandardAutoloader.php';
2 $loader = new Zend\Loader\StandardAutoloader(array(
3     'autoregister_zf' => true,
4     'namespaces' => array(
5         'Phly' => APPLICATION_PATH . '/../library/Phly',
```

```

6      ),
7      'prefixes' => array(
8          'Scapi' => APPLICATION_PATH . '/../library/Scapi',
9      ),
10     'fallback_autoloader' => true,
11 );
12
13 // Register with spl_autoload:
14 $loader->register();

```

Configuration Options

The `StandardAutoloader` defines the following 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.

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()`.

Examples

Please review the *examples in the quick start* for usage.

The ClassMapAutoloader

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.

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.

```
1 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.

```
1 // This example assumes ZF is on your include_path.  
2 // You could also load the autoloader class from a path relative to the  
3 // current script, or via an absolute path.  
4 require_once 'Zend/Loader/ClassMapAutoloader.php';  
5 $loader = new Zend\Loader\ClassMapAutoloader();  
6  
7 // Register the class map:  
8 $loader->registerAutoloadMap('Some/Directory/autoload_classmap.php');  
9  
10 // Register with spl_autoload:  
11 $loader->register();
```

At this point, you may now use any classes referenced in your class map.

Configuration Options

The `ClassMapAutoloader` defines the following 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:

```
1 // Configuration defining both a file-based class map, and an array map
2 $config = array(
3     __DIR__ . '/library/autoloader_classmap.php', // file-based class map
4     array(                                       // array class map
5         'Application\Bootstrap' => __DIR__ . '/application/Bootstrap.php',
6         'Test\Bootstrap'        => __DIR__ . '/tests/Bootstrap.php',
7     ),
8 );
```

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()`.

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:

```

1 // Configuration defining both a file-based class map, and an array map
2 $config = array(
3     APPLICATION_PATH . '/../library/autoloader_classmap.php', // file-based class map
4     array( // array class map
5         'Application\Bootstrap' => APPLICATION_PATH . '/Bootstrap.php',
6         'Test\Bootstrap'       => APPLICATION_PATH . '/../tests/Bootstrap.php',
7     ),
8 );

```

An equivalent INI style configuration might look like this:

```

1 classmap.library = APPLICATION_PATH "../library/autoloader_classmap.php"
2 classmap.resources.Application\Bootstrap = APPLICATION_PATH "/Bootstrap.php"
3 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()`.

```

1 /* The following are all equivalent */
2
3 // To the constructor:
4 $loader = new Zend\Loader\ClassMapAutoloader($config);
5
6 // To setOptions():
7 $loader = new Zend\Loader\ClassMapAutoloader();
8 $loader->setOptions($config);
9
10 // To registerAutoloadMaps():
11 $loader = new Zend\Loader\ClassMapAutoloader();
12 $loader->registerAutoloadMaps($config);

```

The ModuleAutoloader

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.

Quickstart

As the `ModuleAutoloader` is meant to be used with the `ModuleManager`, for examples of its usage and how to configure it, please see the [Module Autoloader Usage](#) section of the `ModuleManager` documentation.

Configuration Options

The `ModuleAutoloader` defines the following 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.

```
1 $options = array(  
2     '/path/to/modules',  
3     '/path/to/other/modules',  
4     'MyModule' => '/explicit/path/mymodule-v1.2'  
5 );
```

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.

Examples

Please review the *[examples in the quick start](#)* for usage.

The SplAutoloader Interface

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.

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.

```
1 namespace Custom;
2
3 use Zend\Loader\SplAutoloader;
4
5 class ModifiedIncludePathAutoloader implements SplAutoloader
6 {
7     protected $paths = array();
8
9     public function __construct($options = null)
10     {
11         if (null !== $options) {
12             $this->setOptions($options);
13         }
14     }
15
16     public function setOptions($options)
```

```
17 {
18     if (!is_array($options) && !($options instanceof \Traversable)) {
19         throw new \InvalidArgumentException();
20     }
21
22     foreach ($options as $path) {
23         if (!in_array($path, $this->paths)) {
24             $this->paths[] = $path;
25         }
26     }
27     return $this;
28 }
29
30 public function autoload($classname)
31 {
32     $filename = $classname . '.php';
33     foreach ($this->paths as $path) {
34         $test = $path . DIRECTORY_SEPARATOR . $filename;
35         if (file_exists($test)) {
36             return include($test);
37         }
38     }
39     return false;
40 }
41
42 public function register()
43 {
44     spl_autoload_register(array($this, 'autoload'));
45 }
46 }
```

To use this ModifiedIncludePathAutoloader from the previous example:

```
1 $options = array(
2     '/path/one',
3     '/path/two'
4 );
5 $autoloader = new Custom\ModifiedIncludePathAutoloader($options);
6 $autoloader->register();
```

Configuration Options

This component defines no configuration options, as it is an interface.

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:

```
1 public function register()
2 {
3     spl_autoload_register(array($this, 'autoload'));
4 }
```

Examples

Please see the [Quick Start](#) for a complete example.

The PluginClassLoader

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.

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.

```
1 use Zend\Http\HeaderLoader;
2
3 // Provide a global map, or override defaults:
4 HeaderLoader::addStaticMap(array(
5     'xrequestedfor' => 'My\Http\Header\XRequestedFor',
6 ));
7
8 // Instantiate the loader:
9 $loader = new Zend\Http\HeaderLoader();
10
11 // Register a new plugin:
12 $loader->registerPlugin('xForwardedFor', 'My\Http\Header\XForwardedFor');
```

```
13 // Load/retrieve the associated plugin class:
14 $class = $loader->load('xrequestedfor'); // 'My\Http\Header\XRequestedFor'
15
```

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.

Configuration 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.

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.

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:

```
1 use Zend\Loader\PluginClassLoader;
2
3 PluginClassLoader::addStaticMap(array(
4     'xrequestedfor' => 'My\Http\Header\XRequestedFor',
5 ));
```

Any later instances created will now have this map defined, allowing you to load that plugin.

```
1 use Zend\Loader\PluginClassLoader;
2
3 $loader = new PluginClassLoader();
4 $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.

```

1 namespace My\Plugins;
2
3 use Zend\Loader\PluginClassLoader;
4
5 class PluginLoader extends PluginClassLoader
6 {
7     /**
8      * @var array Plugin map
9      */
10    protected $plugins = array(
11        'foo'    => 'My\Plugins\Foo',
12        'bar'    => 'My\Plugins\Bar',
13        'foobar' => 'My\Plugins\FooBar',
14    );
15 }

```

At this point, you can simply instantiate the map and use it.

```

1 $loader = new My\Plugins\PluginLoader();
2 $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:

```

1 namespace My\Plugins;
2
3 use Zend\Loader\PluginClassLoader;
4
5 class PluginLoader extends PluginClassLoader
6 {
7     protected static $staticMap = array();
8
9     // ...
10 }

```

To inject the static map, use the extending class' name to call the static `addStaticMap()` method.

```

1 PluginLoader::addStaticMap(array(
2     'baz'    => 'My\Plugins\Baz',
3 ));

```

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()`.

```

1 namespace My\Plugins;
2
3 use Zend\Loader\PluginClassLoader;

```

```
4 use Zend\Http\HeaderLoader;
5
6 class PluginLoader extends PluginClassLoader
7 {
8     /**
9      * @var array Plugin map
10     */
11     protected $plugins = array(
12         'foo' => 'My\Plugins\Foo',
13         'bar' => 'My\Plugins\Bar',
14         'foobar' => 'My\Plugins\FooBar',
15     );
16 }
17
18 // Inject in constructor:
19 $loader = new HeaderLoader('My\Plugins\PluginLoader');
20 $loader = new HeaderLoader(new PluginLoader());
21
22 // Or via registerPlugins():
23 $loader->registerPlugins('My\Plugins\PluginLoader');
24 $loader->registerPlugins(new PluginLoader());
```

The ShortNameLocator Interface

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*.

Quick Start

Implementing a `ShortNameLocator` is trivial, and requires only three methods, as shown below.

```
1 namespace Zend\Loader;
2
3 interface ShortNameLocator
4 {
5     public function isLoaded($name);
6     public function getClassName($name);
7     public function load($name);
8 }
```

Configuration Options

This component defines no configuration options, as it is an interface.

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.

Examples

Please see the [Quick Start](#) for the interface specification.

The PluginClassLocator interface

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.

Quick Start

Classes implementing the `PluginClassLocator` need to implement only three methods, as illustrated below.

```
1 namespace Zend\Loader;
2
3 interface PluginClassLocator
4 {
5     public function registerPlugin($shortName, $className);
6     public function unregisterPlugin($shortName);
7     public function getRegisteredPlugins();
8 }
```

Configuration Options

This component defines no configuration options, as it is an interface.

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.

Examples

Please see the [Quick Start](#) for the interface specification.

The Class Map Generator utility: bin/classmap_generator.php

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.

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/
```

Configuration 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 “autoloader_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.

Overview

`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**.

Creating a Log

To get started logging, instantiate a **Writer** and then pass it to a **Logger** instance:

```
1 $logger = new Zend\Log\Logger;
2 $writer = new Zend\Log\Writer\Stream('php://output');
3
4 $logger->addWriter($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:

```
1 $logger = new Zend\Log\Logger;
2
3 $logger->addWriter('stream', null, array('stream' => 'php://output'));
```

In this example we passed the stream `php://output` as parameter (as array).

Logging Messages

To log a message, call the `log()` method of a Log instance and pass it the message with a corresponding priority:

```
1 $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:

```
1 $logger->log(Zend\Log\Logger::INFO, 'Informational message');
2 $logger->info('Informational message');
3
4 $logger->log(Zend\Log\Logger::EMERG, 'Emergency message');
5 $logger->emerg('Emergency message');
```

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:

```
1 $logger = null;
```

Explicitly destroying the log in this way is optional and is performed automatically at *PHP* shutdown.

Using Built-in Priorities

The `Zend\Log\Logger` class defines the following priorities:

```
1 EMERG   = 0; // Emergency: system is unusable
2 ALERT   = 1; // Alert: action must be taken immediately
3 CRIT    = 2; // Critical: critical conditions
4 ERR     = 3; // Error: error conditions
5 WARN    = 4; // Warning: warning conditions
6 NOTICE = 5; // Notice: normal but significant condition
7 INFO    = 6; // Informational: informational messages
8 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.

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.

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.

```

1 $logger = new Zend\Log\Logger;
2 $writer = new Zend\Log\Writer\Stream('php://output');
3
4 $logger->addWriter($writer);
5
6 Zend\Log\Logger::registerErrorHandler($logger);

```

If you want to unregister the error handler you can use the `unregisterErrorHandler()` static method.

Table 192.1: `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, <code>errcontext</code> 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)`.

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.

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`).

```
1 $writer = new Zend\Log\Writer\Stream('php://output');
2 $logger = new Zend\Log\Logger();
3 $logger->addWriter($writer);
4
5 $logger->info('Informational message');
```

To write data to a file, use one of the [Filesystem URLs](#):

```
1 $writer = new Zend\Log\Writer\Stream('/path/to/logfile');
2 $logger = new Zend\Log\Logger();
3 $logger->addWriter($writer);
4
5 $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:

```
1 $stream = @fopen('/path/to/logfile', 'a', false);
2 if (! $stream) {
3     throw new Exception('Failed to open stream');
4 }
```

```
5
6 $writer = new Zend\Log\Writer\Stream($stream);
7 $logger = new Zend\Log\Logger();
8 $logger->addWriter($writer);
9
10 $logger->info('Informational message');
```

You cannot specify the mode for existing stream resources. Doing so causes a `Zend\Log\Exception` to be thrown.

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:

```
1 $dbconfig = array(
2     // Sqlite Configuration
3     'driver' => 'Pdo',
4     'dsn' => 'sqlite:' . __DIR__ . '/tmp/sqlite.db',
5 );
6 $db = new Zend\Db\Adapter\Adapter($dbconfig);
7
8 $writer = new Zend\Log\Writer\Db($db, 'log_table_name');
9 $logger = new Zend\Log\Logger();
10 $logger->addWriter($writer);
11
12 $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.

```
1 $dbconfig = array(
2     // Sqlite Configuration
3     'driver' => 'Pdo',
4     'dsn' => 'sqlite:' . __DIR__ . '/tmp/sqlite.db',
5 );
6 $db = new Zend\Db\Adapter\Adapter($dbconfig);
7
8 $mapping = array(
9     'timestamp' => 'date',
10    'priority'   => 'type',
11    'message'    => 'event'
12 );
13 $writer = new Zend\Log\Writer\Db($db, 'log_table_name', $mapping);
14 $logger = new Zend\Log\Logger();
15 $logger->addWriter($writer);
16
17 $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.

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`:

```
1 {
2     [ .. ]
3
4     "repositories": [{
5         "type" : "pear",
6         "url" : "pear.firephp.org",
7         "vendor-alias" : "firephp"
8     }],
9     "minimum-stability": "dev",
10    "require" : {
11        [ ... ]
12        "firephp/FirePHPCore" : "*"
13    }
14 }
15
```

Writing to ChromePHP

`Zend\Log\Writer\ChromePHP` sends log data to the [ChromePHP](#) Chrome extension.

To use the ChromePHP writer, you will also need to include the [ChromePHP Library](#) library in your application include path. The easiest way to do this is to include the library in your `composer.json` file.

Writing to Mail

Writing to MongoDB

Writing to Syslog

Writing to Zend Monitor

Stubbing Out the Writer

The `Zend\Log\Writer\Noop` is a stub that does not write log data to anything. It is useful for disabling logging or stubbing out logging during tests:

```
1 $writer = new Zend\Log\Writer\Noop;
2 $logger = new Zend\Log\Logger();
3 $logger->addWriter($writer);
4
5 // goes nowhere
6 $logger->info('Informational message');
```

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `null` is a reserved keyword, which required renaming the `Null` log writer. If you were using the `Null` writer directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `Noop` class instead.

Users pulling their `Null` writer instance from the writer plugin manager receive a `Noop` instance instead starting in 2.4.0.

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.

```
1 $mock = new Zend\Log\Writer\Mock;
2 $logger = new Zend\Log\Logger();
3 $logger->addWriter($mock);
4
5 $logger->info('Informational message');
6
7 var_dump($mock->events[0]);
8
9 // Array
10 // (
11 //     [timestamp] => 2007-04-06T07:16:37-07:00
12 //     [message] => Informational message
13 //     [priority] => 6
14 //     [priorityName] => INFO
15 // )
```

To clear the events logged by the mock, simply set `$mock->events = array()`.

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:

```
1 $writer1 = new Zend\Log\Writer\Stream('/path/to/first/logfile');
2 $writer2 = new Zend\Log\Writer\Stream('/path/to/second/logfile');
3
4 $logger = new Zend\Log\Logger();
5 $logger->addWriter($writer1);
6 $logger->addWriter($writer2);
7
```

```
8 // goes to both writers
9 $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.

Overview

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:

```
1 use Zend\Log\Logger;
2
3 $logger = new Logger();
4
5 $writer1 = new Zend\Log\Writer\Stream('/path/to/first/logfile');
6 $logger->addWriter($writer1);
7
8 $writer2 = new Zend\Log\Writer\Stream('/path/to/second/logfile');
9 $logger->addWriter($writer2);
10
11 // add a filter only to writer2
12 $filter = new Zend\Log\Filter\Priority(Logger::CRIT);
13 $writer2->addFilter($filter);
14
15 // logged to writer1, blocked from writer2
16 $logger->info('Informational message');
17
18 // logged by both writers
19 $logger->emerg('Emergency message');
```

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.
- **Timestamp**, filters log events based on the time when they were triggered. It can be configured by specifying either `idate()`-compliant format character along with the desired value, or a full `DateTime` instance. Appropriate comparison operator must also be supplied in either cases.
- **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.

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.

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:

```
1 $format = '%timestamp% %priorityName% (%priority%): %message%' . PHP_EOL;
2 $formatter = new Zend\Log\Formatter\Simple($format);
```

A formatter is set on an individual Writer object using the Writer's `setFormatter()` method:

```
1 $writer = new Zend\Log\Writer\Stream('php://output');
2 $formatter = new Zend\Log\Formatter\Simple('hello %message%' . PHP_EOL);
3 $writer->setFormatter($formatter);
4
5 $logger = new Zend\Log\Logger();
6 $logger->addWriter($writer);
7
8 $logger->info('there');
9
10 // 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`.

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:

```
1 $writer = new Zend\Log\Writer\Stream('php://output');
2 $formatter = new Zend\Log\Formatter\Xml();
3 $writer->setFormatter($formatter);
4
5 $logger = new Zend\Log\Logger();
6 $logger->addWriter($writer);
7
8 $logger->info('informational message');
```

The code above outputs the following *XML* (space added for clarity):

```
1 <logEntry>
2   <timestamp>2007-04-06T07:24:37-07:00</timestamp>
3   <message>informational message</message>
4   <priority>6</priority>
5   <priorityName>INFO</priorityName>
6 </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:

```
1 $writer = new Zend\Log\Writer\Stream('php://output');
2 $formatter = new Zend\Log\Formatter\Xml('log',
3                                     array('msg' => 'message',
4                                           'level' => 'priorityName')
5                                     );
6 $writer->setFormatter($formatter);
7
8 $logger = new Zend\Log\Logger();
9 $logger->addWriter($writer);
10
11 $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:

```
1 <log>
2   <msg>informational message</msg>
3   <level>INFO</level>
4 </log>
```

Formatting to FirePhp

Zend\Log\Formatter\FirePhp formats log data for the [Firebug](#) extension for Firefox.

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 Zend\Mail

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:

```
1 use Zend\Mail;
2
3 $mail = new Mail\Message();
4 $mail->setBody('This is the text of the email.');
```

```
5 $mail->setFrom('Freeaqingme@example.org', 'Sender\'s name');
6 $mail->addTo('Matthew@example.com', 'Name of recipient');
7 $mail->setSubject('TestSubject');
```

```
8
9 $transport = new Mail\Transport\Sendmail();
10 $transport->send($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.

```
1 use Zend\Mail;
2
3 $mail = new Mail\Message();
4 $mail->setBody('This is the text of the mail.')
5     ->setFrom('somebody@example.com', 'Some Sender')
6     ->addTo('somebody_else@example.com', 'Some Recipient')
7     ->setSubject('TestSubject');
```

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

This example shows how to change the Return-Path of the `mail()` function.

```
1 use Zend\Mail;
2
3 $mail = new Mail\Message();
4 $mail->setBody('This is the text of the email.');
5 $mail->setFrom('Freeaqingme@example.org', 'Dolf');
6 $mail->addTo('matthew@example.com', 'Matthew');
7 $mail->setSubject('TestSubject');
8
9 $transport = new Mail\Transport\Sendmail('-freturn_to_me@example.com');
10 $transport->send($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 197

Zend\Mail\Message

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.

Quick Start

Creating a `Message` is simple: simply instantiate it.

```
1 use Zend\Mail\Message;
2
3 $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:

```
1 $message->addFrom("matthew@zend.com", "Matthew Weier O'Phinney")
2     ->addTo("foobar@example.com")
3     ->setSubject("Sending an email from Zend\Mail!");
4 $message->setBody("This is the message body.");
```

You can also add recipients to carbon-copy ("Cc:") or blind carbon-copy ("Bcc:").

```
1 $message->addCc("ralph.schindler@zend.com")
2     ->addBcc("enrico.z@zend.com");
```

If you want to specify an alternate address to which replies may be sent, that can be done, too.

```
1 $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.

```
1 /*
2  * Mail headers created:
3  * From: Ralph Schindler <ralph.schindler@zend.com>, Enrico Zimuel <enrico.z@zend.com>
4  * Sender: Matthew Weier O'Phinney <matthew@zend.com></matthew>
5  */
6 $message->addFrom("ralph.schindler@zend.com", "Ralph Schindler")
7     ->addFrom("enrico.z@zend.com", "Enrico Zimuel")
8     ->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.

```
1 $message->setEncoding("UTF-8");
```

If you wish to set other headers, you can do that as well.

```
1 /*
2  * Mail headers created:
3  * X-API-Key: FOO-BAR-BAZ-BAT
4  */
5 $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.

In addition you can check how to add attachment to your message E-mail Attachments.

```
1 use Zend\Mail\Message;
2 use Zend\Mime\Message as MimeMessage;
3 use Zend\Mime\Part as MimePart;
4
5 $text = new MimePart($textContent);
6 $text->type = "text/plain";
7
8 $html = new MimePart($htmlMarkup);
9 $html->type = "text/html";
10
11 $image = new MimePart(fopen($pathToImage, 'r'));
12 $image->type = "image/jpeg";
13
14 $body = new MimeMessage();
15 $body->setParts(array($text, $html, $image));
16
17 $message = new Message();
18 $message->setBody($body);
```

If you want a string representation of your email, you can get that:

```
1 echo $message->toString();
```

Finally, you can fully introspect the message – including getting all addresses of recipients and senders, all headers, and the message body.

```

1 // Headers
2 // Note: this will also grab all headers for which accessors/mutators exist in
3 // the Message object itself.
4 foreach ($message->getHeaders() as $header) {
5     echo $header->toString();
6     // or grab values: $header->getFieldName(), $header->getFieldValue()
7 }
8
9 // The logic below also works for the methods cc(), bcc(), to(), and replyTo()
10 foreach ($message->getFrom() as $address) {
11     printf("%s: %s\n", $address->getEmail(), $address->getName());
12 }
13
14 // Sender
15 $address = $message->getSender();
16 if(!is_null($address)) {
17     printf("%s: %s\n", $address->getEmail(), $address->getName());
18 }
19
20 // Subject
21 echo "Subject: ", $message->getSubject(), "\n";
22
23 // Encoding
24 echo "Encoding: ", $message->getEncoding(), "\n";
25
26 // Message body:
27 echo $message->getBody(); // raw body, or MIME object
28 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!

```

1 $transport->send($message);

```

Configuration Options

The Message class has no configuration options, and is instead a value object.

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.

getFrom `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.

Examples

Please *see the Quick Start section*.

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.

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

```
1 use Zend\Mail\Message;
2 use Zend\Mail\Transport\Sendmail as SendmailTransport;
3
4 $message = new Message();
5 $message->addTo('matthew@zend.com')
6           ->addFrom('ralph.schindler@zend.com')
7           ->setSubject('Greetings and Salutations!')
8           ->setBody("Sorry, I'm going to be late today!");
9
10 $transport = new SendmailTransport();
11 $transport->send($message);
```

SMTP Transport Usage

```
1 use Zend\Mail\Message;
2 use Zend\Mail\Transport\Smtp as SmtpTransport;
3 use Zend\Mail\Transport\SmtpOptions;
4
5 $message = new Message();
6 $message->addTo('matthew@zend.com')
7     ->addFrom('ralph.schindler@zend.com')
8     ->setSubject('Greetings and Salutations!')
9     ->setBody("Sorry, I'm going to be late today!");
10
11 // Setup SMTP transport using LOGIN authentication
12 $transport = new SmtpTransport();
13 $options = new SmtpOptions(array(
14     'name' => 'localhost.localdomain',
15     'host' => '127.0.0.1',
16     'connection_class' => 'login',
17     'connection_config' => array(
18         'username' => 'user',
19         'password' => 'pass',
20     ),
21 ));
22 $transport->setOptions($options);
23 $transport->send($message);
```

File Transport Usage

```
1 use Zend\Mail\Message;
2 use Zend\Mail\Transport\File as FileTransport;
3 use Zend\Mail\Transport\FileOptions;
4
5 $message = new Message();
6 $message->addTo('matthew@zend.com')
7     ->addFrom('ralph.schindler@zend.com')
8     ->setSubject('Greetings and Salutations!')
9     ->setBody("Sorry, I'm going to be late today!");
10
11 // Setup File transport
12 $transport = new FileTransport();
13 $options = new FileOptions(array(
14     'path' => 'data/mail/',
15     'callback' => function (FileTransport $transport) {
16         return 'Message_' . microtime(true) . '_' . mt_rand() . '.txt';
17     },
18 ));
19 $transport->setOptions($options);
20 $transport->send($message);
```

InMemory Transport Usage

```
1 use Zend\Mail\Message;
2 use Zend\Mail\Transport\InMemory as InMemoryTransport;
3
```

```

4 $message = new Message();
5 $message->addTo('matthew@zend.com')
6     ->addFrom('ralph.schindler@zend.com')
7     ->setSubject('Greetings and Salutations!')
8     ->setBody("Sorry, I'm going to be late today!");
9
10 // Setup InMemory transport
11 $transport = new InMemoryTransport();
12 $transport->send($message);
13
14 // Verify the message:
15 $received = $transport->getLastMessage();

```

The InMemory transport is primarily of interest when in development or when testing.

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `null` is a reserved keyword, which required renaming the `Null` transport. If you were using the `Null` transport directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `InMemory` class instead.

Users pulling their `Null` transport instance from the transport factory (`Zend\Mail\Transport\Factory`) receive an `InMemory` instance instead starting in 2.4.0.

Configuration Options

Configuration options are per transport. Please follow the links below for transport-specific options.

- [SMTP Transport Options](#)
- [File Transport Options](#)

Available Methods

send `send(Zend\Mail\Message $message)`

Send a mail message.

Returns void

Examples

Please see the [Quick Start section](#) for examples.

Zend\Mail\Transport\SmtOptions

Overview

This document details the various options available to the Zend\Mail\Transport\Smt mail transport.

Quick Start

Basic SMTP Transport Usage

```
1 use Zend\Mail\Transport\Smt as SmtTransport;
2 use Zend\Mail\Transport\SmtOptions;
3
4 // Setup SMTP transport
5 $transport = new SmtTransport();
6 $options   = new SmtOptions(array(
7     'name' => 'localhost.localdomain',
8     'host' => '127.0.0.1',
9     'port' => 25,
10 ));
11 $transport->setOptions($options);
```

SMTP Transport Usage with PLAIN AUTH

```
1 use Zend\Mail\Transport\Smt as SmtTransport;
2 use Zend\Mail\Transport\SmtOptions;
3
4 // Setup SMTP transport using PLAIN authentication
5 $transport = new SmtTransport();
6 $options   = new SmtOptions(array(
7     'name'           => 'localhost.localdomain',
```

```
8     'host'           => '127.0.0.1',
9     'connection_class' => 'plain',
10    'connection_config' => array(
11        'username' => 'user',
12        'password' => 'pass',
13    ),
14    ));
15    $transport->setOptions($options);
```

SMTP Transport Usage with LOGIN AUTH

```
1  use Zend\Mail\Transport\Smtp as SmtptTransport;
2  use Zend\Mail\Transport\SmtpOptions;
3
4  // Setup SMTP transport using LOGIN authentication
5  $transport = new SmtptTransport();
6  $options = new SmtptOptions(array(
7      'name'           => 'localhost.localdomain',
8      'host'           => '127.0.0.1',
9      'connection_class' => 'login',
10     'connection_config' => array(
11         'username' => 'user',
12         'password' => 'pass',
13     ),
14 ));
15 $transport->setOptions($options);
```

SMTP Transport Usage with CRAM-MD5 AUTH

```
1  use Zend\Mail\Transport\Smtp as SmtptTransport;
2  use Zend\Mail\Transport\SmtpOptions;
3
4  // Setup SMTP transport using CRAM-MD5 authentication
5  $transport = new SmtptTransport();
6  $options = new SmtptOptions(array(
7      'name'           => 'localhost.localdomain',
8      'host'           => '127.0.0.1',
9      'connection_class' => 'crammd5',
10     'connection_config' => array(
11         'username' => 'user',
12         'password' => 'pass',
13     ),
14 ));
15 $transport->setOptions($options);
```

SMTP Transport Usage with PLAIN AUTH over TLS

```
1  use Zend\Mail\Transport\Smtp as SmtptTransport;
2  use Zend\Mail\Transport\SmtpOptions;
3
4  // Setup SMTP transport using PLAIN authentication over TLS
5  $transport = new SmtptTransport();
```

```

6 $options = new SmtOptions(array(
7     'name' => 'example.com',
8     'host' => '127.0.0.1',
9     'port' => 587, // Notice port change for TLS is 587
10    'connection_class' => 'plain',
11    'connection_config' => array(
12        'username' => 'user',
13        'password' => 'pass',
14        'ssl' => 'tls',
15    ),
16));
17 $transport->setOptions($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\SmtLoader`. 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.

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.

Examples

Please see the [Quick Start](#) for examples.

Zend\Mail\Transport\FileOptions

Overview

This document details the various options available to the `Zend\Mail\Transport\File` mail transport.

Quick Start

```
1 use Zend\Mail\Transport\File as FileTransport;
2 use Zend\Mail\Transport\FileOptions;
3
4 // Setup File transport
5 $transport = new FileTransport();
6 $options = new FileOptions(array(
7     'path' => 'data/mail/',
8     'callback' => function (FileTransport $transport) {
9         return 'Message_' . microtime(true) . '_' . mt_rand() . '.txt';
10    },
11 ));
12 $transport->setOptions($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:

```
1 function (Zend\Mail\FileTransport $transport) {
2     return 'ZendMail_' . time() . '_' . mt_rand() . '.tmp';
3 }
```

Available Methods

`Zend\Mail\Transport\FileOptions` extends `Zend\Stdlib\AbstractOptions`, and inherits all functionality from that class; this includes property overloading. Additionally, the following explicit setters and getters are provided.

setPath `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.

Examples

Please see the *[Quick Start](#)* for examples.

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.

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 [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`.

```
1 use Zend\Math\Rand;
2
3 $bytes = Rand::getBytes(32, true);
4 printf("Random bytes (in Base64): %s\n", base64_encode($bytes));
5
6 $boolean = Rand::getBoolean();
7 printf("Random boolean: %s\n", $boolean ? 'true' : 'false');
8
9 $integer = Rand::getInteger(0,1000);
10 printf("Random integer in [0-1000]: %d\n", $integer);
11
12 $float = Rand::getFloat();
13 printf("Random float in [0-1): %f\n", $float);
14
15 $string = Rand::getString(32, 'abcdefghijklmnopqrstuvwxyz', true);
16 printf("Random string in latin alphabet: %s\n", $string);
```

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.

```

1 use Zend\Math\BigInteger\BigInteger;
2 use Zend\Math\Rand;
3
4 $bigInt = BigInteger::factory('bcmath');
5
6 $x = Rand::getString(100, '0123456789');
7 $y = Rand::getString(100, '0123456789');
8
9 $sum = $bigInt->add($x, $y);
10 $len = strlen($sum);
11
12 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.

```

1 use Zend\Math\BigInteger\BigInteger;
2 use Zend\Math\Rand;
3
4 $bigInt = BigInteger::factory('bcmath');
5
6 $digit = 100;
7 $x = '-' . Rand::getString($digit, '0123456789');
8
9 $byte = $bigInt->intToBin($x);
10
11 printf("The binary representation of the big integer with $digit digit:\n%s\nis (in_
↳Base64 format): %s\n",
12     $x, base64_encode($byte));
13 printf("Length in bytes: %d\n", strlen($byte));
14
15 $byte = $bigInt->intToBin($x, true);
16
17 printf("The two's complement binary representation of the big integer with $digit_
↳digit:\n%s\nis (in Base64 format): %s\n",
18     $x, base64_encode($byte));
19 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).

Introduction

The `Zend\Memory` component is intended to manage data in an environment with limited memory.

Memory objects (memory containers) are generated by memory manager by request and transparently swapped/loaded when it's necessary.

For example, if creating or loading a managed object would cause the total memory usage to exceed the limit you specify, some managed objects are copied to cache storage outside of memory. In this way, the total memory used by managed objects does not exceed the limit you need to enforce.

The memory manager uses `ZendCache` backends as storage providers.

Using ZendMemory component

`Zend\Memory\Memory::factory()` instantiates the memory manager class with specified backend options.

```
1  $backendOptions = array(  
2      'cache_dir' => './tmp/' // Directory where to put the swapped memory blocks  
3  );  
4  
5  $memoryManager = Zend\Memory\Memory::factory('File', $backendOptions);  
6  
7  $loadedFiles = array();  
8  
9  for ($count = 0; $count < 10000; $count++) {  
10     $f = fopen($fileNames[$count], 'rb');  
11     $data = fread($f, filesize($fileNames[$count]));  
12     fclose($f);  
13  
14     $loadedFiles[] = $memoryManager->create($data);  
15 }  
16
```

```
17 echo $loadedFiles[$index1]->value;
18
19 $loadedFiles[$index2]->value = $newValue;
20
21 $loadedFiles[$index3]->value[$charIndex] = '_';
```

Theory of Operation

Zend\Memory component operates with the following concepts:

- Memory manager
- Memory container
- Locked memory object
- Movable memory object

Memory manager

The memory manager generates memory objects (locked or movable) by request of user application and returns them wrapped into a memory container object.

Memory container

The memory container has a virtual or actual `value` attribute of string type. This attribute contains the data value specified at memory object creation time.

You can operate with this `value` attribute as an object property:

```
1 $memObject = $memoryManager->create($data);
2
3 echo $memObject->value;
4
5 $memObject->value = $newValue;
6
7 $memObject->value[$index] = '_';
8
9 echo ord($memObject->value[$index1]);
10
11 $memObject->value = substr($memObject->value, $start, $length);
```

Note: If you are using a *PHP* version earlier than 5.2, use the *getRef()* method instead of accessing the value property directly.

Locked memory

Locked memory objects are always stored in memory. Data stored in locked memory are never swapped to the cache backend.

Movable memory

Movable memory objects are transparently swapped and loaded to/from the cache backend by `Zend\Memory` when it's necessary.

The memory manager doesn't swap objects with size less than the specified minimum, due to performance considerations. See [this section](#) for more details.

Creating a Memory Manager

You can create new a memory manager (Zend\Memory\MemoryManager object) using its constructor: `Zend\Memory\MemoryManager::__construct (Zend\Cache\Storage\StorageInterface $cache = null)`.

```
1 $cache = Zend\Cache\StorageFactory::factory(array(  
2     'adapter' => array(  
3         'name' => 'Filesystem',  
4         'options' => array(  
5             'cache_dir' => './tmp/', // Directory where to put the swapped memory_  
6             ↪ blocks  
7         ),  
8     ),  
9 ));  
10 $memoryManager = new Zend\Memory\MemoryManager($cache);
```

Zend\Memory\MemoryManager uses *ZendCache storage adapters* to cache memory blocks; if no cache instance is provided the system temporary directory is used.

```
1 $memoryManager = new Zend\Memory\MemoryManager();
```

This is useful if you know that memory is not limited or the overall size of objects never reaches the memory limit.

Managing Memory Objects

This section describes creating and destroying objects in the managed memory, and settings to control memory manager behavior.

Creating Movable Objects

Create movable objects (objects, which may be swapped) using the `Zend\Memory\MemoryManager::create([$data])` method:

```
1 $memObject = $memoryManager->create($data);
```

The `$data` argument is optional and used to initialize the object value. If the `$data` argument is omitted, the value is an empty string.

Creating Locked Objects

Create locked objects (objects, which are not swapped) using the `Zend\Memory\MemoryManager::createLocked([$data])` method:

```
1 $memObject = $memoryManager->createLocked($data);
```

The `$data` argument is optional and used to initialize the object value. If the `$data` argument is omitted, the value is an empty string.

Destroying Objects

Memory objects are automatically destroyed and removed from memory when they go out of scope:

```
1 function foo()  
2 {  
3     global $memoryManager, $memList;  
4  
5     ...  
6  
7     $memObject1 = $memoryManager->create($data1);  
8     $memObject2 = $memoryManager->create($data2);  
9     $memObject3 = $memoryManager->create($data3);  
10  
11     ...  
12  
13     $memList[] = $memObject3;  
14  
15     ...  
16  
17     unset($memObject2); // $memObject2 is destroyed here  
18  
19     ...  
20     // $memObject1 is destroyed here  
21     // but $memObject3 object is still referenced by $memList  
22     // and is not destroyed  
23 }
```

This applies to both movable and locked objects.

Memory Manager Settings

Memory Limit

Memory limit is a number of bytes allowed to be used by loaded movable objects.

If loading or creation of an object causes memory usage to exceed of this limit, then the memory manager swaps some other objects.

You can retrieve or set the memory limit setting using the `getMemoryLimit()` and `setMemoryLimit($newLimit)` methods:

```
1 $oldLimit = $memoryManager->getMemoryLimit(); // Get memory limit in bytes
2 $memoryManager->setMemoryLimit($newLimit);    // Set memory limit in bytes
```

A negative value for memory limit means ‘no limit’.

The default value is two-thirds of the value of ‘memory_limit’ in `php.ini` or ‘no limit’ (-1) if ‘memory_limit’ is not set in `php.ini`.

MinSize

MinSize is a minimal size of memory objects, which may be swapped by memory manager. The memory manager does not swap objects that are smaller than this value. This reduces the number of swap/load operations.

You can retrieve or set the minimum size using the `getMinSize()` and `setMinSize($newSize)` methods:

```
1 $oldMinSize = $memoryManager->getMinSize(); // Get MinSize in bytes
2 $memoryManager->setMinSize($newSize);      // Set MinSize limit in bytes
```

The default minimum size value is 16KB (16384 bytes).

CHAPTER 204

Memory Objects

Movable

Create movable memory objects using the `create([$data])` method of the memory manager:

```
1 $memObject = $memoryManager->create($data);
```

“Movable” means that such objects may be swapped and unloaded from memory and then loaded when application code accesses the object.

Locked

Create locked memory objects using the `createLocked([$data])` method of the memory manager:

```
1 $memObject = $memoryManager->createLocked($data);
```

“Locked” means that such objects are never swapped and unloaded from memory.

Locked objects provides the same interface as movable objects (`Zend\Memory\Container\Interface`). So locked object can be used in any place instead of movable objects.

It’s useful if an application or developer can decide, that some objects should never be swapped, based on performance considerations.

Access to locked objects is faster, because the memory manager doesn’t need to track changes for these objects.

The locked objects class (`Zend\Memory\Container\Locked`) guarantees virtually the same performance as working with a string variable. The overhead is a single dereference to get the class property.

Memory container ‘value’ property

Use the memory container (movable or locked) ‘value’ property to operate with memory object data:

```
1 $memObject = $memoryManager->create($data);
2
3 echo $memObject->value;
4
5 $memObject->value = $newValue;
6
7 $memObject->value[$index] = '_';
8
9 echo ord($memObject->value[$index1]);
10
11 $memObject->value = substr($memObject->value, $start, $length);
```

An alternative way to access memory object data is to use the *getRef()* method. This method **must** be used for *PHP* versions before 5.2. It also may have to be used in some other cases for performance reasons.

Memory container interface

Memory container provides the following methods:

getRef() method

```
1 public function &getRef();
```

The *getRef()* method returns reference to the object value.

Movable objects are loaded from the cache at this moment if the object is not already in memory. If the object is loaded from the cache, this might cause swapping of other objects if the memory limit would be exceeded by having all the managed objects in memory.

The *getRef()* method **must** be used to access memory object data for *PHP* versions before 5.2.

Tracking changes to data needs additional resources. The *getRef()* method returns reference to string, which is changed directly by user application. So, it's a good idea to use the *getRef()* method for value data processing:

```
1 $memObject = $memoryManager->create($data);
2
3 $value = &$memObject->getRef();
4
5 for ($count = 0; $count < strlen($value); $count++) {
6     $char = $value[$count];
7     ...
8 }
```

touch() method

```
1 public function touch();
```

The *touch()* method should be used in common with *getRef()*. It signals that object value has been changed:

```
1 $memObject = $memoryManager->create($data);
2 ...
3
4 $value = &$memObject->getRef();
```



```

5
6 for ($count = 0; $count < strlen($value); $count++) {
7     ...
8     if ($condition) {
9         $value[$count] = $char;
10    }
11    ...
12 }
13
14 $memObject->touch();

```

lock() method

```

1 public function lock();

```

The `lock()` methods locks object in memory. It should be used to prevent swapping of some objects you choose. Normally, this is not necessary, because the memory manager uses an intelligent algorithm to choose candidates for swapping. But if you exactly know, that at this part of code some objects should not be swapped, you may lock them.

Locking objects in memory also guarantees that reference returned by the `getRef()` method is valid until you unlock the object:

```

1 $memObject1 = $memoryManager->create($data1);
2 $memObject2 = $memoryManager->create($data2);
3 ...
4
5 $memObject1->lock();
6 $memObject2->lock();
7
8 $value1 = &$memObject1->getRef();
9 $value2 = &$memObject2->getRef();
10
11 for ($count = 0; $count < strlen($value2); $count++) {
12     $value1 .= $value2[$count];
13 }
14
15 $memObject1->touch();
16 $memObject1->unlock();
17 $memObject2->unlock();

```

unlock() method

```

1 public function unlock();

```

`unlock()` method unlocks object when it's no longer necessary to be locked. See the example above.

isLocked() method

```

1 public function isLocked();

```

The `isLocked()` method can be used to check if object is locked. It returns `TRUE` if the object is locked, or `FALSE` if it is not locked. This is always `TRUE` for “locked” objects, and may be either `TRUE` or `FALSE` for “movable” objects.

Introduction

Zend\Mime\Mime is a support class for handling multipart *MIME* messages. It is used by *ZendMail* and *Zend\Mime\Message* and may be used by applications requiring *MIME* support.

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'

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.

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*.

Instantiation

There is no explicit constructor for `Zend\Mime\Message`.

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.

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. `setMessage(Zend\Mime\Mime $mime)` sets a special instance of `Zend\Mime\Mime` to be used by this `Zend\Mime\Message`.

`getMessage()` 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.

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()`

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.
- `setMessage(Zend\Mime\Mime $mime)`: Set a custom `Zend\Mime\Mime` object for the message.
- `getMessage`: Get the `Zend\Mime\Mime` object for the message.
- `generateMessage($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.

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.

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:

```
1 public $type = Zend\Mime\Mime::TYPE_OCTETSTREAM;
2 public $encoding = Zend\Mime\Mime::ENCODING_8BIT;
3 public $id;
4 public $disposition;
5 public $filename;
6 public $description;
7 public $charset;
8 public $boundary;
9 public $location;
10 public $language;
```

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 `Zend\Mime\Mime::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.

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.

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>
```

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).

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.

Module Manager Events

The Module Manager events are defined in `Zend\ModuleManager\ModuleEvent`.

Events triggered by `Zend\ModuleManager\ModuleManager`

loadModules (`ModuleEvent::EVENT_LOAD_MODULES`) 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 (`ModuleEvent::EVENT_LOAD_MODULE_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 (`ModuleEvent::EVENT_LOAD_MODULE`) 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.

mergeConfig (`ModuleEvent::EVENT_MERGE_CONFIG`) After all modules have been loaded, the `mergeConfig` event is triggered. By default, `Zend\ModuleManager\Listener\ConfigListener`

listens on this event at priority 1000, and merges all configuration. You may attach additional listeners to this event in order to manipulate the merged configuration. See [the tutorial on manipulating merged configuration](#) for more information.

loadModules.post (ModuleEvent::EVENT_LOAD_MODULES_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.

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 `getModuleDependencies()` method, the listener will call `getModuleDependencies()`. 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 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\ServiceManagerInterface, 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\Mvc\Controller\ControllerManager	controllers	ControllerProviderInterface	getControllerConfig
Zend\Mvc\Controller\PluginManager	controller_plugins	ControllerPluginProviderInterface	getControllerPluginConfig
Zend\Filter\FilterPluginManager	filters	FilterProviderInterface	getFilterConfig
Zend\Form\FormElementManager	form_element	FormElementProviderInterface	getFormElementConfig
Zend\Stdlib\Hydrator\HydratorPluginManager	hydrator_plugins	HydratorProviderInterface	getHydratorConfig
Zend\InputFilter\InputFilterPluginManager	input_filters	InputFilterProviderInterface	getInputFilterConfig
Zend\Mvc\Router\RoutePluginManager	routes	RouteProviderInterface	getRouteConfig
Zend\Serializer\AdapterPluginManager	serializers	SerializerProviderInterface	getSerializerConfig
Zend\ServiceManager\ServiceManager	service_manager	ServiceProviderInterface	getServiceConfig
Zend\Validator\ValidatorPluginManager	validators	ValidatorProviderInterface	getValidatorConfig
Zend\View\HelperPluginManager	view_helpers	ViewHelperProviderInterface	getViewHelperConfig
Zend\Log\ProcessorPluginManager	log_processors	LogProcessorProviderInterface	getLogProcessorConfig
Zend\Log\WriterPluginManager	log_writers	LogWriterProviderInterface	getLogWriterConfig

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:

```

1 public function getViewHelperConfig()
2 {
3     return array('factories' => array(
4         'foo' => function ($helpers) {
5             $services = $helpers->getServiceLocator();
6             $someService = $services->get('SomeService');
7             $helper = new Helper\Foo($someService);
8
9             return $helper;
10        },
11    ));
12 }
```

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.

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.

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:

```
1 namespace MyModule;  
2  
3 class Module  
4 {  
5 }
```

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.

A Typical Module Class

The following example shows a more typical usage of the Module class:

```
1 namespace MyModule;
2
3 class Module
4 {
5     public function getAutoloaderConfig()
6     {
7         return array(
8             'Zend\Loader\ClassMapAutoloader' => array(
9                 __DIR__ . '/autoload_classmap.php',
10            ),
11            'Zend\Loader\StandardAutoloader' => array(
12                'namespaces' => array(
13                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
14                ),
15            ),
16        );
17    }
18
19    public function getConfig()
20    {
21        return include __DIR__ . '/config/module.config.php';
22    }
23 }
```

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 [module manager listeners](#) and the [module manager events](#) documentations.

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](#).

Sample Usage of “loadModules.post” Event

```
1 use Zend\EventManager\EventInterface as Event;
2 use Zend\ModuleManager\ModuleManager;
3
4 class Module
5 {
6     public function init(ModuleManager $moduleManager)
7     {
8         // Remember to keep the init() method as lightweight as possible
9         $events = $moduleManager->getEventManager();
```



```

10     $events->attach('loadModules.post', array($this, 'modulesLoaded'));
11 }
12
13 public function modulesLoaded(Event $e)
14 {
15     // This method is called once all modules are loaded.
16     $moduleManager = $e->getTarget();
17     $loadedModules = $moduleManager->getLoadedModules();
18     // To get the configuration from another module named 'FooModule'
19     $config = $moduleManager->getModule('FooModule')->getConfig();
20 }
21 }

```

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.

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.

Sample Usage of the MVC “bootstrap” Event

```

1 use Zend\EventManager\EventInterface as Event;
2
3 class Module
4 {
5     public function onBootstrap(Event $e)
6     {
7         // This method is called once the MVC bootstrapping is complete
8         $application = $e->getApplication();
9         $services     = $application->getServiceManager();
10    }
11 }

```

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.

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.

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.

```
1 // public/index.php
2 use Zend\ModuleManager\Listener;
3 use Zend\ModuleManager\ModuleManager;
4
5 chdir(dirname(__DIR__));
6
7 // Instantiate and configure the default listener aggregate
8 $listenerOptions = new Listener\ListenerOptions(array(
9     'module_paths' => array(
10         './module',
11         './vendor',
```

```
12     '/usr/share/zfmodules',
13 )
14 ));
15 $defaultListeners = new Listener\DefaultListenerAggregate($listenerOptions);
16
17 // Instantiate the module manager
18 $moduleManager = new ModuleManager(array(
19     'Application',
20     'FooModule',
21     'BarModule',
22 ));
23
24 // Attach the default listener aggregate and load the modules
25 $moduleManager->getEventManager()->attachAggregate($defaultListeners);
26 $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.

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.

```
1 // ./public/index.php
2 use Zend\Loader\ModuleAutoloader;
3 use Zend\ModuleManager\Listener;
4 use Zend\ModuleManager\ModuleManager;
5
6 chdir(dirname(__DIR__));
7
8 // Instantiate and configure the default listener aggregate
9 $listenerOptions = new Listener\ListenerOptions(array(
10     'module_paths' => array(
11         './module',
12         './vendor',
13         '/usr/share/zfmodules',
14         'MyModule' => '/path/to/mymoduledir-v1.2',
15     )
16 ));
17 $defaultListeners = new Listener\DefaultListenerAggregate($listenerOptions);
18
19 /**
20  * Without DefaultListenerAggregate:
21  *
22  * $moduleAutoloader = new ModuleAutoloader(array(
```

```
23 *     './module',
24 *     './vendor',
25 *     '/usr/share/zfmodules',
26 *     'MyModule' => '/path/to/mymoduledir-v1.2',
27 * );
28 * $moduleAutoloader->register();
29 *
30 */
31
32 // Instantiate the module manager
33 $moduleManager = new ModuleManager(array(
34     'MyModule',
35     'FooModule',
36     'BarModule',
37 ));
38
39 // Attach the default listener aggregate and load the modules
40 $moduleManager->getEventManager()->attachAggregate($defaultListeners);
41 $moduleManager->loadModules();
```

This same method works if you provide the path to a phar archive.

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.

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.

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.

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.

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.

```

1 namespace ZendUser;
2
3 class Module
4 {
5 }

```

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 213.1: `autoload_*.php` Files

File	Description
<code>autoload_classmap.php</code>	Should return an array classmap of class name/filename pairs (with the filenames resolved via the <code>__DIR__</code> magic constant).
<code>autoload_function.php</code>	Should return a PHP callback that can be passed to <code>spl_autoload_register()</code> . Typically, this callback should utilize the map returned by <code>autoload_classmap.php</code> .
<code>autoload_register.php</code>	Should register a PHP callback (is typically returned by <code>autoload_function.php</code> with <code>spl_autoload_register()</code>).

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.

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:

```

1 use Zend\EventManager\EventManager;
2 use Zend\Http\PhpEnvironment;
3 use Zend\ModuleManager\ModuleManager;
4 use Zend\Mvc\Application;
5 use Zend\ServiceManager\ServiceManager;
6
7 $config = include 'config/application.config.php';
8
9 $serviceManager = new ServiceManager();
10 $serviceManager->setService('EventManager', new EventManager());
11 $serviceManager->setService('ModuleManager', new ModuleManager($config));
12 $serviceManager->setService('Request', new PhpEnvironment\Request());
13 $serviceManager->setService('Response', new PhpEnvironment\Response());
14
15 $application = new Application($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.

```

1 use Zend\Loader\AutoloaderFactory;
2 use Zend\Mvc\Service\ServiceManagerConfig;
3 use Zend\ServiceManager\ServiceManager;
4
5 // setup autoloader
6 AutoloaderFactory::factory();
7
8 // get application stack configuration
9 $configuration = include 'config/application.config.php';
10
11 // setup service manager
12 $serviceManager = new ServiceManager(new ServiceManagerConfig());
13 $serviceManager->setService('ApplicationConfig', $configuration);
14
15 // load modules -- which will provide services, configuration, and more
16 $serviceManager->get('ModuleManager')->loadModules();
17
18 // bootstrap and run application
19 $application = $serviceManager->get('Application');
20 $application->bootstrap();
21 $application->run();

```

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`.

```

1 use Zend\Loader\AutoloaderFactory;
2 use Zend\Mvc\Application;
3 use Zend\Mvc\Service\ServiceManagerConfig;
4 use Zend\ServiceManager\ServiceManager;
5
6 // setup autoloader
7 AutoloaderFactory::factory();
8
9 // get application stack configuration
10 $configuration = include 'config/application.config.php';
11
12 // The init() method does something very similar with the previous example.
13 Application::init($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.

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.

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.

```
1 <?php
2 // config/application.config.php
3 return array(
4     'modules' => array(
5         /* ... */
6     ),
7     'module_listener_options' => array(
8         'module_paths' => array(
9             './module',
10            './vendor',
11        ),
12    ),
13 );
```

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:

```
1 namespace ZendUser;
2
3 class Module
4 {
5     public function getConfig()
6     {
7         return include __DIR__ . '/config/module.config.php'
8     }
9 }
```

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.

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.

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.

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.

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
```

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-application
```

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.

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.

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:

```
1 <?php
2 // autoload_classmap.php
3 return array();
```

We’ll also edit our `config/module.config.php` file to read as follows:

```
1 return array(
2     'view_manager' => array(
3         'template_path_stack' => array(
4             '<module-name>' => __DIR__ . '/../view'
5         ),
6     ),
7 );
```

```

6     ),
7 );

```

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:

```

1 namespace <your module name here>;
2
3 use Zend\ModuleManager\Feature\AutoloaderProviderInterface;
4 use Zend\ModuleManager\Feature\ConfigProviderInterface;
5
6 class Module implements AutoloaderProviderInterface, ConfigProviderInterface
7 {
8     public function getAutoloaderConfig()
9     {
10         return array(
11             'Zend\Loader\ClassMapAutoloader' => array(
12                 __DIR__ . '/autoload_classmap.php',
13             ),
14             'Zend\Loader\StandardAutoloader' => array(
15                 'namespaces' => array(
16                     __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
17                 ),
18             ),
19         );
20     }
21
22     public function getConfig()
23     {
24         return include __DIR__ . '/config/module.config.php';
25     }
26 }

```

At this point, you now have your module configured properly. Let’s create a controller!

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 identifier 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:

```
1 <?php
2 namespace <module name>\Controller;
3
4 use Zend\Mvc\Controller\AbstractActionController;
5 use Zend\View\Model\ViewModel;
6
7 class HelloController extends AbstractActionController
8 {
9     public function worldAction()
10     {
11         $message = $this->params()->fromQuery('message', 'foo');
12         return new ViewModel(array('message' => $message));
13     }
14 }
```

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.

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:

```
1 <h1>Greetings!</h1>
2
3 <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.

View scripts for module names with subnamespaces

As per PSR-0, module should be named following this rule: \<Vendor Name>\(<Namespace>\)* Default controller class to template mapping does not work very well with those: it will remove subnamespace.

To address that issue new mapping was introduced since 2.3.0. To maintain backwards compatibility that mapping is not enabled by default. To enable it, you need to add your module namespace to whitelist in your module config:

```
1 'view_manager' => array(
2     // Controller namespace to template map
3     // or whitelisting for controller FQCN to template mapping
4     'controller_map' => array(
5         '<Module\Name>' => true,
6     ),
7 ),
```

Now, create the directory view/<module>/<name>/hello. Inside that directory, create a file named world.phtml. Inside that, paste in the following:

```
1 <h1>Greetings!</h1>
2
3 <p>You said "<?php echo $this->escapeHtml($message) ?>".</p>
```

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.

```
1 // module.config.php
2 return array(
3     '<module-name>' => array(
4         'type' => 'Literal',
5         'options' => array(
6             'route' => '/<module-name>',
7             'defaults' => array(
8                 'controller' => '<module-namespace>\Controller\Index',
9                 'action' => 'index',
10            ),
11        ),
12        'may_terminate' => true,
13        'child_routes' => array(
14            'default' => array(
15                'type' => 'Segment',
16                'options' => array(
17                    'route' => '/[:controller[:action]]',
18                    'constraints' => array(
```

```
19         'controller' => '[a-zA-Z][a-zA-Z0-9_]*',
20         'action'      => '[a-zA-Z][a-zA-Z0-9_]*',
21     ),
22     'defaults' => array(
23     ),
24 ),
25 ),
26 ),
27 ),
28 // ... other configuration ...
29 );
```

Additionally, we need to tell the application we have a controller:

```
1 // module.config.php
2 return array(
3     'controllers' => array(
4         'invokables' => array(
5             '<module-namespace>\Controller\Index' => '<module-namespace>'
↪ '\Controller\IndexController',
6             // Do similar for each other controller in your module
7         ),
8     ),
9     // ... other configuration ...
10 );
```

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:

```
1 return array(
2     'router' => array(
3         'routes' => array(
4             '<module name>-hello-world' => array(
5                 'type' => 'Literal',
6                 'options' => array(
7                     'route' => '/hello/world',
8                     'defaults' => array(
9                         'controller' => '<module name>\Controller\Hello',
10                        'action' => 'world',
11                    ),
12                ),
13            ),
14        ),
15    ),
16    'controllers' => array(
17        'invokables' => array(
18            '<module namespace>\Controller\Hello' => '<module namespace>'
↪ '\Controller\HelloController',
19        ),
20    ),
```

```

21     // ... other configuration ...
22 );

```

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:

```

1 <?php
2 return array(
3     'modules' => array(
4         'Application',
5         '<module namespace>',
6     ),
7     'module_listener_options' => array(
8         'module_paths' => array(
9             './module',
10            './vendor',
11        ),
12    ),
13 );

```

Replace `<module namespace>` with the namespace of your module.

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:

```

1 <h1>Greetings!</h1>
2
3 <p>You said "foo".</p>

```

Now alter the location to append `"?message=bar"` and load the page. You should now get:

```

1 <h1>Greetings!</h1>
2
3 <p>You said "bar".</p>

```

Congratulations! You've created your first ZF2 MVC module!

CHAPTER 215

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.

Many of the services are provided by other components, and the factories and abstract factories themselves are defined in the individual components. We will cover those factories in this chapter, however, as usage is generally the same between each.

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\ServiceListenerFactory`. This approach allows us to keep the main application configuration concise, 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.

ServiceManager

As a quick review, the following service types may be configured:

- **Invokable services**, which map a service name to a class that has no constructor or a constructor that accepts no arguments.
- **Factories**, which map a service name to a factory which will create and return an object. A factory receives the service manager as an argument, and may be any PHP callable, or a class or object that implements `Zend\ServiceManager\FactoryInterface`.
- **Abstract factories**, which are factories that can create any number of named services that share the same instantiation pattern; examples include database adapters, cache adapters, loggers, etc. The factory receives the service manager as an argument, the resolved service name, and the requested service name; it **must** be a class or object implementing `Zend\ServiceManager\AbstractFactoryInterface`. See the [section on abstract factories](#) for configuration information.
- **Aliases**, which alias one service name to another. Aliases can also reference other aliases.
- **Initializers**, which receive the newly created instance and the service manager, and which can be used to perform additional initialization tasks. The most common use case is to test the instance against specific “Aware” interfaces, and, if matching, inject them with the appropriate service.
- **Plugin managers**, which are specialized service managers used to manage objects that are of a related type, such as view helpers, controller plugins, controllers, etc. Plugin managers accept configuration just like service managers, and as such can compose invokable services, factories, abstract factories, aliases, and initializers. They are also `ServiceLocatorAware`, and will be injected with the application service manager instance, giving factories and abstract factories access to application-level services when needed. See the heading [Plugin managers](#) for a list of available plugin managers.

The application service manager is referenced directly during bootstrapping, and has the following services configured out of the box.

- **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.
 - `ControllerManager`, 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.
 - `ConsoleViewManager`, mapping to `Zend\Mvc\Service ConsoleViewManagerFactory`. This creates and returns an instance of `Zend\Mvc View Console ViewManager`, which in turn registers and initializes a number of console-specific view services.
 - `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 DiStrictAbstractServiceFactoryFactory`. This creates an instance of `Zend\Mvc Service DiStrictAbstractServiceFactoryFactory` injecting the `Di` service instance.
 - `EventManager`, mapping to `Zend\Mvc\Service EventManagerFactory`. This factory returns a new instance of `Zend\EventManager\EventManager` on each request. 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.

- `HttpViewModelManager`, mapping to `Zend\Mvc\Service\HttpViewModelManagerFactory`. This creates and returns an instance of `Zend\Mvc\View\Http\ViewModelManager`, which in turn registers and initializes a number of HTTP-specific view services.
- `HydratorManager`, mapping to `Zend\Mvc\Service\HydratorManagerFactory`. This creates and returns an instance of `Zend\Stdlib\Hydrator\HydratorPluginManager`, which can be used to manage and persist hydrator instances.
- `InputFilterManager`, mapping to `Zend\Mvc\Service\InputFilterManagerFactory`. This creates and returns an instance of `Zend\InputFilter\InputFilterPluginManager`, which can be used to manage and persist input filter instances.
- `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`.

- `MvcTranslator`, mapping to `Zend\Mvc\Service\TranslatorServiceFactory`, and returning an instance of `Zend\Mvc\I18n\Translator`, which extends `Zend\I18n\Translator\Translator` and implements `Zend\Validator\Translator\TranslatorInterface`, allowing the instance to be used anywhere a translator may be required in the framework.
- `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 environment, 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.
- `SerializerAdapterManager`, mapping to `Zend\Mvc\Service\SerializerAdapterManagerFactory` which returns an instance of `Zend\Serializer\AdapterManager`. This is a plugin manager for managing serializer adapter instances.
- `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.
- `ViewFeedRenderer`, mapping to `Zend\Mvc\Service\ViewFeedRendererFactory`, which returns an instance of `Zend\View\Renderer\FeedRenderer`, used to render feeds.
- `ViewFeedStrategy`, mapping to `Zend\Mvc\Service\ViewFeedStrategyFactory`, which returns an instance of `Zend\View\Strategy\FeedStrategy`, used to select the `ViewFeedRenderer` given the appropriate criteria.
- `ViewHelperManager`, mapping to `Zend\Mvc\Service\ViewHelperManagerFactory`, which returns an instance of `Zend\View\HelperManager`. This is a plugin manager for managing view helper instances.
- `ViewJsonRenderer`, mapping to `Zend\Mvc\Service\ViewJsonRendererFactory`, which returns an instance of `Zend\View\Renderer\JsonRenderer`, used to render JSON structures.
- `ViewJsonStrategy`, mapping to `Zend\Mvc\Service\ViewJsonStrategyFactory`, which returns an instance of `Zend\View\Strategy\JsonStrategy`, used to select the `ViewJsonRenderer` given the appropriate criteria.
- `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`.

- **Abstract factories**

- `Zend\Cache\Service\StorageCacheAbstractServiceFactory` (opt-in; registered by default in the skeleton application).
- `Zend\Db\Adapter\AdapterAbstractServiceFactory` (opt-in).
- `Zend\Form\FormAbstractServiceFactory` is registered by default.
- `Zend\Log\LoggerAbstractServiceFactory` (opt-in; registered by default in the skeleton application).

- **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.

- **Initializers**

- 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.
- The `ServiceManager` registers itself as the `ServiceManager` service, and aliases itself to the class names `Zend\ServiceManager\ServiceLocatorInterface` and `Zend\ServiceManager\ServiceManager`.

Abstract Factories

As noted in the previous section, Zend Framework provides a number of abstract service factories by default. Each is noted below, along with sample configuration.

In each instance, the abstract factory looks for a top-level configuration key, consisting of key/value pairs where the key is the service name, and the value is the configuration to use to create the given service.

Zend\Cache\Service\StorageCacheAbstractServiceFactory

This abstract factory is opt-in, but registered by default in the skeleton application. It uses the top-level configuration key “caches”.

```

1  return array(
2      'caches' => array(
3          'Cache\Transient' => array(
4              'adapter' => 'redis',
5              'ttl'      => 60,
6              'plugins' => array(
7                  'exception_handler' => array(
8                      'throw_exceptions' => false,
9                  ),
10             ),
11         ),
12         'Cache\Persistence' => array(
13             'adapter' => 'filesystem',
14             'ttl'      => 86400,
15         ),
16     ),
17 );

```

See the [cache documentation](#) for more configuration options.

Zend\Db\Adapter\AdapterAbstractServiceFactory

This abstract factory is opt-in. It uses the top-level configuration key “db”, with a subkey “adapters”.

```

1  return array(
2      'db' => array('adapters' => array(
3          'Db\ReadOnly' => array(
4              'driver'      => 'Pdo_Sqlite',
5              'database'    => 'data/db/users.db',
6          ),
7          'Db\Writeable' => array(
8              'driver'      => 'Mysqli',
9              'database'    => 'users',
10             'username'    => 'developer',
11             'password'    => 'developer_password',
12         ),
13     )),
14 );

```

See the [DB adapter documentation](#) for more configuration options.

Zend\Form\FormAbstractServiceFactory

This abstract factory is registered by default. It uses the top-level configuration key “forms”. It makes use of the `FilterManager`, `FormElementManager`, `HydratorManager`, `InputFilterManager`, and `ValidatorManager` plugin managers in order to allow instantiation and creation of form objects and all related objects in the form hierarchy.

```

1  return array(
2      'forms' => array(

```

```
3         'Form\Foo' => array(
4             'hydrator' => 'ObjectProperty',
5             'type'      => 'Zend\Form\Form',
6             'elements' => array(
7                 array(
8                     'spec' => array(
9                         'type' => 'Zend\Form\Element\Email',
10                        'name' => 'email',
11                        'options' => array(
12                            'label' => 'Your email address',
13                        ),
14                    ),
15                ),
16            ),
17        ),
18    ),
19 );
```

Form configuration follows the same configuration you would use with a form factory; the primary difference is that all plugin managers have already been injected for you, allowing you the possibility of custom objects or substitutions.

See the [form factory documentation](#) for more configuration options.

Zend\Log\LoggerAbstractServiceFactory

This abstract factory is opt-in, but registered by default in the skeleton application. It uses the top-level configuration key “log”.

```
1 return array(
2     'log' => array(
3         'Log\App' => array(
4             'writers' => array(
5                 array(
6                     'name' => 'stream',
7                     'priority' => 1000,
8                     'options' => array(
9                         'stream' => 'data/logs/app.log',
10                    ),
11                ),
12            ),
13        ),
14    ),
15 );
```

See the [log documentation](#) for more configuration options.

Plugin Managers

The following plugin managers are configured by default:

- **ControllerManager**, corresponding to `Zend\Mvc\Controller\ControllerManager`, and used to manage controller instances.
- **ControllerPluginManager**, corresponding to `Zend\Mvc\Controller\PluginManager`, and used to manage controller plugin instances.

- **FilterManager**, corresponding to `Zend\Filter\FilterPluginManager`, and used to manage filter instances.
- **FormElementManager**, corresponding to `Zend\Form\FormElementManager`, and used to manage instances of form elements and fieldsets.
- **HydratorManager**, corresponding to `Zend\Stdlib\Hydrator\HydratorPluginManager`, and used to manage hydrator instances.
- **InputFilterManager**, corresponding to `Zend\InputFilter\InputFilterPluginManager`, and used to manage input filter instances.
- **RoutePluginManager**, corresponding to `Zend\Mvc\Router\RoutePluginManager`, and used to manage route instances.
- **SerializerAdapterManager**, corresponding to `Zend\Serializer\AdapterPluginManager`, and used to manage serializer instances.
- **ValidatorManager**, corresponding to `Zend\Validator\ValidatorPluginManager`, and used to manage validator instances.
- **ViewHelperManager**, corresponding to `Zend\View\HelperPluginManager`, and used to manage view helper instances.

As noted in the previous section, all plugin managers share the same configuration and service types as the standard service manager; they are simply scoped, and only allow instances of certain types to be created or registered. Default types available are listed in the documentation for each component.

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 `base_path_console` key is present, it is used to inject the `BasePath` view helper for console requests; otherwise, the `Request` service is retrieved, and the value of its `getBasePath()` method is used. This can be useful for sending urls in emails via a cronjob.
 - 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.

- If a `default_template_suffix` key is present, it will be used as the default suffix for template scripts resolving.
- `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`).

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.

```

1 <?php
2 return array(
3     // This should be an array of module namespaces used in the application.
4     'modules' => array(
5     ),
6
7     // These are various options for the listeners attached to the ModuleManager
8     'module_listener_options' => array(

```

```

9      // This should be an array of paths in which modules reside.
10     // If a string key is provided, the listener will consider that a module
11     // namespace, the value of that key the specific path to that module's
12     // Module class.
13     'module_paths' => array(
14     ),
15
16     // An array of paths from which to glob configuration files after
17     // modules are loaded. These effectively override configuration
18     // provided by modules themselves. Paths may use GLOB_BRACE notation.
19     'config_glob_paths' => array(
20     ),
21
22     // Whether or not to enable a configuration cache.
23     // If enabled, the merged configuration will be cached and used in
24     // subsequent requests.
25     'config_cache_enabled' => $booleanValue,
26
27     // The key used to create the configuration cache file name.
28     'config_cache_key' => $stringKey,
29
30     // Whether or not to enable a module class map cache.
31     // If enabled, creates a module class map cache which will be used
32     // by in future requests, to reduce the autoloading process.
33     'module_map_cache_enabled' => $booleanValue,
34
35     // The key used to create the class map cache file name.
36     'module_map_cache_key' => $stringKey,
37
38     // The path in which to cache merged configuration.
39     'cache_dir' => $stringPath,
40
41     // Whether or not to enable modules dependency checking.
42     // Enabled by default, prevents usage of modules that depend on other modules
43     // that weren't loaded.
44     'check_dependencies' => $booleanValue,
45 ),
46
47 // Used to create an own service manager. May contain one or more child arrays.
48 'service_listener_options' => array(
49     array(
50         'service_manager' => $stringServiceManagerName,
51         'config_key'      => $stringConfigKey,
52         'interface'       => $stringOptionalInterface,
53         'method'          => $stringRequiredMethodName,
54     ),
55 )
56
57 // Initial configuration with which to seed the ServiceManager.
58 // Should be compatible with Zend\ServiceManager\Config.
59 'service_manager' => array(
60 ),
61 );

```

For an example, see the `ZendSkeletonApplication` configuration file.

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.php` files, or in the `module/<module name>/config/module.config.php` configuration files. The merging of these configuration 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.

```

1 <?php
2 return array(
3     // The following are used to configure controller loader
4     // Should be compatible with Zend\ServiceManager\Config.
5     'controllers' => array(
6         // Map of controller "name" to class
7         // This should be used if you do not need to inject any dependencies
8         // in your controller
9         'invokables' => array(
10             ),
11
12         // Map of controller "name" to factory for creating controller instance
13         // You may provide either the class name of a factory, or a PHP callback.
14         'factories' => array(
15             ),
16     ),
17
18     // The following are used to configure controller plugin loader
19     // Should be compatible with Zend\ServiceManager\Config.
20     'controller_plugins' => array(
21     ),
22
23     // The following are used to configure view helper manager
24     // Should be compatible with Zend\ServiceManager\Config.
25     'view_helpers' => array(
26     ),
27
28     // The following is used to configure a Zend\Di\Di instance.
29     // The array should be in a format that Zend\Di\Config can understand.
30     'di' => array(
31     ),
32
33     // Configuration for the Router service
34     // Can contain any router configuration, but typically will always define
35     // the routes for the application. See the router documentation for details
36     // on route configuration.
37     'router' => array(

```

```

38     'routes' => array(
39     ),
40 ),
41
42 // ViewManager configuration
43 'view_manager' => array(
44     // Base URL path to the application
45     'base_path' => $stringBasePath,
46
47     // Doctype with which to seed the Doctype helper
48     'doctype' => $doctypeHelperConstantString, // e.g. HTML5, XHTML1
49
50     // TemplateMapResolver configuration
51     // template/path pairs
52     'template_map' => array(
53     ),
54
55     // TemplatePathStack configuration
56     // module/view script path pairs
57     'template_path_stack' => array(
58     ),
59     // Default suffix to use when resolving template scripts, if none, 'phtml' is
→used
60     'default_template_suffix' => $templateSuffix, // e.g. 'php'
61
62     // Controller namespace to template map
63     // or whitelisting for controller FQCN to template mapping
64     'controller_map' => array(
65     ),
66
67     // Layout template name
68     'layout' => $layoutTemplateName, // e.g. 'layout/layout'
69
70     // ExceptionStrategy configuration
71     'display_exceptions' => $bool, // display exceptions in template
72     'exception_template' => $stringTemplateName, // e.g. 'error'
73
74     // RouteNotFoundStrategy configuration
75     'display_not_found_reason' => $bool, // display 404 reason in template
76     'not_found_template' => $stringTemplateName, // e.g. '404'
77
78     // Additional strategies to attach
79     // These should be class names or service names of View strategy classes
80     // that act as ListenerAggregates. They will be attached at priority 100,
81     // in the order registered.
82     'strategies' => array(
83         'ViewJsonStrategy', // register JSON renderer strategy
84         'ViewFeedStrategy', // register Feed renderer strategy
85     ),
86 ),
87 );

```

For an example, see the Application module configuration file in the *ZendSkeletonApplication*.

CHAPTER 216

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`:

```
1 namespace Zend\Mvc\Router;
2
3 use Zend\Stdlib\RequestInterface as Request;
4
5 interface RouteInterface
6 {
7     public static function factory(array $options = array());
8     public function match(Request $request);
9     public function assemble(array $params = array(), array $options = array());
10 }
```

A `Route` accepts a `Request`, and determines if it matches. If so, it returns a `RouteMatch` object:

```

1 namespace Zend\Mvc\Router;
2
3 class RouteMatch
4 {
5     public function __construct(array $params);
6     public function setMatchedRouteName($name);
7     public function getMatchedRouteName();
8     public function setParam($name, $value);
9     public function getParams();
10    public function getParam($name, $default = null);
11 }
    
```

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.

```

1 $id = $routeMatch->getParam('id', false);
2 if (!$id) {
3     throw new Exception('Required identifier is missing!');
4 }
5 $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:

```

1 namespace Zend\Mvc\Router;
2
3 interface RouteStackInterface extends RouteInterface
4 {
5     public function addRoute($name, $route, $priority = null);
6     public function addRoutes(array $routes);
7     public function removeRoute($name);
8     public function setRoutes(array $routes);
9 }
    
```

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().

```

1 // One at a time:
2 $route = Literal::factory(array(
3     'route' => '/foo',
4     'defaults' => array(
5         'controller' => 'foo-index',
6         'action' => 'index',
7     ),
8 ));
9 $router->addRoute('foo', $route);
10
11 // In bulk:
12 $router->addRoutes(array(
13     // using already instantiated routes:
14     'foo' => $route,
15
16     // providing configuration to allow lazy-loading routes:
17     'bar' => array(
18         'type' => 'literal',
19         'options' => array(
20             'route' => '/bar',
    
```



```

21         'defaults' => array(
22             'controller' => 'bar-index',
23             'action'      => 'index',
24         ),
25     ),
26 ),
27 );

```

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.

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.

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.

HTTP Route Types

Zend Framework 2.0 ships with the following HTTP route types.

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:

```
1 $route = Hostname::factory(array(  
2     'route' => ':subdomain.domain.tld',  
3     'constraints' => array(  
4         'subdomain' => 'fw\d{2}',  
5     ),  
6 ));
```

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.

```
1 $route = Hostname::factory(array(  
2     'route' => ':subdomain.domain.tld',  
3     'constraints' => array(  
4         'subdomain' => 'fw\d{2}',  
5     ),  
6     'defaults' => array(  
7         'type' => 'json',  
8     ),  
9 ));
```

When matched, the above will return two keys in the `RouteMatch`, “subdomain” and “type”.

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.

```
1 $route = Literal::factory(array(  
2     'route' => '/foo',  
3     'defaults' => array(  
4         'controller' => 'Application\Controller\IndexController',  
5         'action' => 'foo',  
6     ),  
7 ));
```

The above route would match a path “/foo”, and return the key “action” in the `RouteMatch`, with the value “foo”.

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.

```

1 $route = Method::factory(array(
2     'verb' => 'post,put',
3     'defaults' => array(
4         'controller' => 'Application\Controller\IndexController',
5         'action' => 'form-submit',
6     ),
7 ));

```

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”.

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.

```

1 $route = Part::factory(array(
2     'route' => array(
3         'type' => 'literal',
4         'options' => array(
5             'route' => '/',
6             'defaults' => array(
7                 'controller' => 'Application\Controller\IndexController',
8                 'action' => 'index',
9             ),
10        ),
11    ),
12    'route_plugins' => $routePlugins,
13    'may_terminate' => true,
14    'child_routes' => array(
15        'blog' => array(
16            'type' => 'literal',
17            'options' => array(
18                'route' => '/blog',
19                'defaults' => array(
20                    'controller' => 'Application\Controller\BlogController',
21                    'action' => 'index',
22                ),
23            ),
24            'may_terminate' => true,
25            'child_routes' => array(
26                'rss' => array(
27                    'type' => 'literal',
28                    'options' => array(
29                        'route' => '/rss',
30                        'defaults' => array(
31                            'action' => 'rss',
32                        ),
33                    ),
34                    'may_terminate' => true,
35                    'child_routes' => array(
36                        'subrss' => array(
37                            'type' => 'literal',
38                            'options' => array(
39                                'route' => '/sub',

```

```
40         'defaults' => array(
41             'action' => 'subrss',
42         ),
43     ),
44 ),
45 ),
46 ),
47 ),
48 ),
49 'forum' => array(
50     'type' => 'literal',
51     'options' => array(
52         'route' => 'forum',
53         'defaults' => array(
54             'controller' => 'Application\Controller\ForumController',
55             'action' => 'index',
56         ),
57     ),
58 ),
59 ),
60 );
```

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`.

```
1 $routePlugins = new Zend\Mvc\Router\RoutePluginManager();
2 $plugins = array(
3     'hostname' => 'Zend\Mvc\Router\Http\Hostname',
4     'literal' => 'Zend\Mvc\Router\Http\Literal',
5     'part' => 'Zend\Mvc\Router\Http\Part',
6     'regex' => 'Zend\Mvc\Router\Http\Regex',
7     'scheme' => 'Zend\Mvc\Router\Http\Scheme',
8     'segment' => 'Zend\Mvc\Router\Http\Segment',
9     'wildcard' => 'Zend\Mvc\Router\Http\Wildcard',
10    'query' => 'Zend\Mvc\Router\Http\Query',
11    'method' => 'Zend\Mvc\Router\Http\Method',
12 );
13 foreach ($plugins as $name => $class) {
```

```

14     $routePlugins->setInvokableClass($name, $class);
15 }
    
```

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.

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.

```

1  $route = Regex::factory(array(
2      'regex' => '/blog/(?<id>[a-zA-Z0-9_-]+) (\.(?<format>(json|html|xml|rss)))?/',
3      'defaults' => array(
4          'controller' => 'Application\Controller\BlogController',
5          'action'      => 'view',
6          'format'      => 'html',
7      ),
8      'spec' => '/blog/%id%.%format%',
9  ));
    
```

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.

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.

```

1  $route = Scheme::factory(array(
2      'scheme' => 'https',
3      'defaults' => array(
4          'https' => true,
5      ),
6  ));
    
```

The above route would match the “https” scheme, and return the key “https” in the `RouteMatch` with a boolean `true` value.

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:

```
1 $route = Segment::factory(array(  
2     'route' =>('/:controller[:action]'),  
3     'constraints' => array(  
4         'controller' => '[a-zA-Z][a-zA-Z0-9_-]+',  
5         'action' => '[a-zA-Z][a-zA-Z0-9_-]+',  
6     ),  
7     'defaults' => array(  
8         'controller' => 'Application\Controller\IndexController',  
9         'action' => 'index',  
10    ),  
11 ));
```

Zend\Mvc\Router\Http\Query (Deprecated)

Warning: Potential security issue

A misuse of this route part can lead to a potential security issue.

Note: 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

```
1 $route = Part::factory(array(  
2     'route' => array(  
3         'type' => 'literal',  
4         'options' => array(  
5             'route' => 'page',  
6             'defaults' => array(  
7                 ),  
8         ),  
9     ),  
10    'may_terminate' => true,  
11    'route_plugins' => $routePlugins,  
12    'child_routes' => array(  
13        'query' => array(  
14            'type' => 'Query',
```

```

15         'options' => array(
16             'defaults' => array(
17                 'foo' => 'bar',
18             ),
19         ),
20     ),
21 ),
22 );

```

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.

```

1 $this->url(
2     'page/query',
3     array(
4         'name'    => 'my-test-page',
5         'format'  => 'rss',
6         'limit'   => 10,
7     )
8 );

```

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”

Zend\Mvc\Router\Http\Wildcard (Deprecated)

Warning: Potential security issue

A misuse of this route type can lead to a potential security issue.

Note: Deprecated

This route type is deprecated. Use the `Segment` route type.

The `Wildcard` route type matches all segments of a URI path, like in version 1 of Zend Framework.

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

```
1 return array(  
2     'router' => array(  
3         'routes' => array(  
4             // Literal route named "home"  
5             'home' => array(  
6                 'type' => 'literal',  
7                 'options' => array(  
8                     'route' => '/',  
9                     'defaults' => array(  
10                        'controller' => 'Application\Controller\IndexController',  
11                        'action' => 'index',  
12                    ),  
13                ),  
14            ),  
15            // Literal route named "contact"  
16            'contact' => array(  
17                'type' => 'literal',  
18                'options' => array(  
19                    'route' => 'contact',  
20                    'defaults' => array(  
21                        'controller' => 'Application\Controller>ContactController',  
22                        'action' => 'form',  
23                    ),  
24                ),  
25            ),  
26        ),  
27    ),  
28 );
```

A complex example with child routes

```
1 return array(  
2     'router' => array(  
3         'routes' => array(  
4             // Literal route named "home"  
5             'home' => array(  
6                 'type' => 'literal',  
7                 'options' => array(  
8                     'route' => '/',  
9                     'defaults' => array(  
10                        'controller' => 'Application\Controller\IndexController',  
11                        'action' => 'index',  
12                    ),  
13                ),  
14            ),  
15            // Literal route named "blog", with child routes  
16            'blog' => array(  
17                'type' => 'literal',  
18                'options' => array(  
19                    'route' => '/blog',  
20                    'defaults' => array(  
21                        'controller' => 'Application\Controller\BlogController',  
22                        'action' => 'index',  
23                    ),  
24                ),  
25                'may_terminate' => true,
```



```

26         'child_routes' => array(
27             // Segment route for viewing one blog post
28             'post' => array(
29                 'type' => 'segment',
30                 'options' => array(
31                     'route' => '[:slug]',
32                     'constraints' => array(
33                         'slug' => '[a-zA-Z0-9_-]+',
34                     ),
35                     'defaults' => array(
36                         'action' => 'view',
37                     ),
38                 ),
39             ),
40             // Literal route for viewing blog RSS feed
41             'rss' => array(
42                 'type' => 'literal',
43                 'options' => array(
44                     'route' => '/rss',
45                     'defaults' => array(
46                         'action' => 'rss',
47                     ),
48                 ),
49             ),
50         ),
51     ),
52 ),
53 );
54

```

When using child routes, naming of the routes follows the parent/child pattern, so to use the child routes from the above example:

```

1 echo $this->url('blog'); // gives "/blog"
2 echo $this->url('blog/post', array('slug' => 'my-post')); // gives "/blog/my-post"
3 echo $this->url('blog/rss'); // gives "/blog/rss"

```

An example with multiple Hostnames and subdomains within a single application

```

1 return array(
2     'router' => array(
3         'routes' => array(
4             'modules.zendframework.com' => array(
5                 'type' => 'Zend\Mvc\Router\Http\Hostname',
6                 'options' => array(
7                     'route' => ':4th.[:3rd.]:2nd.:1st', // domain levels from right_
8                     // to left
9                     'constraints' => array(
10                        '4th' => 'modules',
11                        '3rd' => '.*?', // optional 3rd level domain such as .ci, .
12                        // dev or .test
13                        '2nd' => 'zendframework',
14                        '1st' => 'com',
15                    ),
16                    // Purposely omit default controller and action
17                    // to let the child routes control the route match
18                ),
19            ),
20        ),
21    ),
22 );

```

```

16         ),
17         // child route controllers may span multiple modules as desired
18         'child_routes' => array(
19             'index' => array(
20                 'type' => 'Zend\Mvc\Router\Http\Literal',
21                 'options' => array(
22                     'route' => '/',
23                     'defaults' => array(
24                         'controller' => 'Module\Controller\Index',
25                         'action' => 'index',
26                     ),
27                 ),
28                 'may_terminate' => true,
29             ),
30         ),
31     ),
32     'packages.zendframework.com' => array(
33         'type' => 'Zend\Mvc\Router\Http\Hostname',
34         'options' => array(
35             'route' => ':4th.[:3rd.]:2nd.:1st', // domain levels from right_
36             // to left
37             'constraints' => array(
38                 '4th' => 'packages',
39                 '3rd' => '.*?', // optional 3rd level domain such as .ci, .
40                 // dev or .test
41                 '2nd' => 'zendframework',
42                 '1st' => 'com',
43             ),
44             // Purposely omit default controller and action
45             // to let the child routes control the route match
46         ),
47         // child route controllers may span multiple modules as desired
48         'child_routes' => array(
49             'index' => array(
50                 'type' => 'Zend\Mvc\Router\Http\Literal',
51                 'options' => array(
52                     'route' => '/',
53                     'defaults' => array(
54                         'controller' => 'Package\Controller\Index',
55                         'action' => 'index',
56                     ),
57                 ),
58                 'may_terminate' => true,
59             ),
60         ),
61     ),
62 );
    
```

The above would match the following:

- `modules.zendframework.com` would dispatch the Index controller's index action of the Module module.
- `modules.ci.zendframework.com` would dispatch the Index controller's index action of the Module module.
- `packages.zendframework.com` would dispatch the Index controller's index action of the Package

module.

- `packages.dev.zendframework.com` would dispatch the Index controller's index action of the Package module.

The `Url` controller plugin or view helper may be used to generate URLs following the above example:

```
1 // reuse the route matched parameters to generate URLs
2 echo $this->url('modules.zendframework.com/index', array(), array(), true);
3 echo $this->url('packages.zendframework.com/index', array(), array(), true);
```

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.

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.

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.

Order of events

The following events are triggered, in the following order:

Table 217.1: 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.

MvcEvent::EVENT_BOOTSTRAP

Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Table 217.2: `MvcEvent::EVENT_BOOTSTRAP` Listeners

Class	Priority	Method Called	Itself Triggers	Description
Zend\Mvc\View\Http\ViewManager	1000	bootstrap	<code>name</code>	Prepares the view layer (instantiate a Zend\Mvc\View\Http\ViewManager).

Triggerers

This event is triggered by the following classes:

Table 217.3: `MvcEvent::EVENT_BOOTSTRAP` Triggerers

Class	In Method
Zend\Mvc\Application	<code>bootstrap</code>

MvcEvent::EVENT_ROUTE

Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Table 217.4: `MvcEvent::EVENT_ROUTE` Listeners

Class	Priority	Method Called	Itself Triggers	Description
Zend\Mvc\ModuleRouteListener		<code>onRoute</code>	<code>name</code>	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 <code>MODULE_NAMESPACE</code> constant.
Zend\Mvc\RouteListener		<code>onRoute</code>	<code>MvcEvent::EVENT_DISPATCH</code> (if no route is matched)	Try to match the request to the router and return a <code>RouteMatch</code> object.

Triggerers

This event is triggered by the following classes:

Table 217.5: `MvcEvent::EVENT_ROUTE` Triggerers

Class	In Method	Description
Zend\Mvc\Application	<code>dispatch</code>	It also has a short circuit callback that allows to stop the propagation of the event if an error is raised during the routing.

MvcEvent::EVENT_DISPATCH

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 217.6: MvcEvent::EVENT_DISPATCH Listeners for Console context only

Class	Pri- or- ity	Method Called	Description
Zend\Mvc\View\Console\InjectParams	100	injectParams	Merge all the params (route matched params and params in the command) and add them to the Request object.
Zend\Mvc\View\Console\CreateViewModel	80	createViewModel	If the controller action returned an associative array, it casts it to a ConsoleModel object.
Zend\Mvc\View\Console\CreateViewModel	80	createViewModel	If the controller action returned a string, it casts it to a ConsoleModel object.
Zend\Mvc\View\Console\CreateViewModel	80	createViewModel	If the controller action returned null, it casts it to a ConsoleModel object.
Zend\Mvc\View\Console\InjectViewModel	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 217.7: MvcEvent::EVENT_DISPATCH Listeners for Http context only

Class	Pri- or- ity	Method Called	Description
Zend\Mvc\View\Http\CreateViewModel	80	createViewModel	If the controller action returned an associative array, it casts it to a ViewModel object.
Zend\Mvc\View\Http\CreateViewModel	80	createViewModel	If the controller action returned null, it casts it to a ViewModel object.
Zend\Mvc\View\Http\RenderNotFound	90	renderNotFound	Throws an exception and return a 404 ViewModel.
Zend\Mvc\View\Http\InjectTemplate	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\InjectViewModel	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 217.8: `MvcEvent::EVENT_DISPATCH` Listeners for both contexts

Class	Pri- or- ity	Method Called	Itself Triggers	Description
<code>Zend\Mvc\DispatchEvent</code>		<code>onDispatch</code>	<code>MvcEvent::EVENT_DISPATCH</code> (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).
<code>Zend\Mvc\AbstractController</code>		<code>onDispatch</code>		The <code>onDispatch</code> method of the <code>AbstractController</code> is an abstract method. In <code>AbstractActionController</code> for instance, it simply calls the action method.

Triggerers

This event is triggered by the following classes:

Table 217.9: `MvcEvent::EVENT_DISPATCH` Triggerers

Class	In Method	Description
<code>Zend\Mvc\Application</code>	<code>run</code>	It also has a short circuit callback that allows to stop the propagation of the event if an error is raised during the routing.
<code>Zend\Mvc\Controller\AbstractController</code>	<code>dispatch</code>	If a listener returns a <code>Response</code> object, it stops propagation. Note: every <code>AbstractController</code> listen to this event and execute the <code>onDispatch</code> method when it is triggered.

`MvcEvent::EVENT_DISPATCH_ERROR`

Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Console context only

Table 217.10: `MvcEvent::EVENT_DISPATCH_ERROR` Listeners for Console context only

Class	Pri- or- ity	Method Called	Description
<code>Zend\Mvc\View\Console\RouteNotFoundException</code>	1	<code>reportMonitorEvent</code>	Used to monitor when a route not found error is encountered, sets the response status code to 404.
<code>Zend\Mvc\View\Console\ExceptionEscaperStrategy</code>	1	<code>prepareException</code>	Creates an exception view model and set the status code to 404.
<code>Zend\Mvc\View\Console\InjectViewModel</code>	100	<code>injectViewModel</code>	Inserts the ViewModel (in this case, a <code>ConsoleModel</code>) and adds it to the <code>MvcEvent</code> 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 217.11: `MvcEvent::EVENT_DISPATCH_ERROR` Listeners for Http context only

Class	Pri- or- ity	Method Called	Description
<code>Zend\Mvc\View\Http\RouteNotFoundException</code>	1	<code>reportMonitorEvent</code>	Used to monitor when a 404 condition. If a “controller not found” or “invalid controller” error type is encountered, sets the response status code to 404.
<code>Zend\Mvc\View\Http\RouteNotFoundException</code>	1	<code>prepareResponse</code>	Creates and return a 404 view model.
<code>Zend\Mvc\View\Http\ExceptionEscaperStrategy</code>	1	<code>prepareException</code>	Creates an exception view model and set the status code to 404
<code>Zend\Mvc\View\Http\InjectViewModel</code>	100	<code>injectViewModel</code>	Inserts the ViewModel (in this case, a <code>ViewModel</code>) and adds it to the <code>MvcEvent</code> 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 217.12: `MvcEvent::EVENT_DISPATCH_ERROR` Listeners for both contexts

Class	Pri- or- ity	Method Called	Description
<code>Zend\Mvc\DispatchListener</code>	1	<code>reportMonitorEvent</code>	Used to monitoring when Zend Server is used.

Triggerers

Table 217.13: `MvcEvent::EVENT_DISPATCH_ERROR` Triggerers

Class	In Method
<code>Zend\Mvc\DispatchListener</code>	<code>onDispatch</code>
<code>Zend\Mvc\DispatchListener</code>	<code>marshallControllerNotFoundEvent</code>
<code>Zend\Mvc\DispatchListener</code>	<code>marshallBadControllerEvent</code>

MvcEvent::EVENT_RENDER

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 217.14: `MvcEvent::EVENT_RENDER` Listeners for Console context only

Class	Prior-ity	Method Called	Description
<code>Zend\Mvc\View\Console\DefaultRenderingStrategy</code>	-10000	<code>render</code>	Render the view.

Http context only

Those listeners are only attached in a Http context:

Table 217.15: `MvcEvent::EVENT_RENDER` Listeners for Http context only

Class	Prior-ity	Method Called	Description
<code>Zend\Mvc\View\Http\DefaultRenderingStrategy</code>	-10000	<code>render</code>	Render the view.

Triggerers

This event is triggered by the following classes:

Table 217.16: `MvcEvent::EVENT_RENDER` Triggerers

Class	In Method	Description
<code>Zend\Mvc\Application</code>	<code>completeRequest</code>	This event is triggered just before the <code>MvcEvent::FINISH</code> event.

MvcEvent::EVENT_RENDER_ERROR

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 217.17: `MvcEvent::EVENT_RENDER_ERROR` Listeners for Console context only

Class	Pri- or- ity	Method Called	Description
Zend\Mvc\View\ConsoleExceptionStrategy		ExceptionStrategy	Creates an exception view model and set the status code to 404.
Zend\Mvc\View\ConsoleInjectViewModel	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 217.18: `MvcEvent::EVENT_RENDER_ERROR` Listeners for Http context only

Class	Pri- or- ity	Method Called	Description
Zend\Mvc\View\ConsoleExceptionStrategy		ExceptionStrategy	Creates an exception view model and set the status code to 404.
Zend\Mvc\View\ConsoleInjectViewModel	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.

Triggerers

This event is triggered by the following classes:

Table 217.19: `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.

MvcEvent::EVENT_FINISH

Listeners

The following classes are listening to this event (they are sorted from higher priority to lower priority):

Table 217.20: MvcEvent::EVENT_FINISH Listeners

Class	Pri- or- ity	Method Called	Description
Zend\Mvc\SendResponseEvent	10000	sendResponse	Triggers the SendResponseEvent in order to prepare the response (see the next page for more information about SendResponseEvent).

Triggerers

This event is triggered by the following classes:

Table 217.21: 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).

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()`

Listeners

Currently, three listeners are listening to this event at different priorities based on which listener is used most.

Table 218.1: SendResponseEvent Listeners

Class	Pri- ority	Method Called	Description
<code>Zend\Mvc\SendResponseListener\PhpEnvironmentResponseSender</code>	1000	<code>sendResponse()</code>	This is used in context of HTTP (this is the most often used).
<code>Zend\Mvc\SendResponseListener\ConsoleResponseSender</code>	2000	<code>sendResponse()</code>	This is used in context of Console.
<code>Zend\Mvc\SendResponseListener\SimpleStreamResponseSender</code>	3000	<code>sendResponse()</code>	

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.

Triggerers

This event is executed when `MvcEvent::FINISH` event is triggered, with a priority of -10000.

CHAPTER 219

Available Controllers

Controllers in the MVC layer simply need to be objects implementing `Zend\Stdlib\DispatchableInterface`. That interface describes a single method:

```
1 use Zend\Stdlib\DispatchableInterface;
2 use Zend\Stdlib\RequestInterface as Request;
3 use Zend\Stdlib\ResponseInterface as Response;
4
5 class Foo implements DispatchableInterface
6 {
7     public function dispatch(Request $request, Response $response = null)
8     {
9         // ... do something, and preferably return a Response ...
10    }
11 }
```

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.

Common Interfaces Used With Controllers

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:

```
1 $matches = $this->getEvent()->getRouteMatch();
2 $id      = $matches->getParam('id', false);
3 if (!$id) {
4     $response = $this->getResponse();
5     $response->setStatusCode(500);
6     $this->getEvent()->setResult('Invalid identifier; cannot complete request');
7     return;
8 }
```

The `InjectApplicationEventInterface` defines simply two methods:

```
1 public function setEvent(Zend\EventManager\EventInterface $event);
2 public function getEvent();
```

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:

```
1 use Zend\ServiceManager\ServiceLocatorInterface;
2 use Zend\ServiceManager\ServiceLocatorAwareInterface;
3
4 public function setServiceLocator(ServiceLocatorInterface $serviceLocator);
5 public function getServiceLocator();
```

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.

```
1 use Zend\EventManager\EventManagerAwareInterface;
2 use Zend\EventManager\EventManagerInterface;
3
4 public function setEventManager(EventManagerInterface $events);
5 public function getEventManager();
```

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:

```

1  use Zend\Mvc\Controller\PluginManager;
2
3  public function setPluginManager(PluginManager $plugins)
4  {
5      $this->plugins = $plugins;
6      $this->plugins->setController($this);
7
8      return $this;
9  }
10
11 public function getPluginManager()
12 {
13     if (!$this->plugins) {
14         $this->setPluginManager(new PluginManager());
15     }
16
17     return $this->plugins;
18 }
19
20 public function plugin($name, array $options = null)
21 {
22     return $this->getPluginManager()->get($name, $options);
23 }

```

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:

```
1 namespace Foo\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4
5 classBarController extends AbstractActionController
6 {
7     public function bazAction()
8     {
9         return array('title' => __METHOD__);
10    }
11
12    public function batAction()
13    {
14        return array('title' => __METHOD__);
15    }
16 }
```

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.

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 `AbstractRestController`, 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.

Interfaces and Collaborators

`AbstractRestController` 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\AbstractRestController`
- `Zend\Mvc\Controller\AbstractController`

Additionally, if you extend the class, it will listen on the extending class’s name.

Controller Plugins

When using the `AbstractActionController` or `AbstractRestController`, 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:

```
1 $plugin = $this->plugin('url');
```

For an extra layer of convenience, both `AbstractActionController` and `AbstractRestController` have `__call()` implementations that allow you to retrieve plugins via method calls:

```
1 $plugin = $this->url();
```

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:

```
1 use Zend\Mvc\Controller\AbstractActionController;
2 use Zend\View\Model\JsonModel;
3
4 class SomeController extends AbstractActionController
5 {
6     protected $acceptCriteria = array(
7         'Zend\View\Model\JsonModel' => array(
8             'application/json',
9         ),
10        'Zend\View\Model\FeedModel' => array(
11            'application/rss+xml',
12        ),
13    );
14
15    public function apiAction()
16    {
17        $viewModel = $this->acceptableViewModelSelector($this->acceptCriteria);
18
19        // Potentially vary execution based on model returned
20        if ($viewModel instanceof JsonModel) {
21            // ...
22        }
23    }
24 }
```

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.”

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`

clearMessagesFromContainer()

Clear all messages from the container. Returns `true` if any messages were cleared, `false` if none existed.

Return type `boolean`

This plugin also provides four meaningful namespaces, namely: `INFO`, `ERROR`, `WARNING`, `SUCCESS`. The following functions are related to these namespaces:

addInfoMessage()

Add a message to “info” namespace

Return type `Zend\Mvc\Controller\Plugin\FlashMessenger`

hasCurrentInfoMessages()

Check to see if messages have been added to “info” namespace within this request

Return type `boolean`

addWarningMessage()

Add a message to “warning” namespace

Return type `Zend\Mvc\Controller\Plugin\FlashMessenger`

hasCurrentWarningMessages()

Check to see if messages have been added to “warning” namespace within this request

Return type `boolean`

addErrorMessage()

Add a message to “error” namespace

Return type `Zend\Mvc\Controller\Plugin\FlashMessenger`

hasCurrentErrorMessages()

Check to see if messages have been added to “error” namespace within this request

Return type `boolean`

addSuccessMessage()

Add a message to “success” namespace

Return type `Zend\Mvc\Controller\Plugin\FlashMessenger`

hasCurrentSuccessMessages()

Check to see if messages have been added to “success” namespace within this request

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

```

1 public function processAction()
2 {
3     // ... do some work ...
4     $this->flashMessenger()->addMessage('You are now logged in.');
```

```

5     return $this->redirect()->toRoute('user-success');
```

```

6 }
```

```

7
8 public function successAction()
```

```

9 {
```

```

10     $return = array('success' => true);
```

```

11     $flashMessenger = $this->flashMessenger();
```

```

12     if ($flashMessenger->hasMessages()) {
```

```

13         $return['messages'] = $flashMessenger->getMessages();
```

```

14     }
```

```

15     return $return;
```

```

16 }
```

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 seed a `RouteMatch` object for purposes of this specific request. Meaning the parameters will be matched by their key to the routing identifiers in the config (otherwise non-matching keys are ignored)

`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:

```
1 $foo = $this->forward()->dispatch('foo', array('action' => 'process'));
2 return array(
3     'somekey' => $somevalue,
4     'foo'     => $foo,
5 );
```

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:

```
1 public function testAction()
2 {
3     if ($user = $this->identity()) {
4         // someone is logged !
5     } else {
6         // not logged in
7     }
8 }
```

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:

```
1 // In a configuration file...
2 return array(
3     'service_manager' => array(
4         'aliases' => array(
```

```

5         'Zend\Authentication\AuthenticationService' => 'my_auth_service',
6     ),
7     'invokables' => array(
8         'my_auth_service' => 'Zend\Authentication\AuthenticationService',
9     ),
10 ),
11 );

```

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

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:

```

1 $this->layout()->setTemplate('layout/newlayout');

```

It also implements the `__invoke` magic method, which allows for even easier setting of the template:

```

1 $this->layout('layout/newlayout');

```

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:

```
1 $this->params()->fromRoute('param', $default);
2 // or
3 $this->params('param', $default);
```

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

```
1 // Pass in the route/url you want to redirect to after the POST
2 $prg = $this->prg('/user/register', true);
3
4 if ($prg instanceof \Zend\Http\PhpEnvironment\Response) {
5     // returned a response to redirect us
6     return $prg;
7 } elseif ($prg === false) {
8     // this wasn't a POST request, but there were no params in the flash messenger
9     // probably this is the first time the form was loaded
10    return array('form' => $myForm);
11 }
12
13 // $prg is an array containing the POST params from the previous request
14 $form->setData($prg);
15
16 // ... your form processing code here
```

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

```

1  $myForm = new Zend\Form\Form('my-form');
2  $myForm->add(array(
3      'type' => 'Zend\Form\Element\File',
4      'name' => 'file',
5  ));
6  // NOTE: Without a filter to move the file,
7  //         our files will disappear between the requests
8  $myForm->getInputFilter()->getFilterChain()->attach(
9      new Zend\Filter\File\RenameUpload(array(
10         'target'    => './data/tmpuploads/file',
11         'randomize' => true,
12     ))
13 );
14
15 // Pass in the form and optional the route/url you want to redirect to after the POST
16 $prg = $this->fileprg($myForm, '/user/profile-pic', true);
17
18 if ($prg instanceof \Zend\Http\PhpEnvironment\Response) {
19     // Returned a response to redirect us
20     return $prg;
21 } elseif ($prg === false) {
22     // First time the form was loaded
23     return array('form' => $myForm);
24 }
25
26 // Form was submitted.
27 // $prg is now an array containing the POST params from the previous request,
28 // but we don't have to apply it to the form since that has already been done.
```

```

29
30 // Process the form
31 if ($form->isValid()) {
32     // ...Save the form...
33     return $this->redirect()->toRoute('/user/profile-pic/success');
34 } else {
35     // Form not valid, but file uploads might be valid and uploaded
36     $fileErrors = $form->get('file')->getMessages();
37     if (empty($fileErrors)) {
38         $tempFile = $form->get('file')->getValue();
39     }
40 }

```

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:

```

1 return $this->redirect()->toRoute('login-success');

```

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:

```
1 $router = $this->getEvent()->getRouter();
2 $url    = $router->assemble($params, array('name' => 'route-name'));
```

The `Url` helper makes this slightly more convenient:

```
1 $url = $this->url()->fromRoute('route-name', $params);
```

The `fromRoute()` method is the only public method defined, and has the following signature:

fromRoute (*string* \$route = null, *array* \$params = array(), *array* \$options = array(), *boolean* \$reuseMatchedParams = false)

Generate url string from given parameters:

Parameters

- **\$name** (*string*) – The name of the route you want to output
- **\$params** (*array*) – An array of parameters that is defined within the respective route configuration
- **\$options** (*array*) – An array of options that will be used to create the URL (e.g., `force_canonical`, `query`)
- **\$reuseMatchedParams** (*boolean*) – A flag indicating if the currently matched route parameters should be used when generating the new URL

Return type `string`

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.

Controllers

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:

```
1 // Using explicit accessor methods
2 $request = $this->getRequest();
3 $response = $this->getResponse();
4
5 // Using direct property access
6 $request = $this->request;
7 $response = $this->response;
```

Additionally, if your controller implements `InjectApplicationEventInterface` (as both `AbstractActionController` and `AbstractRestfulController` do), you can access these objects from the attached `MvcEvent`:

```
1 $event = $this->getEvent();
2 $request = $event->getRequest();
3 $response = $event->getResponse();
```

The above can be useful when composing event listeners into your controller.

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`:

```
1 $event = $this->getEvent();
2 $matches = $event->getRouteMatch();
```

Once you have the `RouteMatch` object, you can pull parameters from it.

The same can be done using the *Params plugin*.

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 Unauthorized” results, etc.

Bootstrapping

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:

```
1 namespace SomeCustomModule;
2
3 class Module
4 {
5     /**
6      * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
7      * @return void
8      */
9     public function onBootstrap($e)
10     {
11         $application = $e->getApplication();
12         $config       = $application->getConfig();
13         $view         = $application->getServiceManager()->get('ViewHelperManager');
14         // You must have these keys in you application config
15         $view->headTitle($config['view']['base_title']);
16
17         // This is your custom listener
```

```
18     $listener = new Listeners\ViewListener();
19     $listener->setView($view);
20     $application->getEventManager()->attachAggregate($listener);
21 }
22 }
```

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.

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.

Pages and Containers

There are two main concepts in Zend\Navigation:

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.

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.

View Helpers

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.

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.

```
1 <?php
2 // your configuration file, e.g. config/autoload/global.php
3 return array(
4     // ...
5
6     'navigation' => array(
7         'default' => array(
8             array(
9                 'label' => 'Home',
10                'route' => 'home',
11            ),
12            array(
13                'label' => 'Page #1',
14                'route' => 'page-1',
15                'pages' => array(
16                    array(
17                        'label' => 'Child #1',
18                        'route' => 'page-1-child',
19                    ),
20                ),
21            ),
22            array(
23                'label' => 'Page #2',
24                'route' => 'page-2',
25            ),
26        ),
27    ),
28    'service_manager' => array(
29        'factories' => array(
30            'navigation' => 'Zend\Navigation\Service\DefaultNavigationFactory',
31        ),
```

```
32     ),
33     // ...
34 );
```

```
1 <!-- in your layout -->
2 <!-- ... -->
3
4 <body>
5     <?php echo $this->navigation('navigation')->menu(); ?>
6 </body>
7 <!-- ... -->
```

Using multiple navigations

If you want to use more than one navigation, you can register the abstract factory `\Zend\Navigation\Service\NavigationAbstractServiceFactory` in the *service manager*.

Once the service factory is registered, you can create as many navigation definitions as you wish, and the factory will create navigation containers automatically. This factory can also be used for the default container.

```
1 <?php
2 // your configuration file, e.g. config/autoload/global.php
3 return array(
4     // ...
5
6     'navigation' => array(
7
8         // navigation with name default
9         'default' => array(
10             array(
11                 'label' => 'Home',
12                 'route' => 'home',
13             ),
14             array(
15                 'label' => 'Page #1',
16                 'route' => 'page-1',
17                 'pages' => array(
18                     array(
19                         'label' => 'Child #1',
20                         'route' => 'page-1-child',
21                     ),
22                 ),
23             ),
24             array(
25                 'label' => 'Page #2',
26                 'route' => 'page-2',
27             ),
28         ),
29
30         // navigation with name special
31         'special' => array(
32             array(
33                 'label' => 'Special',
34                 'route' => 'special',
35             ),
```



```

36         array(
37             'label' => 'Special Page #2',
38             'route' => 'special-2',
39         ),
40     ),
41
42     // navigation with name sitemap
43     'sitemap' => array(
44         array(
45             'label' => 'Sitemap',
46             'route' => 'sitemap',
47         ),
48         array(
49             'label' => 'Sitemap Page #2',
50             'route' => 'sitemap-2',
51         ),
52     ),
53 ),
54 'service_manager' => array(
55     'abstract_factories' => array(
56         'Zend\Navigation\Service\NavigationAbstractServiceFactory'
57     ),
58 ),
59 // ...
60 );

```

Note: There is one important point if you use the `NavigationAbstractServiceFactory`: The name of the service in your view must start with `Zend\Navigation\` followed by the name of the configuration key. This helps ensure that no naming collisions occur with other services.

The following example demonstrates rendering the navigation menus for the named default, special and sitemap containers.

```

1  <!-- in your layout -->
2  <!-- ... -->
3
4  <body>
5      <?php echo $this->navigation('Zend\Navigation\Default')->menu(); ?>
6
7      <?php echo $this->navigation('Zend\Navigation\Special')->menu(); ?>
8
9      <?php echo $this->navigation('Zend\Navigation\Sitemap')->menu(); ?>
10 </body>
11 <!-- ... -->

```


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 `foo<a>`).

orphan

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](#).

Table 224.1: Common page options

Key	Type	Default	Description
label	String	NULL	LA page label, such as ‘Home’ or ‘Blog’.
fragment	String NULL	NULL	LA 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: <code>http://www.example.org/foo.html#bar</code> (<i>bar</i> is the fragment identifier)
id	String Integer	NULL	An <i>id</i> 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 <code>'alternate' => 'format/plain.html'</code> . To allow full flexibility, there are no restrictions on relation values. The value does not have to be a string. Read more about <code>rel</code> and <code>rev</code> in the section on the Links helper.
rev	Array	array()	Specifies reverse relations for the page. Works exactly like <code>rel</code> .
order	String Integer NULL	NULL	Works like <code>order</code> for elements in <code>Zend\Form</code> . 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 <code>order</code> attribute to a low number, e.g. <code>-100</code> . If a String is given, it must parse to a valid int. If <code>NULL</code> is given, it will be reset, meaning the order in which the page was added to the container will be used.
resource	String <code>Zend\Permissions\Acl\Resource</code> 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 <code>active</code> is <code>FALSE</code> or not given, MVC pages will check its properties against the request object upon calling <code>\$page->isActive()</code> .
visible	Boolean	TRUE	Whether page should be visible for the user, or just be a part of the structure. Invisible pages are skipped by view helpers.
pages	Array <code>Zend\Config</code> NULL	NULL	Child pages of the page. This could be an Array or <code>Zend\Config</code> object containing either page options that can be passed to the <code>factory()</code> method, or actual <code>Zend\Navigation\Page\AbstractPage</code> instances, or a mixture of both.

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.

```
1 $page = new Zend\Navigation\Page\Mvc();
2 $page->foo    = 'bar';
3 $page->meaning = 42;
4
5 echo $page->foo;
6
7 if ($page->meaning != 42) {
8     // action should be taken
9 }
```

orphan

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.

Note: Starting in version 2.2.0, if you want to re-use any matched route parameters when generating a link, you can do so via the “useRouteMatch” flag. This is particularly useful when creating segment routes that include the currently selected language or locale as an initial segment, as it ensures the links generated all include the matched value.

Table 224.2: 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.
useRouteMatch	Boolean	FALSE	If true, then getHref method will use the routeMatch parameters to assemble the URI
router	Zend\Mvc\Router\RouteStackInterface	NULL	Router for assembling URLs
query	Array	array()	User query params to use when generating href to page

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 *URIs* when calling `$page->getHref()`.

```

1 // Create route
2 $route = Zend\Mvc\Router\Http\Segment::factory(array(
3     'route' => '/[:controller][:action][:id]]',
4     'constraints' => array(
5         'controller' => '[a-zA-Z][a-zA-Z0-9_-]+',
6         'action' => '[a-zA-Z][a-zA-Z0-9_-]+',
7         'id' => '[0-9]+',
8     ),
9     array(
10        'controller' => 'Album\Controller\Album',
11        'action' => 'index',
12    )
13 ));
14 $router = new Zend\Mvc\Router\Http\TreeRouteStack();
15 $router->addRoute('default', $route);
16
17 // getHref() returns /album/add
18 $page = new Zend\Navigation\Page\Mvc(array(
19     'action' => 'add',
20     'controller' => 'album',
21 ));
22
23 // getHref() returns /album/edit/1337
24 $page = new Zend\Navigation\Page\Mvc(array(
25     'action' => 'edit',
26     'controller' => 'album',
27     'params' => array('id' => 1337),
28 ));

```

```

29 // getHref() returns /album/1337?format=json
30
31 $page = new Zend\Navigation\Page\Mvc(array(
32     'action'      => 'edit',
33     'controller'  => 'album',
34     'params'      => array('id' => 1337),
35     'query'       => array('format' => 'json'),
36 ));

```

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.

```

1  /**
2   * Dispatched request:
3   * - controller: album
4   * - action:      index
5   */
6  $page1 = new Zend\Navigation\Page\Mvc(array(
7      'action'      => 'index',
8      'controller'  => 'album',
9  ));
10
11  $page2 = new Zend\Navigation\Page\Mvc(array(
12      'action'      => 'edit',
13      'controller'  => 'album',
14  ));
15
16  $page1->isActive(); // returns true
17  $page2->isActive(); // returns false
18
19  /**
20   * Dispatched request:
21   * - controller: album
22   * - action:      edit
23   * - id:          1337
24   */
25  $page = new Zend\Navigation\Page\Mvc(array(
26      'action'      => 'edit',
27      'controller'  => 'album',
28      'params'      => array('id' => 1337),
29  ));
30
31  // returns true, because request has the same controller and action
32  $page->isActive();
33
34  /**
35   * Dispatched request:
36   * - controller: album
37   * - action:      edit
38   */
39  $page = new Zend\Navigation\Page\Mvc(array(
40      'action'      => 'edit',
41      'controller'  => 'album',
42      'params'      => array('id' => null),

```

```
43 ));  
44  
45 // returns false, because page requires the id param to be set in the request  
46 $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.).

```
1 // the following route is added to the ZF router  
2 $route = Zend\Mvc\Router\Http\Segment::factory(array(  
3     'route' => '/a:id',  
4     'constraints' => array(  
5         'id' => '[0-9]+',  
6     ),  
7     array(  
8         'controller' => 'Album\Controller\Album',  
9         'action' => 'show',  
10    )  
11 ));  
12 $router = new Zend\Mvc\Router\Http\TreeRouteStack();  
13 $router->addRoute('albumShow', $route);  
14  
15 // a page is created with a 'route' option  
16 $page = new Zend\Navigation\Page\Mvc(array(  
17     'label' => 'Show album',  
18     'route' => 'albumShow',  
19     'params' => array('id' => 42)  
20 ));  
21  
22 // returns: /a/42  
23 $page->getHref();
```

orphan

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 224.3: URI page options

Key	Type	Default	Description
uri	String	NULL	URI to page. This can be any string or NULL.

orphan

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.

```

1 class My\Simple\Page extends Zend\Navigation\Page\AbstractPage
2 {
3     public function getHref()
4     {
5         return 'something-completely-different';
6     }
7 }
```

A custom page with properties

When adding properties to an extended page, there is no need to override/modify `setOptions()` or `setConfig()`.

```

1 class My\Navigation\Page extends Zend\Navigation\Page\AbstractPage
2 {
3     protected $foo;
4     protected $fooBar;
5
6     public function setFoo($foo)
7     {
8         $this->foo = $foo;
9     }
10
11    public function getFoo()
12    {
13        return $this->foo;
14    }
15
16    public function setFooBar($fooBar)
17    {
18        $this->fooBar = $fooBar;
19    }
20 }
```

```
19     }
20
21     public function getFooBar()
22     {
23         return $this->fooBar;
24     }
25
26     public function getHref()
27     {
28         return $this->foo . '/' . $this->fooBar;
29     }
30 }
31
32 // can now construct using
33 $page = new My\Navigation\Page(array(
34     'label'    => 'Property names are mapped to setters',
35     'foo'      => 'bar',
36     'foo_bar' => 'baz'
37 ));
38
39 // ...or
40 $page = Zend\Navigation\Page\AbstractPage::factory(array(
41     'type'     => 'My\Navigation\Page',
42     'label'    => 'Property names are mapped to setters',
43     'foo'      => 'bar',
44     'foo_bar' => 'baz'
45 ));
```

orphan

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

```
1 $page = Zend\Navigation\Page\AbstractPage::factory(array(
2     'label' => 'My MVC page',
3     'action' => 'index',
4 ));
5
6 $page = Zend\Navigation\Page\AbstractPage::factory(array(
7     'label'    => 'Search blog',
8     'action'    => 'index',
9     'controller' => 'search',
10 ));
11
```

```

12 $page = Zend\Navigation\Page\AbstractPage::factory(array(
13     'label' => 'Home',
14     'route' => 'home',
15 ));
16
17 $page = Zend\Navigation\Page\AbstractPage::factory(array(
18     'type'    => 'mvc',
19     'label'   => 'My MVC page',
20 ));

```

Creating a URI page using the page factory

```

1  $page = Zend\Navigation\Page\AbstractPage::factory(array(
2      'label' => 'My URI page',
3      'uri'   => 'http://www.example.com/',
4  ));
5
6  $page = Zend\Navigation\Page\AbstractPage::factory(array(
7      'label' => 'Search',
8      'uri'   => 'http://www.example.com/search',
9      'active' => true,
10 ));
11
12 $page = Zend\Navigation\Page\AbstractPage::factory(array(
13     'label' => 'My URI page',
14     'uri'   => '#',
15 ));
16
17 $page = Zend\Navigation\Page\AbstractPage::factory(array(
18     'type' => 'uri',
19     'label' => 'My URI page',
20 ));

```

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.

```

1  class My\Navigation\Page extends Zend\Navigation\Page\AbstractPage
2  {
3      protected $_fooBar = 'ok';
4
5      public function setFooBar($fooBar)
6      {
7          $this->_fooBar = $fooBar;
8      }
9  }
10
11 $page = Zend\Navigation\Page\AbstractPage::factory(array(
12     'type'    => 'My\Navigation\Page',
13     'label'   => 'My custom page',
14     'foo_bar' => 'foo bar',
15 ));

```


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.

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

```
1  /*
2   * Create a container from an array
3   *
4   * Each element in the array will be passed to
5   * Zend\Navigation\Page\AbstractPage::factory() when constructing.
6   */
7  $container = new Zend\Navigation\Navigation(array(
8      array(
9          'label' => 'Page 1',
10         'id' => 'home-link',
11         'uri' => '/',
12     ),
13     array(
14         'label' => 'Zend',
15         'uri' => 'http://www.zend-project.com/',
```

```
16     'order' => 100,
17 ),
18 array(
19     'label' => 'Page 2',
20     'controller' => 'page2',
21     'pages' => array(
22         array(
23             'label' => 'Page 2.1',
24             'action' => 'page2_1',
25             'controller' => 'page2',
26             'class' => 'special-one',
27             'title' => 'This element has a special class',
28             'active' => true,
29         ),
30         array(
31             'label' => 'Page 2.2',
32             'action' => 'page2_2',
33             'controller' => 'page2',
34             'class' => 'special-two',
35             'title' => 'This element has a special class too',
36         ),
37     ),
38 ),
39 array(
40     'label' => 'Page 2 with params',
41     'action' => 'index',
42     'controller' => 'page2',
43     // specify a param or two,
44     'params' => array(
45         'format' => 'json',
46         'foo' => 'bar',
47     )
48 ),
49 array(
50     'label' => 'Page 2 with params and a route',
51     'action' => 'index',
52     'controller' => 'page2',
53     // specify a route name and a param for the route
54     'route' => 'nav-route-example',
55     'params' => array(
56         'format' => 'json',
57     ),
58 ),
59 array(
60     'label' => 'Page 3',
61     'action' => 'index',
62     'controller' => 'index',
63     'module' => 'mymodule',
64     'reset_params' => false,
65 ),
66 array(
67     'label' => 'Page 4',
68     'uri' => '#',
69     'pages' => array(
70         array(
71             'label' => 'Page 4.1',
72             'uri' => '/page4',
73             'title' => 'Page 4 using uri',
```

```

74         'pages' => array(
75             array(
76                 'label' => 'Page 4.1.1',
77                 'title' => 'Page 4 using mvc params',
78                 'action' => 'index',
79                 'controller' => 'page4',
80                 // let's say this page is active
81                 'active' => '1',
82             )
83         ),
84     ),
85 ),
86 ),
87 array(
88     'label' => 'Page 0?',
89     'uri' => '/setting/the/order/option',
90     // setting order to -1 should make it appear first
91     'order' => -1,
92 ),
93 array(
94     'label' => 'Page 5',
95     'uri' => '/',
96     // this page should not be visible
97     'visible' => false,
98     'pages' => array(
99         array(
100             'label' => 'Page 5.1',
101             'uri' => '#',
102             'pages' => array(
103                 array(
104                     'label' => 'Page 5.1.1',
105                     'uri' => '#',
106                     'pages' => array(
107                         array(
108                             'label' => 'Page 5.1.2',
109                             'uri' => '#',
110                             // let's say this page is active
111                             'active' => true,
112                         ),
113                     ),
114                 ),
115             ),
116         ),
117     ),
118 ),
119 array(
120     'label' => 'ACL page 1 (guest)',
121     'uri' => '#acl-guest',
122     'resource' => 'nav-guest',
123     'pages' => array(
124         array(
125             'label' => 'ACL page 1.1 (foo)',
126             'uri' => '#acl-foo',
127             'resource' => 'nav-foo',
128         ),
129         array(
130             'label' => 'ACL page 1.2 (bar)',
131             'uri' => '#acl-bar',

```

```

132         'resource' => 'nav-bar',
133     ),
134     array(
135         'label' => 'ACL page 1.3 (baz)',
136         'uri' => '#acl-baz',
137         'resource' => 'nav-baz',
138     ),
139     array(
140         'label' => 'ACL page 1.4 (bat)',
141         'uri' => '#acl-bat',
142         'resource' => 'nav-bat',
143     ),
144 ),
145 ),
146 array(
147     'label' => 'ACL page 2 (member)',
148     'uri' => '#acl-member',
149     'resource' => 'nav-member',
150 ),
151 array(
152     'label' => 'ACL page 3 (admin',
153     'uri' => '#acl-admin',
154     'resource' => 'nav-admin',
155     'pages' => array(
156         array(
157             'label' => 'ACL page 3.1 (nothing)',
158             'uri' => '#acl-nada',
159         ),
160     ),
161 ),
162 array(
163     'label' => 'Zend Framework',
164     'route' => 'zf-route',
165 ),
166 ));

```

Creating a container using a config object

```

1  /* CONTENTS OF /path/to/navigation.xml:
2  <?xml version="1.0" encoding="UTF-8"?>
3  <nav>
4
5      <zend>
6          <label>Zend</label>
7          <uri>http://www.zend-project.com/</uri>
8          <order>100</order>
9      </zend>
10
11     <page1>
12         <label>Page 1</label>
13         <uri>page1</uri>
14         <pages>
15
16             <page1_1>
17                 <label>Page 1.1</label>
18                 <uri>page1/page1_1</uri>

```



```

19         </page1_1>
20
21     </pages>
22 </page1>
23
24 <page2>
25     <label>Page 2</label>
26     <uri>page2</uri>
27     <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>
86         </pages>
87     </page2_3_2_2>
88
89     </pages>
90 </page2_3_2>
91
92 <page2_3_3>
93     <label>Page 2.3.3</label>
94     <uri>page2/page2_3/page2_3_3</uri>
95     <resource>admin</resource>
96     <pages>
97
98         <page2_3_3_1>
99             <label>Page 2.3.3.1</label>
100             <uri>page2/page2_3/page2_3_3/1</uri>
101             <active>1</active>
102         </page2_3_3_1>
103
104         <page2_3_3_2>
105             <label>Page 2.3.3.2</label>
106             <uri>page2/page2_3/page2_3_3/2</uri>
107             <resource>guest</resource>
108             <active>1</active>
109         </page2_3_3_2>
110
111     </pages>
112 </page2_3_3>
113
114 </pages>
115 </page2_3>
116
117 </pages>
118 </page2>
119
120 <page3>
121     <label>Page 3</label>
122     <uri>page3</uri>
123     <pages>
124
125         <page3_1>
126             <label>Page 3.1</label>
127             <uri>page3/page3_1</uri>
128             <resource>guest</resource>
129         </page3_1>
130
131         <page3_2>
132             <label>Page 3.2</label>
133             <uri>page3/page3_2</uri>
134             <resource>member</resource>
```

```

135         <pages>
136
137             <page3_2_1>
138                 <label>Page 3.2.1</label>
139                 <uri>page3/page3_2/page3_2_1</uri>
140             </page3_2_1>
141
142             <page3_2_2>
143                 <label>Page 3.2.2</label>
144                 <uri>page3/page3_2/page3_2_2</uri>
145                 <resource>admin</resource>
146             </page3_2_2>
147
148         </pages>
149     </page3_2>
150
151     <page3_3>
152         <label>Page 3.3</label>
153         <uri>page3/page3_3</uri>
154         <resource>special</resource>
155         <pages>
156
157             <page3_3_1>
158                 <label>Page 3.3.1</label>
159                 <uri>page3/page3_3/page3_3_1</uri>
160                 <visible>0</visible>
161             </page3_3_1>
162
163             <page3_3_2>
164                 <label>Page 3.3.2</label>
165                 <uri>page3/page3_3/page3_3_2</uri>
166                 <resource>admin</resource>
167             </page3_3_2>
168
169         </pages>
170     </page3_3>
171
172 </pages>
173 </page3>
174
175 <home>
176     <label>Home</label>
177     <order>-100</order>
178     <module>default</module>
179     <controller>index</controller>
180     <action>index</action>
181 </home>
182
183 </nav>
184 */
185
186 $reader = new Zend\Config\Reader\Xml();
187 $config = $reader->fromFile('/path/to/navigation.xml');
188 $container = new Zend\Navigation\Navigation($config);

```

Adding pages

Adding pages to a container can be done with the methods `addPage()`, `addPages()`, or `setPages()`. See examples below for explanation.

```
1 // create container
2 $container = new Zend\Navigation\Navigation();
3
4 // add page by giving a page instance
5 $container->addPage(
6     Zend\Navigation\Page\AbstractPage::factory(
7         array(
8             'uri' => 'http://www.example.com/',
9         )
10    );
11);
12
13 // add page by giving an array
14 $container->addPage(
15     array(
16         'uri' => 'http://www.example.com/',
17     )
18 );
19
20 // add page by giving a config object
21 $container->addPage(
22     new Zend\Config\Config(
23         array(
24             'uri' => 'http://www.example.com/',
25         )
26     )
27 );
28
29 $pages = array(
30     array(
31         'label' => 'Save',
32         'action' => 'save',
33     ),
34     array(
35         'label' => 'Delete',
36         'action' => 'delete',
37     )
38 );
39
40 // add two pages
41 $container->addPages($pages);
42
43 // remove existing pages and add the given pages
44 $container->setPages($pages);
```

Removing pages

Removing pages can be done with `removePage()` or `removePages()`. The first method accepts 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.

```

1 $container = new Zend\Navigation\Navigation(array(
2     array(
3         'label' => 'Page 1',
4         'action' => 'page1',
5     ),
6     array(
7         'label' => 'Page 2',
8         'action' => 'page2',
9         'order' => 200,
10    ),
11    array(
12        'label' => 'Page 3',
13        'action' => 'page3',
14    )
15));
16
17 // remove page by implicit page order
18 $container->removePage(0); // removes Page 1
19
20 // remove page by instance
21 $page3 = $container->findOneByAction('page3');
22 $container->removePage($page3); // removes Page 3
23
24 // remove page by explicit page order
25 $container->removePage(200); // removes Page 2
26
27 // remove all pages
28 $container->removePages(); // removes all pages

```

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')`.

```

1 $container = new Zend\Navigation\Navigation(array(
2     array(
3         'label' => 'Page 1',
4         'uri'   => 'page-1',
5         'foo'   => 'bar',
6         'pages' => array(
7             array(
8                 'label' => 'Page 1.1',
9                 'uri'   => 'page-1.1',
10                'foo'   => 'bar',
11            ),
12            array(

```

```

13         'label' => 'Page 1.2',
14         'uri'   => 'page-1.2',
15         'class' => 'my-class',
16     ),
17     array(
18         'type'   => 'uri',
19         'label'  => 'Page 1.3',
20         'uri'    => 'page-1.3',
21         'action' => 'about',
22     )
23 )
24 ),
25 array(
26     'label'      => 'Page 2',
27     'id'         => 'page_2_and_3',
28     'class'      => 'my-class',
29     'module'     => 'page2',
30     'controller' => 'index',
31     'action'     => 'page1',
32 ),
33 array(
34     'label'      => 'Page 3',
35     'id'         => 'page_2_and_3',
36     'module'     => 'page3',
37     'controller' => 'index',
38 ),
39 );
40
41 // The 'id' is not required to be unique, but be aware that
42 // having two pages with the same id will render the same id attribute
43 // in menus and breadcrumbs.
44 $found = $container->findBy('id',
45                             'page_2_and_3'); // returns Page 2
46 $found = $container->findOneBy('id',
47                               'page_2_and_3'); // returns Page 2
48 $found = $container->findBy('id',
49                             'page_2_and_3',
50                             true); // returns Page 2 and Page 3
51 $found = $container->findById('page_2_and_3'); // returns Page 2
52 $found = $container->findOneById('page_2_and_3'); // returns Page 2
53 $found = $container->findAllById('page_2_and_3'); // returns Page 2 and Page 3
54
55 // Find all matching CSS class my-class
56 $found = $container->findAllBy('class',
57                               'my-class'); // returns Page 1.2 and Page 2
58 $found = $container->findAllByClass('my-class'); // returns Page 1.2 and Page 2
59
60 // Find first matching CSS class my-class
61 $found = $container->findOneByClass('my-class'); // returns Page 1.2
62
63 // Find all matching CSS class non-existent
64 $found = $container->findAllByClass('non-existent'); // returns array()
65
66 // Find first matching CSS class non-existent
67 $found = $container->findOneByClass('non-existent'); // returns null
68
69 // Find all pages with custom property 'foo' = 'bar'
70 $found = $container->findAllBy('foo', 'bar'); // returns Page 1 and Page 1.1

```

```

71 // To achieve the same magically, 'foo' must be in lowercase.
72 // This is because 'foo' is a custom property, and thus the
73 // property name is not normalized to 'Foo'
74 $found = $container->findAllByfoo('bar');
75
76
77 // Find all with controller = 'index'
78 $found = $container->findAllByController('index'); // returns Page 2 and Page 3

```

Iterating containers

Zend\Navigation\AbstractContainer implements RecursiveIterator, and can be iterated using any Iterator class. To iterate a container recursively, use the RecursiveIteratorIterator class.

```

1  /*
2   * Create a container from an array
3   */
4  $container = new Zend\Navigation\Navigation(array(
5      array(
6          'label' => 'Page 1',
7          'uri'   => '#',
8      ),
9      array(
10         'label' => 'Page 2',
11         'uri'   => '#',
12         'pages' => array(
13             array(
14                 'label' => 'Page 2.1',
15                 'uri'   => '#',
16             ),
17             array(
18                 'label' => 'Page 2.2',
19                 'uri'   => '#',
20             )
21         )
22     )
23     array(
24         'label' => 'Page 3',
25         'uri'   => '#',
26     ),
27 ));
28
29 // Iterate flat using regular foreach:
30 // Output: Page 1, Page 2, Page 3
31 foreach ($container as $page) {
32     echo $page->label;
33 }
34
35 // Iterate recursively using RecursiveIteratorIterator
36 $it = new RecursiveIteratorIterator(
37     $container, RecursiveIteratorIterator::SELF_FIRST);
38
39 // Output: Page 1, Page 2, Page 2.1, Page 2.2, Page 3
40 foreach ($it as $page) {
41     echo $page->label;

```

42 }

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

```
1 $container = new Zend\Navigation\Navigation(array(
2     array(
3         'label' => 'Page 1',
4         'uri'   => '#',
5     ),
6     array(
7         'label' => 'Page 2',
8         'uri'   => '#',
9         'pages' => array(
10             array(
11                 'label' => 'Page 2.1',
12                 'uri'   => '#',
13             ),
14             array(
15                 'label' => 'Page 2.2',
16                 'uri'   => '#',
17             ),
18         ),
19     ),
20 ));
21
22 var_dump($container->toArray());
23
24 /* Output:
25 array(2) {
26     [0]=> array(15) {
27         ["label"]=> string(6) "Page 1"
28         ["id"]=> NULL
29         ["class"]=> NULL
30         ["title"]=> NULL
31         ["target"]=> NULL
32         ["rel"]=> array(0) {
33             }
34         ["rev"]=> array(0) {
35             }
36         ["order"]=> NULL
37         ["resource"]=> NULL
38         ["privilege"]=> NULL
39         ["active"]=> bool(false)
40         ["visible"]=> bool(true)
41         ["type"]=> string(23) "Zend\Navigation\Page\Uri"
```



```

42     ["pages"]=> array(0) {
43     }
44     ["uri"]=> string(1) "#"
45 }
46 [1]=> array(15) {
47     ["label"]=> string(6) "Page 2"
48     ["id"]=> NULL
49     ["class"]=> NULL
50     ["title"]=> NULL
51     ["target"]=> NULL
52     ["rel"]=> array(0) {
53     }
54     ["rev"]=> array(0) {
55     }
56     ["order"]=> NULL
57     ["resource"]=> NULL
58     ["privilege"]=> NULL
59     ["active"]=> bool(false)
60     ["visible"]=> bool(true)
61     ["type"]=> string(23) "Zend\Navigation\Page\Uri"
62     ["pages"]=> array(2) {
63         [0]=> array(15) {
64             ["label"]=> string(8) "Page 2.1"
65             ["id"]=> NULL
66             ["class"]=> NULL
67             ["title"]=> NULL
68             ["target"]=> NULL
69             ["rel"]=> array(0) {
70             }
71             ["rev"]=> array(0) {
72             }
73             ["order"]=> NULL
74             ["resource"]=> NULL
75             ["privilege"]=> NULL
76             ["active"]=> bool(false)
77             ["visible"]=> bool(true)
78             ["type"]=> string(23) "Zend\Navigation\Page\Uri"
79             ["pages"]=> array(0) {
80             }
81             ["uri"]=> string(1) "#"
82         }
83         [1]=>
84         array(15) {
85             ["label"]=> string(8) "Page 2.2"
86             ["id"]=> NULL
87             ["class"]=> NULL
88             ["title"]=> NULL
89             ["target"]=> NULL
90             ["rel"]=> array(0) {
91             }
92             ["rev"]=> array(0) {
93             }
94             ["order"]=> NULL
95             ["resource"]=> NULL
96             ["privilege"]=> NULL
97             ["active"]=> bool(false)
98             ["visible"]=> bool(true)
99             ["type"]=> string(23) "Zend\Navigation\Page\Uri"

```

```
100         ["pages"]=> array(0) {
101         }
102         ["uri"]=> string(1) "#"
103     }
104 }
105 ["uri"]=> string(1) "#"
106 }
107 }
108 */
```

Introduction

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. `isTranslatorEnabled()` and `setTranslatorEnabled()` 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 *Role* 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.

```
1 $this->navigation()->addPage(array(
2     'type' => 'uri',
3     'label' => 'New page'));
```

The call above will add a page to the container in the `Navigation` helper.

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->setTranslatorEnabled(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.

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.

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.

```

1  /*
2   * Navigation container (config/array)
3
4   * Each element in the array will be passed to
5   * Zend\Navigation\Page\AbstractPage::factory() when constructing
6   * the navigation container below.
7   */
8  $pages = array(
9      array(
10         'label'      => 'Home',
11         'title'      => 'Go Home',
12         'module'     => 'default',
13         'controller' => 'index',
14         'action'     => 'index',
15         'order'      => -100 // make sure home is the first page
16     ),
17     array(
18         'label'      => 'Special offer this week only!',
19         'module'     => 'store',
20         'controller' => 'offer',
21         'action'     => 'amazing',
22         'visible'    => false // not visible
23     ),
24     array(
25         'label'      => 'Products',
26         'module'     => 'products',
27         'controller' => 'index',

```

```

28     'action'      => 'index',
29     'pages'      => array(
30         array(
31             'label'      => 'Foo Server',
32             'module'     => 'products',
33             'controller' => 'server',
34             'action'     => 'index',
35             'pages'      => array(
36                 array(
37                     'label'      => 'FAQ',
38                     'module'     => 'products',
39                     'controller' => 'server',
40                     'action'     => 'faq',
41                     'rel'        => array(
42                         'canonical' => 'http://www.example.com/?page=faq',
43                         'alternate' => array(
44                             'module'     => 'products',
45                             'controller' => 'server',
46                             'action'     => 'faq',
47                             'params'     => array('format' => 'xml')
48                         )
49                     )
50                 ),
51                 array(
52                     'label'      => 'Editions',
53                     'module'     => 'products',
54                     'controller' => 'server',
55                     'action'     => 'editions'
56                 ),
57                 array(
58                     'label'      => 'System Requirements',
59                     'module'     => 'products',
60                     'controller' => 'server',
61                     'action'     => 'requirements'
62                 )
63             )
64         ),
65         array(
66             'label'      => 'Foo Studio',
67             'module'     => 'products',
68             'controller' => 'studio',
69             'action'     => 'index',
70             'pages'      => array(
71                 array(
72                     'label'      => 'Customer Stories',
73                     'module'     => 'products',
74                     'controller' => 'studio',
75                     'action'     => 'customers'
76                 ),
77                 array(
78                     'label'      => 'Support',
79                     'module'     => 'products',
80                     'controller' => 'studio',
81                     'action'     => 'support'
82                 )
83             )
84         )
85     )

```

```

86     ),
87     array(
88         'label'      => 'Company',
89         'title'      => 'About us',
90         'module'     => 'company',
91         'controller' => 'about',
92         'action'     => 'index',
93         'pages'      => array(
94             array(
95                 'label'      => 'Investor Relations',
96                 'module'     => 'company',
97                 'controller' => 'about',
98                 'action'     => 'investors'
99             ),
100             array(
101                 'label'      => 'News',
102                 'class'      => 'rss', // class
103                 'module'     => 'company',
104                 'controller' => 'news',
105                 'action'     => 'index',
106                 'pages'      => array(
107                     array(
108                         'label'      => 'Press Releases',
109                         'module'     => 'company',
110                         'controller' => 'news',
111                         'action'     => 'press'
112                     ),
113                     array(
114                         'label'      => 'Archive',
115                         'route'      => 'archive', // route
116                         'module'     => 'company',
117                         'controller' => 'news',
118                         'action'     => 'archive'
119                     )
120                 )
121             )
122         )
123     ),
124     array(
125         'label'      => 'Community',
126         'module'     => 'community',
127         'controller' => 'index',
128         'action'     => 'index',
129         'pages'      => array(
130             array(
131                 'label'      => 'My Account',
132                 'module'     => 'community',
133                 'controller' => 'account',
134                 'action'     => 'index',
135                 'resource'    => 'mvc:community.account' // resource
136             ),
137             array(
138                 'label'      => 'Forums',
139                 'uri'        => 'http://forums.example.com/',
140                 'class'      => 'external' // class
141             )
142         )
143     ),

```

```

144     array(
145         'label'      => 'Administration',
146         'module'     => 'admin',
147         'controller' => 'index',
148         'action'     => 'index',
149         'resource'   => 'mvc:admin', // resource
150         'pages'      => array(
151             array(
152                 'label'      => 'Write new article',
153                 'module'     => 'admin',
154                 'controller' => 'post',
155                 'action'     => 'write'
156             )
157         )
158     )
159 );
160
161 // Create container from array
162 $container = new Zend\Navigation\Navigation($pages);
163
164 // Store the container in the proxy helper:
165 $view->plugin('navigation')->setContainer($container);
166
167 // ...or simply:
168 $view->navigation($container);

```

In addition to the container above, the following setup is assumed:

```

1 <?php
2 // module/MyModule/config/module.config.php
3
4 return array(
5     /* ... */
6     'router' array(
7         'routes' => array(
8             'archive' => array(
9                 'type'     => 'Segment',
10                 'options' => array(
11                     'route'     => '/archive/:year',
12                     'defaults' => array(
13                         'module'     => 'company',
14                         'controller' => 'news',
15                         'action'     => 'archive',
16                         'year'      => (int) date('Y') - 1,
17                     ),
18                     'constraints' => array(
19                         'year' => '\d+',
20                     ),
21                 ),
22             ),
23             /* You can have other routes here... */
24         ),
25     ),
26     /* ... */
27 );

```

```

1 <?php
2 // module/MyModule/Module.php

```



```

3
4 namespace MyModule;
5
6 use Zend\View\HelperPluginManager;
7 use Zend\Permissions\Acl\Acl;
8 use Zend\Permissions\Acl\Role\GenericRole;
9 use Zend\Permissions\Acl\Resource\GenericResource;
10
11 class Module
12 {
13     /* ... */
14     public function getViewHelperConfig()
15     {
16         return array(
17             'factories' => array(
18                 // This will overwrite the native navigation helper
19                 'navigation' => function (HelperPluginManager $pm) {
20                     // Setup ACL:
21                     $acl = new Acl();
22                     $acl->addRole(new GenericRole('member'));
23                     $acl->addRole(new GenericRole('admin'));
24                     $acl->addResource(new GenericResource('mvc:admin'));
25                     $acl->addResource(new GenericResource('mvc:community.account'));
26                     $acl->allow('member', 'mvc:community.account');
27                     $acl->allow('admin', null);
28
29                     // Get an instance of the proxy helper
30                     $navigation = $pm->get('Zend\View\Helper\Navigation');
31
32                     // Store ACL and role in the proxy helper:
33                     $navigation->setAcl($acl)
34                         ->setRole('member');
35
36                     // Return the new navigation helper instance
37                     return $navigation;
38                 }
39             )
40         );
41     }
42     /* ... */
43 }

```

View Helper - Breadcrumbs

Introduction

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.

Basic usage

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:

`Products > Foo Server > 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:

<

Customize 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:

```

1 <?php
2 $this->navigation()->breadcrumbs()->setMinDepth(10);
3 echo $this->navigation()->breadcrumbs();
4 ?>

```

Output:

Nothing, because the deepest active page is not at level 10 or deeper.

Rendering using a partial view script

This example shows how to render customized breadcrumbs 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 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:

```

1 echo $this->navigation()->breadcrumbs()
2         ->setPartial('my-module/partials/breadcrumbs');

```

Contents of `module/MyModule/view/my-module/partials/breadcrumbs.phtml`:

```

1 echo implode(' ', array_map(
2     function ($a) { return $a->getLabel(); },
3     $this->pages));

```

Output:

```

1 Products, Foo Server, FAQ

```


Introduction

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 *HTML4 W3C Recommendation*.

There are two types of relations; forward and reverse, indicated by the keywords *‘rel’* and *‘rev’*. Most methods 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 *search\$rel\$type* (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.

Basic usage

Specify relations in pages

This example shows how to specify relations in pages.

```

1  $container = new Zend\Navigation\Navigation(array(
2      array(
3          'label' => 'Relations using strings',
4          'rel'   => array(
5              'alternate' => 'http://www.example.org/'
6          ),
7          'rev'   => array(
8              'alternate' => 'http://www.example.net/'
9          )
10     ),
11     array(
12         'label' => 'Relations using arrays',
13         'rel'   => array(
14             'alternate' => array(
15                 'label' => 'Example.org',
16                 'uri'   => 'http://www.example.org/'
17             )
18         )
19     ),
20     array(
21         'label' => 'Relations using configs',
22         'rel'   => array(
23             'alternate' => new Zend\Config\Config(array(
24                 'label' => 'Example.org',
25                 'uri'   => 'http://www.example.org/'
26             ))
27         )
28     ),
29     array(
30         'label' => 'Relations using pages instance',
31         'rel'   => array(
32             'alternate' => Zend\Navigation\Page\AbstractPage::factory(array(
33                 'label' => 'Example.org',
34                 'uri'   => 'http://www.example.org/'
35             ))
36         )
37     )
38 ));

```

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:

```
1 <?php echo $this->view->navigation()->links(); ?>
```

Output:

```
1 <link rel="alternate" href="/products/server/faq/format/xml">
2 <link rel="start" href="/" title="Home">
3 <link rel="next" href="/products/server/editions" title="Editions">
4 <link rel="prev" href="/products/server" title="Foo Server">
5 <link rel="chapter" href="/products" title="Products">
6 <link rel="chapter" href="/company/about" title="Company">
7 <link rel="chapter" href="/community" title="Community">
8 <link rel="canonical" href="http://www.example.com/?page=server-faq">
9 <link rel="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:

```
1 $helper->setRenderFlag(Zend\View\Helper\Navigation\Links::RENDER_START |
2                       Zend\View\Helper\Navigation\Links::RENDER_NEXT |
3                       Zend\View\Helper\Navigation\Links::RENDER_PREV);
```

Output:

```
1 <link rel="start" href="/" title="Home">
2 <link rel="next" href="/products/server/editions" title="Editions">
3 <link rel="prev" href="/products/server" title="Foo Server">
```

Render only native link types:

```
1 $helper->setRenderFlag(Zend\View\Helper\Navigation\Links::RENDER_ALL ^
2                       Zend\View\Helper\Navigation\Links::RENDER_CUSTOM);
```

Output:

```
1 <link rel="alternate" href="/products/server/faq/format/xml">
2 <link rel="start" href="/" title="Home">
3 <link rel="next" href="/products/server/editions" title="Editions">
4 <link rel="prev" href="/products/server" title="Foo Server">
5 <link rel="chapter" href="/products" title="Products">
6 <link rel="chapter" href="/company/about" title="Company">
7 <link rel="chapter" href="/community" title="Community">
8 <link rel="subsection" href="/products/server" title="Foo Server">
```

Render all but chapter:

```
1 $helper->setRenderFlag(Zend\View\Helper\Navigation\Links::RENDER_ALL ^  
2                        Zend\View\Helper\Navigation\Links::RENDER_CHAPTER);
```

Output:

```
1 <link rel="alternate" href="/products/server/faq/format/xml">  
2 <link rel="start" href="/" title="Home">  
3 <link rel="next" href="/products/server/editions" title="Editions">  
4 <link rel="prev" href="/products/server" title="Foo Server">  
5 <link rel="canonical" href="http://www.example.com/?page=server-faq">  
6 <link rev="subsection" href="/products/server" title="Foo Server">
```


Introduction

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}UIClass()` 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.

Basic usage

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:

```
1 <?php echo $this->navigation()->menu()->render() ?>
2
3 Or simply:
4 <?php echo $this->navigation()->menu() ?>
```

Output:

```
1 <ul class="navigation">
2   <li>
3     <a title="Go Home" href="/">Home</a>
4   </li>
5   <li class="active">
6     <a href="/products">Products</a>
7     <ul>
8       <li class="active">
9         <a href="/products/server">Foo Server</a>
10        <ul>
11          <li class="active">
12            <a href="/products/server/faq">FAQ</a>
13          </li>
14          <li>
15            <a href="/products/server/editions">Editions</a>
16          </li>
17          <li>
18            <a href="/products/server/requirements">System Requirements</
19      ↪ a>
20          </li>
21        </ul>
22      </li>
23      <li>
24        <a href="/products/studio">Foo Studio</a>
25        <ul>
26          <li>
27            <a href="/products/studio/customers">Customer Stories</a>
28          </li>
29          <li>
30            <a href="/products/studio/support">Support</a>
```

```

30         </li>
31     </ul>
32     </li>
33 </ul>
34 </li>
35 <li>
36     <a title="About us" href="/company/about">Company</a>
37     <ul>
38         <li>
39             <a href="/company/about/investors">Investor Relations</a>
40         </li>
41         <li>
42             <a class="rss" href="/company/news">News</a>
43             <ul>
44                 <li>
45                     <a href="/company/news/press">Press Releases</a>
46                 </li>
47                 <li>
48                     <a href="/archive">Archive</a>
49                 </li>
50             </ul>
51         </li>
52     </ul>
53 </li>
54 <li>
55     <a href="/community">Community</a>
56     <ul>
57         <li>
58             <a href="/community/account">My Account</a>
59         </li>
60         <li>
61             <a class="external" href="http://forums.example.com/">Forums</a>
62         </li>
63     </ul>
64 </li>
65 </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.

```

1 <?php
2 // render only the 'Community' menu
3 $community = $this->navigation()->findOneByLabel('Community');
4 $options = array(
5     'indent' => 16,
6     'ulClass' => 'community'
7 );
8 echo $this->navigation()
9     ->menu()
10    ->renderMenu($community, $options);
11 ?>

```

Output:

```
1         <ul class="community">
2             <li>
3                 <a href="/community/account">My Account</a>
4             </li>
5             <li>
6                 <a class="external" href="http://forums.example.com/">Forums</
7                 </li>
8             </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:

```
1 array(
2     'ulClass'      => $ulClass,
3     'indent'       => $indent,
4     'minDepth'     => null,
5     'maxDepth'     => null,
6     'onlyActiveBranch' => true,
7     'renderParents' => false
8 );
```

```
1 <?php
2 echo $this->navigation()
3     ->menu()
4     ->renderSubMenu(null, 'sidebar', 4);
5 ?>
```

The output will be the same if ‘FAQ’ or ‘Foo Server’ is active:

```
1 <ul class="sidebar">
2     <li class="active">
3         <a href="/products/server/faq">FAQ</a>
4     </li>
5     <li>
6         <a href="/products/server/editions">Editions</a>
7     </li>
8     <li>
9         <a href="/products/server/requirements">System Requirements</a>
10    </li>
11 </ul>
```

Rendering with maximum depth

```
1 <?php
2 echo $this->navigation()
3     ->menu()
4     ->setMaxDepth(1);
5 ?>
```


Output:

```

1 <ul class="navigation">
2   <li>
3     <a title="Go Home" href="/">Home</a>
4   </li>
5   <li class="active">
6     <a href="/products">Products</a>
7     <ul>
8       <li class="active">
9         <a href="/products/server">Foo Server</a>
10      </li>
11      <li>
12        <a href="/products/studio">Foo Studio</a>
13      </li>
14    </ul>
15  </li>
16  <li>
17    <a title="About us" href="/company/about">Company</a>
18    <ul>
19      <li>
20        <a href="/company/about/investors">Investor Relations</a>
21      </li>
22      <li>
23        <a class="rss" href="/company/news">News</a>
24      </li>
25    </ul>
26  </li>
27  <li>
28    <a href="/community">Community</a>
29    <ul>
30      <li>
31        <a href="/community/account">My Account</a>
32      </li>
33      <li>
34        <a class="external" href="http://forums.example.com/">Forums</a>
35      </li>
36    </ul>
37  </li>
38 </ul>

```

Rendering with minimum depth

```

1 <?php
2 echo $this->navigation()
3     ->menu()
4     ->setMinDepth(1);
5 ?>

```

Output:

```

1 <ul class="navigation">
2   <li class="active">
3     <a href="/products/server">Foo Server</a>

```

```
4      <ul>
5          <li class="active">
6              <a href="/products/server/faq">FAQ</a>
7          </li>
8          <li>
9              <a href="/products/server/editions">Editions</a>
10         </li>
11         <li>
12             <a href="/products/server/requirements">System Requirements</a>
13         </li>
14     </ul>
15 </li>
16 <li>
17     <a href="/products/studio">Foo Studio</a>
18     <ul>
19         <li>
20             <a href="/products/studio/customers">Customer Stories</a>
21         </li>
22         <li>
23             <a href="/products/studio/support">Support</a>
24         </li>
25     </ul>
26 </li>
27 <li>
28     <a href="/company/about/investors">Investor Relations</a>
29 </li>
30 <li>
31     <a class="rss" href="/company/news">News</a>
32     <ul>
33         <li>
34             <a href="/company/news/press">Press Releases</a>
35         </li>
36         <li>
37             <a href="/archive">Archive</a>
38         </li>
39     </ul>
40 </li>
41 <li>
42     <a href="/community/account">My Account</a>
43 </li>
44 <li>
45     <a class="external" href="http://forums.example.com/">Forums</a>
46 </li>
47 </ul>
```

Rendering only the active branch

```
1 <?php
2 echo $this->navigation()
3     ->menu()
4     ->setOnlyActiveBranch(true);
5 ?>
```

Output:

```

1 <ul class="navigation">
2   <li class="active">
3     <a href="/products">Products</a>
4     <ul>
5       <li class="active">
6         <a href="/products/server">Foo Server</a>
7         <ul>
8           <li class="active">
9             <a href="/products/server/faq">FAQ</a>
10          </li>
11          <li>
12            <a href="/products/server/editions">Editions</a>
13          </li>
14          <li>
15            <a href="/products/server/requirements">System Requirements</
16      <a>
17          </li>
18        </ul>
19      </li>
20    </ul>
21  </li>
22 </ul>

```

Rendering only the active branch with minimum depth

```

1 <?php
2 echo $this->navigation()
3     ->menu()
4     ->setOnlyActiveBranch(true)
5     ->setMinDepth(1);
6 ?>

```

Output:

```

1 <ul class="navigation">
2   <li class="active">
3     <a href="/products/server">Foo Server</a>
4     <ul>
5       <li class="active">
6         <a href="/products/server/faq">FAQ</a>
7       </li>
8       <li>
9         <a href="/products/server/editions">Editions</a>
10      </li>
11      <li>
12        <a href="/products/server/requirements">System Requirements</a>
13      </li>
14    </ul>
15  </li>
16 </ul>

```

Rendering only the active branch with maximum depth

```
1 <?php
2 echo $this->navigation()
3     ->menu()
4     ->setOnlyActiveBranch(true)
5     ->setMaxDepth(1);
6 ?>
```

Output:

```
1 <ul class="navigation">
2   <li class="active">
3     <a href="/products">Products</a>
4     <ul>
5       <li class="active">
6         <a href="/products/server">Foo Server</a>
7       </li>
8       <li>
9         <a href="/products/studio">Foo Studio</a>
10      </li>
11    </ul>
12  </li>
13 </ul>
```

Rendering only the active branch with maximum depth and no parents

```
1 <?php
2 echo $this->navigation()
3     ->menu()
4     ->setOnlyActiveBranch(true)
5     ->setRenderParents(false)
6     ->setMaxDepth(1);
7 ?>
```

Output:

```
1 <ul class="navigation">
2   <li class="active">
3     <a href="/products/server">Foo Server</a>
4   </li>
5   <li>
6     <a href="/products/studio">Foo Studio</a>
7   </li>
8 </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:

```
1 $this->navigation()->menu()->setPartial('my-module/partials/menu');
2 echo $this->navigation()->menu()->render();
```

In *module/MyModule/view/my-module/partials/menu.phtml*:

```
1 foreach ($this->container as $page) {
2     echo $this->navigation()->menu()->htmlify($page) . PHP_EOL;
3 }
```

Output:

```
1 <a title="Go Home" href="/">Home</a>
2 <a href="/products">Products</a>
3 <a title="About us" href="/company/about">Company</a>
4 <a href="/community">Community</a>
```

Using menu options in partial view script

In a layout:

```
1 // Set options
2 $this->navigation()->menu()->setUlClass('my-nav')
3     ->setPartial('my-module/partials/menu');
4
5 // Output menu
6 echo $this->navigation()->menu()->render();
```

In *module/MyModule/view/my-module/partials/menu.phtml*:

```
1 <div class"<?php echo $this->navigation()->menu()->getUlClass(); ?">
2     <?php
3         foreach ($this->container as $page) {
4             echo $this->navigation()->menu()->htmlify($page) . PHP_EOL;
5         }
6     ?>
7 </div>
```

Output:

```
1 <div class="my-nav">
2     <a title="Go Home" href="/">Home</a>
3     <a href="/products">Products</a>
4     <a title="About us" href="/company/about">Company</a>
5     <a href="/community">Community</a>
6 </div>
```

Using ACL in partial view script

If you want to use ACL within your partial view script, then you will have to check the access to a page manually.

In *module/MyModule/view/my-module/partials/menu.phtml*:

```
1 foreach ($this->container as $page) {  
2     if ($this->navigation()->accept($page)) {  
3         echo $this->navigation()->menu()->htmlify($page) . PHP_EOL;  
4     }  
5 }
```

View Helper - Sitemap

Introduction

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 230.1: Sitemap XML elements

Element	Description
loc	Absolute URL to page. An absolute URL will be generated by the helper.
last-mod	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.
change-freq	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 *URLs* in the `url()` method. If no server *URL* is specified, it will be determined by the helper.
- `url()` is used to generate absolute *URLs* to pages.
- `getDomSitemap()` generates a `DOMDocument` from a given container.

Basic usage

This example shows how to render an *XML* sitemap based on the setup we did further up.

```
1 // In a view script or layout:
2
3 // format output
4 $this->navigation()
5     ->sitemap()
6     ->setFormatOutput(true); // default is false
7
8 // other possible methods:
9 // ->setUseXmlDeclaration(false); // default is true
10 // ->setServerUrl('http://my.otherhost.com');
11 // default is to detect automatically
12
13 // print sitemap
14 echo $this->navigation()->sitemap();
```

Notice how pages that are invisible or pages with *ACL* roles incompatible with the view helper are filtered out:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
3     <url>
4         <loc>http://www.example.com/</loc>
5     </url>
6     <url>
7         <loc>http://www.example.com/products</loc>
8     </url>
9     <url>
10        <loc>http://www.example.com/products/server</loc>
11    </url>
12    <url>
13        <loc>http://www.example.com/products/server/faq</loc>
14    </url>
15    <url>
16        <loc>http://www.example.com/products/server/editions</loc>
17    </url>
```



```

18 <url>
19   <loc>http://www.example.com/products/server/requirements</loc>
20 </url>
21 <url>
22   <loc>http://www.example.com/products/studio</loc>
23 </url>
24 <url>
25   <loc>http://www.example.com/products/studio/customers</loc>
26 </url>
27 <url>
28   <loc>http://www.example.com/products/studio/support</loc>
29 </url>
30 <url>
31   <loc>http://www.example.com/company/about</loc>
32 </url>
33 <url>
34   <loc>http://www.example.com/company/about/investors</loc>
35 </url>
36 <url>
37   <loc>http://www.example.com/company/news</loc>
38 </url>
39 <url>
40   <loc>http://www.example.com/company/news/press</loc>
41 </url>
42 <url>
43   <loc>http://www.example.com/archive</loc>
44 </url>
45 <url>
46   <loc>http://www.example.com/community</loc>
47 </url>
48 <url>
49   <loc>http://www.example.com/community/account</loc>
50 </url>
51 <url>
52   <loc>http://forums.example.com/</loc>
53 </url>
54 </urlset>

```

Rendering using no ACL role

Render the sitemap using no ACL role (should filter out /community/account):

```

1 echo $this->navigation()
2     ->sitemap()
3     ->setFormatOutput(true)
4     ->setRole();

```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
3   <url>
4     <loc>http://www.example.com/</loc>
5   </url>
6   <url>
7     <loc>http://www.example.com/products</loc>
8   </url>

```

```
9 <url>
10 <loc>http://www.example.com/products/server</loc>
11 </url>
12 <url>
13 <loc>http://www.example.com/products/server/faq</loc>
14 </url>
15 <url>
16 <loc>http://www.example.com/products/server/editions</loc>
17 </url>
18 <url>
19 <loc>http://www.example.com/products/server/requirements</loc>
20 </url>
21 <url>
22 <loc>http://www.example.com/products/studio</loc>
23 </url>
24 <url>
25 <loc>http://www.example.com/products/studio/customers</loc>
26 </url>
27 <url>
28 <loc>http://www.example.com/products/studio/support</loc>
29 </url>
30 <url>
31 <loc>http://www.example.com/company/about</loc>
32 </url>
33 <url>
34 <loc>http://www.example.com/company/about/investors</loc>
35 </url>
36 <url>
37 <loc>http://www.example.com/company/news</loc>
38 </url>
39 <url>
40 <loc>http://www.example.com/company/news/press</loc>
41 </url>
42 <url>
43 <loc>http://www.example.com/archive</loc>
44 </url>
45 <url>
46 <loc>http://www.example.com/community</loc>
47 </url>
48 <url>
49 <loc>http://forums.example.com/</loc>
50 </url>
51 </urlset>
```

Rendering using a maximum depth

Render the sitemap using a maximum depth of 1.

```
1 echo $this->navigation()
2     ->sitemap()
3     ->setFormatOutput(true)
4     ->setMaxDepth(1);
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
```

```
3 <url>
4   <loc>http://www.example.com/</loc>
5 </url>
6 <url>
7   <loc>http://www.example.com/products</loc>
8 </url>
9 <url>
10  <loc>http://www.example.com/products/server</loc>
11 </url>
12 <url>
13  <loc>http://www.example.com/products/studio</loc>
14 </url>
15 <url>
16  <loc>http://www.example.com/company/about</loc>
17 </url>
18 <url>
19  <loc>http://www.example.com/company/about/investors</loc>
20 </url>
21 <url>
22  <loc>http://www.example.com/company/news</loc>
23 </url>
24 <url>
25  <loc>http://www.example.com/community</loc>
26 </url>
27 <url>
28  <loc>http://www.example.com/community/account</loc>
29 </url>
30 <url>
31  <loc>http://forums.example.com/</loc>
32 </url>
33 </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. So if you want to use another encoding with `Sitemap`, you will have to 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 from HeadStyle documentation](#) to see how you can achieve this.

View Helper - Navigation Proxy

Introduction

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.

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.

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

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 233.1: Adapters for `Zend\Paginator`

Adapter	Description
ArrayAdapter	Accepts a PHP array
DbSelect	Accepts a <code>Zend\Db\Sql\Select</code> plus either a <code>Zend\Db\Adapter\Adapter</code> or <code>Zend\Db\Sql\Sql</code> to paginate rows from a database
Iterator	Accepts an Iterator instance
NullFill	Does not use <code>Zend\Paginator</code> 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 `NullFill` 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.

```
1 $paginator->setCurrentPageNumber($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:

```
1 return array(  
2     'routes' => array(  
3         'paginator' => array(  
4             'type' => 'segment',  
5             'options' => array(  
6                 'route' => '/list/[page/:page]',  
7                 'defaults' => array(  
8                     'page' => 1,  
9                 ),  
10            ),  
11        ),  
12    ),  
13 );
```

With the above route (and using Zend Framework *MVC* components), you might set the current page number in your controller action like so:

```
1 $paginator->setCurrentPageNumber($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:

```
1 $vm = new ViewModel();  
2 $vm->setVariable('paginator', $paginator);  
3 return $vm;
```

The DbSelect adapter

The usage of most adapters is pretty straight-forward. However, the DbSelect 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 `count` 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:

```

1 class MyDbSelect extends Zend\Paginator\Adapter\DbSelect
2 {
3     public function count()
4     {
5         $select = new Zend\Db\Sql\Select();
6         $select->from('item_counts')->columns(array('c'=>'post_count'));
7
8         $statement = $this->sql->prepareStatementForSqlObject($select);
9         $result = $statement->execute();
10        $row = $result->current();
11        $this->rowCount = $row['c'];
12
13        return $this->rowCount;
14    }
15 }
16
17 $adapter = new MyDbSelect($query, $adapter);
18 $paginator = new Zend\Paginator\Paginator($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:

```

1 // $objectPrototype is an instance of our custom entity
2 // $hydrator is a custom hydrator for our entity (implementing
3 // Zend\Stdlib\Hydrator\HydratorInterface)
4 $resultSet = new Zend\Db\ResultSet\HydratingResultSet($hydrator, $objectPrototype);
5
6 $adapter = new Zend\Paginator\Adapter\DbSelect($query, $dbAdapter, $resultSet)
7 $paginator = new Zend\Paginator\Paginator($adapter);
    
```

Now when we iterate over \$paginator we will get instances of our custom entity instead of key-value-pair arrays.

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.

```

1 <html>
2 <body>
3 <h1>Example</h1>
4 <?php if (count($this->paginator)): ?>
5 <ul>
6 <?php foreach ($this->paginator as $item): ?>
7     <li><?php echo $item; ?></li>
8 <?php endforeach; ?>
9 </ul>
10 <?php endif; ?>
11
    
```

```

12 <?php echo $this->paginationControl($this->paginator,
13     'Sliding',
14     'my_pagination_control', array('route' =>
15     'application/paginator')); ?>
16 </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:

Results Page:

[Prev](#) ◀ [2](#) [3](#) [4](#) [5](#) [6](#) 7 [8](#) [9](#) [10](#) [11](#) ▶ [Next](#)

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 233.2: 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:

```

1 Zend\Paginator\Paginator::setDefaultScrollingStyle('Sliding');
2 Zend\View\Helper\PaginationControl::setDefaultViewPartial(
3     'my_pagination_control'
4 );

```

When all of these values are set, you can render the pagination control inside your view script with a simple echo statement:

```

1 <?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 }
```

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
-->
```

```

5 <?php if ($this->pageCount): ?>
6 <div class="paginationControl">
7 <?php echo $this->firstItemNumber; ?> - <?php echo $this->lastItemNumber; ?>
8 of <?php echo $this->totalItemCount; ?>
9
10 <!-- First page link -->
11 <?php if (isset($this->previous)): ?>
12     <a href="<?php echo $this->url($this->route, array('page' => $this->first)); ?>">
13         First
14     </a> |
15 <?php else: ?>
16     <span class="disabled">First</span> |
17 <?php endif; ?>
18
19 <!-- Previous page link -->
20 <?php if (isset($this->previous)): ?>
21     <a href="<?php echo $this->url($this->route, array('page' => $this->previous)); ?>">
22         &lt; Previous
23     </a> |
24 <?php else: ?>
25     <span class="disabled">&lt; Previous</span> |
26 <?php endif; ?>
27
28 <!-- Next page link -->
29 <?php if (isset($this->next)): ?>
30     <a href="<?php echo $this->url($this->route, array('page' => $this->next)); ?>">
31         Next &gt;
32     </a> |
33 <?php else: ?>
34     <span class="disabled">Next &gt;</span> |
35 <?php endif; ?>
36
37 <!-- Last page link -->
38 <?php if (isset($this->next)): ?>
39     <a href="<?php echo $this->url($this->route, array('page' => $this->last)); ?>">
40         Last
41     </a>
42 <?php else: ?>
43     <span class="disabled">Last</span>
44 <?php endif; ?>
45
46 </div>
47 <?php endif; ?>
    
```

Dropdown pagination:

```

1 <?php if ($this->pageCount): ?>
2 <select id="paginationControl" size="1">
3 <?php foreach ($this->pagesInRange as $page): ?>
4     <?php $selected = ($page == $this->current) ? ' selected="selected"' : ''; ?>
5     <option value="<?php
6         echo $this->url($this->route, array('page' => $page)); ?>">?php echo $selected_
7         <?php echo $page; ?>
8     </option>
9 <?php endforeach; ?>
10 </select>
11 <?php endif; ?>
    
```

```

12 <script type="text/javascript"
13     src="http://ajax.googleapis.com/ajax/libs/prototype/1.6.0.2/prototype.js">
14 </script>
15 <script type="text/javascript">
16 $('paginationControl').observe('change', function() {
17     window.location = this.options[this.selectedIndex].value;
18 })
19 </script>
20

```

Listing of properties

The following options are available to pagination control view scripts:

Table 233.3: 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

Configuration

`Zend\Paginator` has several configuration methods that can be called:

Table 234.1: Configuration methods for `Zend\Paginator\Paginator`

Method	Description
<code>setCurrentPageNumber</code>	Sets the current page number (default 1).
<code>setItemCountPerPage</code>	Sets the maximum number of items to display on a page (default 10).
<code>setPageRange</code>	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).
<code>setView</code>	Sets the view instance, for rendering convenience.

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.

Custom scrolling styles

Creating your own scrolling style requires that you implement `Zend\Paginator\ScrollingStyle\ScrollingStyleInterface` which defines a single method, `getPages()`. Specifically,

```
1 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:

```
1 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:

```
1 $manager = Zend\Paginator\Paginator::getScrollingStyleManager();
2 $manager->setInvokableClass('my-style', 'My\Paginator\ScrollingStyle');
```

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.

```
1 $cache = StorageFactory::adapterFactory('filesystem', array(
2     'cache_dir' => '/tmp',
3     'ttl'       => 3600,
4     'plugins'   => array( 'serializer' ),
5 ));
6 Zend\Paginator\Paginator::setCache($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.

```
1 // $cache is a Zend\Cache\Storage\StorageInterface instance
2 Zend\Paginator\Paginator::setCache($cache);
3 // ... later on the script
4 $paginator->setCacheEnable(false);
5 // 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:

```
1 // $cache is a Zend\Cache\Storage\StorageInterface instance
2 Zend\Paginator\Paginator::setCache($cache);
3 // $paginator is a fully configured Zend\Paginator\Paginator instance
4 $items = $paginator->getCurrentItems();
5 // page 1 is now in cache
6 $page3Items = $paginator->getItemsByPage(3);
7 // page 3 is now in cache
```

```
8
9 // clear the cache of the results for page 3
10 $paginator->clearPageItemCache(3);
11
12 // clear all the cache data
13 $paginator->clearPageItemCache();
```

Changing the item count per page will empty the whole cache as it would have become invalid:

```
1 // $cache is a Zend\Cache\Storage\StorageInterface instance
2 Zend\Paginator\Paginator::setCache($cache);
3 // fetch some items
4 // $paginator is a fully configured Zend\Paginator\Paginator instance
5 $items = $paginator->getCurrentItems();
6
7 // all the cache data will be flushed:
8 $paginator->setItemCountPerPage(2);
```

It is also possible to see the data in cache and ask for them directly. `getPageItemCache()` can be used for that:

```
1 // $cache is a Zend\Cache\Storage\StorageInterface instance
2 Zend\Paginator\Paginator::setCache($cache);
3 // $paginator is a fully configured Zend\Paginator\Paginator instance
4 $paginator->setItemCountPerPage(3);
5 // fetch some items
6 $items = $paginator->getCurrentItems();
7 $otherItems = $paginator->getItemsPerPage(4);
8
9 // see the cached items as a two-dimension array:
10 var_dump($paginator->getPageItemCache());
```

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.

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.

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, getId(). 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”):

```
1 use Zend\Permissions\Acl\Acl;
2 use Zend\Permissions\Acl\Role\GenericRole as Role;
3 use Zend\Permissions\Acl\Resource\GenericResource as Resource;
4
5 $acl = new Acl();
6
7 $acl->addRole(new Role('guest'))
8     ->addRole(new Role('member'))
9     ->addRole(new Role('admin'));
10
11 $parents = array('guest', 'member', 'admin');
12 $acl->addRole(new Role('someUser'), $parents);
13
14 $acl->addResource(new Resource('someResource'));
15
16 $acl->deny('guest', 'someResource');
17 $acl->allow('member', 'someResource');
18
19 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.

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:

```
1 use Zend\Permissions\Acl\Acl;
2 $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.

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 236.1: 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:

```
1 use Zend\Permissions\Acl\Acl;
2 use Zend\Permissions\Acl\Role\GenericRole as Role;
3
4 $acl = new Acl();
5
```

```
6 // Add groups to the Role registry using Zend\Permissions\Acl\Role\GenericRole
7 // Guest does not inherit access controls
8 $roleGuest = new Role('guest');
9 $acl->addRole($roleGuest);
10
11 // Staff inherits from guest
12 $acl->addRole(new Role('staff'), $roleGuest);
13
14 /*
15 Alternatively, the above could be written:
16 $acl->addRole(new Role('staff'), 'guest');
17 */
18
19 // Editor inherits from staff
20 $acl->addRole(new Role('editor'), 'staff');
21
22 // Administrator does not inherit access controls
23 $acl->addRole(new Role('administrator'));
```

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:

```
1 use Zend\Permissions\Acl\Acl;
2 use Zend\Permissions\Acl\Role\GenericRole as Role;
3
4 $acl = new Acl();
5
6 $roleGuest = new Role('guest');
7 $acl->addRole($roleGuest);
8 $acl->addRole(new Role('staff'), $roleGuest);
9 $acl->addRole(new Role('editor'), 'staff');
10 $acl->addRole(new Role('administrator'));
11
12 // Guest may only view content
13 $acl->allow($roleGuest, null, 'view');
14
15 /*
16 Alternatively, the above could be written:
17 $acl->allow('guest', null, 'view');
18 //*/
19
20 // Staff inherits view privilege from guest, but also needs additional
```

```

21 // privileges
22 $acl->allow('staff', null, array('edit', 'submit', 'revise'));
23
24 // Editor inherits view, edit, submit, and revise privileges from
25 // staff, but also needs additional privileges
26 $acl->allow('editor', null, array('publish', 'archive', 'delete'));
27
28 // Administrator inherits nothing, but is allowed all privileges
29 $acl->allow('administrator');
```

The NULL values in the above allow() calls are used to indicate that the allow rules apply to all resources.

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:

```

1  echo $acl->isAllowed('guest', null, 'view') ?
2      "allowed" : "denied";
3  // allowed
4
5  echo $acl->isAllowed('staff', null, 'publish') ?
6      "allowed" : "denied";
7  // denied
8
9  echo $acl->isAllowed('staff', null, 'revise') ?
10     "allowed" : "denied";
11 // allowed
12
13 echo $acl->isAllowed('editor', null, 'view') ?
14     "allowed" : "denied";
15 // allowed because of inheritance from guest
16
17 echo $acl->isAllowed('editor', null, 'update') ?
18     "allowed" : "denied";
19 // denied because no allow rule for 'update'
20
21 echo $acl->isAllowed('administrator', null, 'view') ?
22     "allowed" : "denied";
23 // allowed because administrator is allowed all privileges
24
25 echo $acl->isAllowed('administrator') ?
26     "allowed" : "denied";
27 // allowed because administrator is allowed all privileges
28
29 echo $acl->isAllowed('administrator', null, 'update') ?
30     "allowed" : "denied";
31 // allowed because administrator is allowed all privileges
```

Refining Access Controls

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 you to 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’:

```
1 // The new marketing group inherits permissions from staff
2 use Zend\Permissions\Acl\Acl;
3 use Zend\Permissions\Acl\Role\GenericRole as Role;
4 use Zend\Permissions\Acl\Resource\GenericResource as Resource;
5
6 $acl = new Acl();
7
8 $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:

```
1 // Create Resources for the rules
2
3 // newsletter
4 $acl->addResource(new Resource('newsletter'));
```

```
5
6 // news
7 $acl->addResource(new Resource('news'));
8
9 // latest news
10 $acl->addResource(new Resource('latest', 'news'));
11
12 // announcement news
13 $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:

```
1 // Marketing must be able to publish and archive newsletters and the
2 // latest news
3 $acl->allow('marketing',
4           array('newsletter', 'latest'),
5           array('publish', 'archive'));
6
7 // Staff (and marketing, by inheritance), are denied permission to
8 // revise the latest news
9 $acl->deny('staff', 'latest', 'revise');
10
11 // Everyone (including administrators) are denied permission to
12 // archive news announcements
13 $acl->deny(null, 'announcement', 'archive');
```

We can now query the ACL with respect to the latest changes:

```
1 echo $acl->isAllowed('staff', 'newsletter', 'publish') ?
2     "allowed" : "denied";
3 // denied
4
5 echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
6     "allowed" : "denied";
7 // allowed
8
9 echo $acl->isAllowed('staff', 'latest', 'publish') ?
10     "allowed" : "denied";
11 // denied
12
13 echo $acl->isAllowed('marketing', 'latest', 'publish') ?
14     "allowed" : "denied";
15 // allowed
16
17 echo $acl->isAllowed('marketing', 'latest', 'archive') ?
18     "allowed" : "denied";
19 // allowed
20
21 echo $acl->isAllowed('marketing', 'latest', 'revise') ?
22     "allowed" : "denied";
23 // denied
24
25 echo $acl->isAllowed('editor', 'announcement', 'archive') ?
26     "allowed" : "denied";
27 // denied
28
29 echo $acl->isAllowed('administrator', 'announcement', 'archive') ?
30     "allowed" : "denied";
```

```
31 // denied
```

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:

```
1 // Remove the denial of revising latest news to staff (and marketing,
2 // by inheritance)
3 $acl->removeDeny('staff', 'latest', 'revise');
4
5 echo $acl->isAllowed('marketing', 'latest', 'revise') ?
6     "allowed" : "denied";
7 // allowed
8
9 // Remove the allowance of publishing and archiving newsletters to
10 // marketing
11 $acl->removeAllow('marketing',
12                 'newsletter',
13                 array('publish', 'archive'));
14
15 echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
16     "allowed" : "denied";
17 // denied
18
19 echo $acl->isAllowed('marketing', 'newsletter', 'archive') ?
20     "allowed" : "denied";
21 // denied
```

Privileges may be modified incrementally as indicated above, but a `NULL` value for the privileges overrides such incremental changes:

```
1 // Allow marketing all permissions upon the latest news
2 $acl->allow('marketing', 'latest');
3
4 echo $acl->isAllowed('marketing', 'latest', 'publish') ?
5     "allowed" : "denied";
6 // allowed
7
8 echo $acl->isAllowed('marketing', 'latest', 'archive') ?
9     "allowed" : "denied";
10 // allowed
11
12 echo $acl->isAllowed('marketing', 'latest', 'anything') ?
13     "allowed" : "denied";
14 // allowed
```


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.

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:

```

1 class CleanIPAssertion implements Zend\Permissions\Acl\Assertion\AssertionInterface
2 {
3     public function assert(Zend\Permissions\Acl\Acl $acl,
4                           Zend\Permissions\Acl\Role\RoleInterface $role = null,
5                           Zend\Permissions\Acl\Resource\ResourceInterface $resource,
6                           $assertion = null,
```

```
6             $privilege = null)
7     {
8         return $this->_isCleanIP($_SERVER['REMOTE_ADDR']);
9     }
10
11     protected function _isCleanIP($ip)
12     {
13         // ...
14     }
15 }
```

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`.

```
1 use Zend\Permissions\Acl\Acl;
2
3 $acl = new Acl();
4 $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.

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.

Roles

The easiest way to create a role is by extending the abstract class `Zend\Permissions\Rbac\AbstractRole` or simply using the default class provided in `Zend\Permissions\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.

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.

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.

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

The following is a list of common use-case examples for `Zend\Permission\Rbac`.

Roles

Extending and adding roles via instantiation.

```
1  <?php
2  use Zend\Permissions\Rbac\Rbac;
3  use Zend\Permissions\Rbac\AbstractRole;
4
5  class MyRole extends AbstractRole
6  {
7      // .. implementation
8  }
9
10 // Creating roles manually
11 $foo = new MyRole('foo');
12
13 $rbac = new Rbac();
14 $rbac->addRole($foo);
15
16 var_dump($rbac->hasRole('foo')); // true
```

Adding roles directly to RBAC with the default `Zend\Permission\Rbac\Role`.

```
1  <?php
2  use Zend\Permissions\Rbac\Rbac;
3
4  $rbac = new Rbac();
5  $rbac->addRole('foo');
6
7  var_dump($rbac->hasRole('foo')); // true
```

Handling roles with children.

```
1 <?php
2 use Zend\Permissions\Rbac\Rbac;
3 use Zend\Permissions\Rbac\Role;
4
5 $rbac = new Rbac();
6 $foo = new Role('foo');
7 $bar = new Role('bar');
8
9 // 1 - Add a role with child role directly with instantiated classes.
10 $foo->addChild($bar);
11 $rbac->addRole($foo);
12
13 // 2 - Same as one, only via rbac container.
14 $rbac->addRole('boo', 'baz'); // baz is a parent of boo
15 $rbac->addRole('baz', array('out', 'of', 'roles')); // create several parents of baz
```

Permissions

```
1 <?php
2 use Zend\Permissions\Rbac\Rbac;
3 use Zend\Permissions\Rbac\Role;
4
5 $rbac = new Rbac();
6 $foo = new Role('foo');
7 $foo->addPermission('bar');
8
9 var_dump($foo->hasPermission('bar')); // true
10
11 $rbac->addRole($foo);
12 $rbac->isGranted('foo', 'bar'); // true
13 $rbac->isGranted('foo', 'baz'); // false
14
15 $rbac->getRole('foo')->addPermission('baz');
16 $rbac->isGranted('foo', 'baz'); // true
```

Dynamic Assertions

Checking permission using `isGranted()` with a class implementing `Zend\Permissions\Rbac\AssertionInterface`.

```
1 <?php
2 use Zend\Permissions\Rbac\AssertionInterface;
3 use Zend\Permissions\Rbac\Rbac;
4
5 class AssertUserIdMatches implements AssertionInterface
6 {
7     protected $userId;
8     protected $article;
9
10     public function __construct($userId)
11     {
12         $this->userId = $userId;
```



```

13     }
14
15     public function setArticle($article)
16     {
17         $this->article = $article;
18     }
19
20     public function assert(Rbac $rbac)
21     {
22         if (!$this->article) {
23             return false;
24         }
25         return $this->userId == $this->article->getUserId();
26     }
27 }
28
29 // User is assigned the foo role with id 5
30 // News article belongs to userId 5
31 // Jazz article belongs to userId 6
32
33 $rbac = new Rbac();
34 $user = $mySessionObject->getUser();
35 $news = $articleService->getArticle(5);
36 $jazz = $articleService->getArticle(6);
37
38 $rbac->addRole($user->getRole());
39 $rbac->getRole($user->getRole())->addPermission('edit.article');
40
41 $assertion = new AssertUserIdMatches($user->getId());
42 $assertion->setArticle($news);
43
44 // true always - bad!
45 if ($rbac->isGranted($user->getRole(), 'edit.article')) {
46     // hacks another user's article
47 }
48
49 // true for user id 5, because he belongs to write group and user id matches
50 if ($rbac->isGranted($user->getRole(), 'edit.article', $assertion)) {
51     // edits his own article
52 }
53
54 $assertion->setArticle($jazz);
55
56 // false for user id 5
57 if ($rbac->isGranted($user->getRole(), 'edit.article', $assertion)) {
58     // can not edit another user's article
59 }

```

Performing the same as above with a Closure.

```

1 <?php
2 // assume same variables from previous example
3
4 $assertion = function($rbac) use ($user, $news) {
5     return $user->getId() == $news->getUserId();
6 };
7
8 // true

```

```
9  if ($rbac->isGranted($user->getRole(), 'edit.article', $assertion)) {  
10     // edits his own article  
11 }
```

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.

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:

```
1 $progressBar = new Zend\ProgressBar\ProgressBar($adapter, 0, $fileSize);
2
3 while (!feof($fp)) {
4     // Do something
5
6     $progressBar->update($currentByteCount);
7 }
8
9 $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.

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.

Standard Adapters

`Zend\ProgressBar` comes with the following three adapters:

- `Zend\ProgressBar\Adapter\Console`
 - `Zend\ProgressBar\Adapter\JsPush`
 - `Zend\ProgressBar\Adapter\JsPull`
- orphan**

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.

orphan

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

```

1 <div id="zend-progressbar-container">
2   <div id="zend-progressbar-done"></div>
3 </div>
4
5 <iframe src="long-running-process.php" id="long-running-process"></iframe>
```

```

1 #long-running-process {
2   position: absolute;
3   left: -100px;
4   top: -100px;
5
6   width: 1px;
7   height: 1px;
8 }
9
10 #zend-progressbar-container {
11   width: 100px;
12   height: 30px;
```

```

13     border: 1px solid #000000;
14     background-color: #ffffff;
15 }
16
17 #zend-progressbar-done {
18     width: 0;
19     height: 30px;
20
21     background-color: #000000;
22 }
23

```

```

1 function Zend\ProgressBar\Update(data)
2 {
3     document.getElementById('zend-progressbar-done').style.width =
4         data.percent + '%';
5 }

```

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: <https://api.jqueryui.com/progressbar/>
- 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.

orphan

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*.

File Upload Handlers

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.

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.

Using a ProgressBar Adapter

A `Zend\ProgressBar` adapter can be used to display upload progress to your users.

```
1 $adapter = new \Zend\ProgressBar\Adapter\JsPush();
2 $progress = new \Zend\ProgressBar\Upload\SessionProgress();
3
4 $filter = new \Zend\I18n\Filter\Alnum(false, 'en-US');
5 $id = $filter->filter($_GET['id']);
6
7 $status = null;
8 while (empty($status['done'])) {
9     $status = $progress->getProgress($id);
10 }
```

Each time the `getProgress()` method is called, the `ProgressBar` adapter will be updated.

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:

```
1 // In a Controller...
2
3 public function sessionProgressAction()
4 {
5     $id = $this->params()->fromQuery('id', null);
6     $progress = new \Zend\ProgressBar\Upload\SessionProgress();
7     return new \Zend\View\Model\JsonModel($progress->getProgress($id));
8 }
9
10 // Returns JSON
11 //{
12 //     "total"    : 204800,
13 //     "current"  : 10240,
14 //     "rate"     : 1024,
15 //     "message"  : "10kB / 200kB",
16 //     "done"     : false
17 //}
```

Standard Handlers

`Zend\ProgressBar\Upload` comes with the following three upload handlers:

- `Zend\ProgressBar\Upload\ApcProgress`
- `Zend\ProgressBar\Upload\SessionProgress`
- `Zend\ProgressBar\Upload\UploadProgress`

orphan

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.

orphan

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.

orphan

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.

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](#).

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.

```
1 use Zend\Serializer\Serializer;
2
3 // Via factory:
4 $serializer = Zend\Serializer\Serializer::factory('PhpSerialize');
5
6 // Alternately:
7 $serializer = new Zend\Serializer\Adapter\PhpSerialize();
8
9 // Now $serializer is an instance of Zend\Serializer\Adapter\AdapterInterface,
10 // specifically Zend\Serializer\Adapter\PhpSerialize
11
12 try {
13     $serialized = $serializer->serialize($data);
14     // now $serialized is a string
15
16     $unserialized = $serializer->unserialize($serialized);
17     // now $data == $unserialized
18 } catch (Zend\Serializer\Exception\ExceptionInterface $e) {
19     echo $e;
20 }
```

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`:

```
1 use Zend\Serializer\Serializer;
2
3 try {
4     $serialized = Serializer::serialize($data);
5     // now $serialized is a string
6
7     $unserialized = Serializer::unserialize($serialized);
8     // now $data == $unserialized
9 } catch (Zend\Serializer\Exception\ExceptionInterface $e) {
10     echo $e;
11 }
```

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.

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` `$adapterName`, *Zend\Serializer\Adapter\AdapterOptions*|*array*|*Traversable* `$adapterOptions` = *null*)

Create a serializer adapter instance.

Return type `Zend\Serializer\Adapter\AdapterInterface`

setAdapterManager (*Zend\Serializer\AdapterPluginManager \$adapters*)

Change the adapter plugin manager.

Return type void

getAdapterManager ()

Get the adapter plugin manager.

Return type Zend\Serializer\AdapterPluginManager

resetAdapterManager ()

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

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.

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.

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.

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 245.1: ZendSerializerAdapterWddx Options

Option	Data Type	Default Value	Description
<code>comment</code>	<code>string</code>		An optional comment that appears in the packet header.

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 245.2: ZendSerializerAdapterJson Options

Option	Data Type	Default Value
<code>cycle_check</code>	<code>boolean</code>	<code>false</code>
<code>object_decode_type</code>	<code>Zend\Json\Json::TYPE_*</code>	<code>Zend\Json\Json::TYPE_ARRAY</code>
<code>enable_json_expr_finder</code>	<code>boolean</code>	<code>false</code>

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 245.3: ZendSerializerAdapterPythonPickle Options

Option	Data Type	Default Value	Description
<code>protocol</code>	<code>integer (0 1 2 3)</code>	<code>0</code>	The Pickle protocol version used on serialize

Table 245.4: Datatype merging (PHP to Python Pickle)

PHP Type	Python Pickle Type
<code>NULL</code>	<code>None</code>
<code>boolean</code>	<code>boolean</code>
<code>integer</code>	<code>integer</code>
<code>float</code>	<code>float</code>
<code>string</code>	<code>string</code>
<code>array list</code>	<code>list</code>
<code>array map</code>	<code>dictionary</code>
<code>object</code>	<code>dictionary</code>

Table 245.5: 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

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.

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.

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.

Usage

Basic usage is simple:

```
1 $class      = Zend\Server\Reflection::reflectClass('My\Class');
2 $function   = Zend\Server\Reflection::reflectFunction('my_function');
3
4 // Get prototypes
5 $prototypes = $function->getPrototypes();
6
7 // Loop through each prototype for the function
8 foreach ($prototypes as $prototype) {
9
10     // Get prototype return type
11     echo "Return type: ", $prototype->getReturnType(), "\n";
12
13     // Get prototype parameters
14     $parameters = $prototype->getParameters();
15
16     echo "Parameters: \n";
17     foreach ($parameters as $parameter) {
18         // Get parameter type
19         echo "      ", $parameter->getType(), "\n";
```

```
20     }
21 }
22
23 // Get namespace for a class, function, or method.
24 // Namespaces may be set at instantiation time (second argument), or using
25 // setNameSpace()
26 $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.

Introduction

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:

```
1 namespace Zend\ServiceManager;
2
3 interface ServiceLocatorInterface
4 {
5     public function get($name);
6     public function has($name);
7 }
```

- `has($name)`, tests whether the `ServiceManager` has a named service;
- `get($name)`, retrieves a service by the given name.

Additional API

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:

```
1 $serviceManager->setService('my-foo', new stdClass());
2 $serviceManager->setService('my-settings', array('password' => 'super-secret'));
3
```

```
4 var_dump($serviceManager->get('my-foo')); // an instance of stdClass
5 var_dump($serviceManager->get('my-settings')); // array('password' => 'super-secret')
```

Invokables (constructor-less services)

`ServiceManager::setInvokableClass` allows you to tell the `ServiceManager` what class to instantiate when a particular service is requested:

```
1 $serviceManager->setInvokableClass('foo-service-name', 'Fully\Qualified\Classname');
2
3 var_dump($serviceManager->get('foo-service-name')); // an instance of
↳Fully\Qualified\Classname
```

Classes registered as invokables must have no constructor defined, or constructors that allow passing no arguments.

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:

```
1 use Zend\ServiceManager\FactoryInterface;
2 use Zend\ServiceManager\ServiceLocatorInterface;
3
4 class MyFactory implements FactoryInterface
5 {
6     public function createService(ServiceLocatorInterface $serviceLocator)
7     {
8         return new \stdClass();
9     }
10 }
11
12 // registering a factory instance
13 $serviceManager->setFactory('foo-service-name', new MyFactory());
14
15 // registering a factory by factory class name
16 $serviceManager->setFactory('bar-service-name', 'MyFactory');
17
18 // registering a callback as a factory
19 $serviceManager->setFactory('baz-service-name', function () { return new \stdClass();
↳});
20
21 var_dump($serviceManager->get('foo-service-name')); // stdClass(1)
22 var_dump($serviceManager->get('bar-service-name')); // stdClass(2)
23 var_dump($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:


```

1 $foo      = new \stdClass();
2 $foo->bar = 'baz!';
3
4 $serviceManager->setService('my-foo', $foo);
5 $serviceManager->setAlias('my-bar', 'my-foo');
6 $serviceManager->setAlias('my-baz', 'my-bar');
7
8 var_dump($serviceManager->get('my-foo')->bar); // baz!
9 var_dump($serviceManager->get('my-bar')->bar); // baz!
10 var_dump($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:

```

1 use Zend\ServiceManager\ServiceLocatorInterface;
2 use Zend\ServiceManager\AbstractFactoryInterface;
3
4 class MyAbstractFactory implements AbstractFactoryInterface
5 {
6     public function canCreateServiceWithName(ServiceLocatorInterface $serviceLocator,
7     ↪ $name, $requestedName)
8     {
9         // this abstract factory only knows about 'foo' and 'bar'
10        return $requestedName === 'foo' || $requestedName === 'bar';
11    }
12
13    public function createServiceWithName(ServiceLocatorInterface $serviceLocator,
14    ↪ $name, $requestedName)
15    {
16        $service = new \stdClass();
17
18        $service->name = $requestedName;
19
20        return $service;
21    }
22 }
23
24 $serviceManager->addAbstractFactory('MyAbstractFactory');
25
26 var_dump($serviceManager->get('foo')->name); // foo
27 var_dump($serviceManager->get('bar')->name); // bar
28 var_dump($serviceManager->get('baz')->name); // exception!
29 ↪ Zend\ServiceManager\Exception\ServiceNotFoundException

```

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** can be either PHP [callbacks](#) or classes implementing `Zend\ServiceManager\InitializerInterface`. They receive the new instance, and can then manipulate it:

```
1 use Zend\ServiceManager\ServiceLocatorInterface;
2 use Zend\ServiceManager\InitializerInterface;
3
4 class MyInitializer implements InitializerInterface
5 {
6     public function initialize($instance, ServiceLocatorInterface $serviceLocator)
7     {
8         if ($instance instanceof \stdClass) {
9             $instance->initialized = 'initialized!';
10        }
11    }
12 }
13
14 $serviceManager->addInitializer('MyInitializer');
15 $serviceManager->setInvokableClass('my-service', 'stdClass');
16
17 var_dump($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.

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`.

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.

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.

Examples

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.

```
1 <?php
2 // a module configuration, "module/SomeModule/config/module.config.php"
3 return array(
4     'service_manager' => array(
5         'abstract_factories' => array(
6             // Valid values include names of classes implementing
7             // AbstractFactoryInterface, instances of classes implementing
8             // AbstractFactoryInterface, or any PHP callbacks
9             'SomeModule\Service\FallbackFactory',
10         ),
11     ),
12 );
```

```

11     'aliases' => array(
12         // Aliasing a FQCN to a service name
13         'SomeModule\Model\User' => 'User',
14         // Aliasing a name to a known service name
15         'AdminUser' => 'User',
16         // Aliasing to an alias
17         'SuperUser' => 'AdminUser',
18     ),
19     'factories' => array(
20         // Keys are the service names.
21         // Valid values include names of classes implementing
22         // FactoryInterface, instances of classes implementing
23         // FactoryInterface, or any PHP callbacks
24         'User' => 'SomeModule\Service\UserFactory',
25         'UserForm' => function ($serviceManager) {
26             $form = new SomeModule\Form\User();
27
28             // Retrieve a dependency from the service manager and inject it!
29             $form->setInputFilter($serviceManager->get('UserInputFilter'));
30             return $form;
31         },
32     ),
33     'invokables' => array(
34         // Keys are the service names
35         // Values are valid class names to instantiate.
36         'UserInputFilter' => 'SomeModule\InputFilter\User',
37     ),
38     'services' => array(
39         // Keys are the service names
40         // Values are objects
41         'Auth' => new SomeModule\Authentication\AuthenticationService(),
42     ),
43     'shared' => array(
44         // Usually, you'll only indicate services that should **NOT** be
45         // shared -- i.e., ones where you want a different instance
46         // every time.
47         'UserForm' => false,
48     ),
49 ),
50 );

```

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.

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.

```
1 namespace SomeModule;
2
3 // you may eventually want to implement
4 ↪ Zend\ModuleManager\Feature\ServiceProviderInterface
5 class Module
6 {
7     public function getServiceConfig()
8     {
9         return array(
10             'abstract_factories' => array(),
11             'aliases' => array(),
12             'factories' => array(),
13             'invokables' => array(),
14             'services' => array(),
15             'shared' => array(),
16         );
17     }
18 }
```

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:

```
1 namespace SomeModule\Controller;
2
3 use Zend\ServiceManager\ServiceLocatorAwareInterface;
4 use Zend\ServiceManager\ServiceLocatorInterface;
5 use Zend\Stdlib\DispatchableInterface as Dispatchable;
6 use Zend\Stdlib\RequestInterface as Request;
7 use Zend\Stdlib\ResponseInterface as Response;
8
9 class BareController implements
10     Dispatchable,
11     ServiceLocatorAwareInterface
12 {
13     protected $services;
14 }
```

```
15     public function setServiceLocator(ServiceLocatorInterface $serviceLocator)
16     {
17         $this->services = $serviceLocator;
18     }
19
20     public function getServiceLocator()
21     {
22         return $this->services;
23     }
24
25     public function dispatch(Request $request, Response $response = null)
26     {
27         // ...
28
29         // Retrieve something from the service manager
30         $router = $this->getServiceLocator()->get('Router');
31
32         // ...
33     }
34 }
```

Delegator service factories

`Zend\ServiceManager` can instantiate **delegators** of requested services, decorating them as specified in a delegate factory implementing the **delegator factory interface**.

The delegate pattern is useful in cases when you want to wrap a real service in a **decorator**, or generally intercept actions being performed on the delegate in an **AOP** fashioned way.

Delegator factory signature

A delegator factory has the following signature:

```
1 namespace Zend\ServiceManager;
2
3 interface DelegatorFactoryInterface
4 {
5     public function createDelegatorWithName(ServiceLocatorInterface $serviceLocator,
6     ↪ $name, $requestedName, $callback);
7 }
```

The parameters passed to the `DelegatorFactoryInterface#createDelegatorWithName` factory are the following:

- `$serviceLocator` is the service locator that is used while creating the delegator for the requested service
- `$name` is the canonical name of the service being requested
- `$requestedName` is the name of the service as originally requested to the service locator
- `$callback` is a **callable** that is responsible of instantiating the delegated service (the real service instance)

A Delegator factory use case

A typical use case for **delegators** is to handle logic before or after a method is called.

In the following example, an event is being triggered before `Buzzer::buzz()` is called and some output text is prepended.

The delegated object `Buzzer` (original object) is defined as following:

```
1 class Buzzer
2 {
3     public function buzz()
4     {
5         return 'Buzz!';
6     }
7 }
```

The delegator class `BuzzerDelegator` has the following structure:

```
1 use Zend\EventManager\EventManagerInterface;
2
3 class BuzzerDelegator extends Buzzer
4 {
5     protected $realBuzzer;
6     protected $eventManager;
7
8     public function __construct(Buzzer $realBuzzer, EventManagerInterface
9     ↪ $eventManager)
10    {
11        $this->realBuzzer = $realBuzzer;
12        $this->eventManager = $eventManager;
13    }
14
15    public function buzz()
16    {
17        $this->eventManager->trigger('buzz', $this);
18
19        return $this->realBuzzer->buzz();
20    }
21 }
```

To use the `BuzzerDelegator`, you can run the following code:

```
1 $wrappedBuzzer = new Buzzer();
2 $eventManager = new Zend\EventManager\EventManager();
3
4 $eventManager->attach('buzz', function () { echo "Stare at the art!\n"; });
5
6 $buzzer = new BuzzerDelegator($wrappedBuzzer, $eventManager);
7
8 echo $buzzer->buzz(); // "Stare at the art!\nBuzz!"
```

This logic is fairly simple as long as you have access to the instantiation logic of the `$wrappedBuzzer` object.

You may not always be able to define how `$wrappedBuzzer` is created, since a factory for it may be defined by some code to which you don't have access, or which you cannot modify without introducing further complexity.

Delegator factories solve this specific problem by allowing you to wrap, decorate or modify any existing service.

A simple delegator factory for the `buzzer` service can be implemented as following:

```
1 use Zend\ServiceManager\DelegatorFactoryInterface;
2 use Zend\ServiceManager\ServiceLocatorInterface;
3
```

```

4 class BuzzerDelegatorFactory implements DelegatorFactoryInterface
5 {
6     public function createDelegatorWithName(ServiceLocatorInterface $serviceLocator,
7     ↪ $name, $requestedName, $callback)
8     {
9         $realBuzzer = call_user_func($callback);
10        $eventManager = $serviceLocator->get('EventManager');
11
12        $eventManager->attach('buzz', function () { echo "Stare at the art!\n"; });
13
14        return new BuzzerDelegator($realBuzzer, $eventManager);
15    }
16 }

```

You can then instruct the service manager to handle the service buzzer as a delegate:

```

1 $serviceManager = new Zend\ServiceManager\ServiceManager();
2
3 $serviceManager->setInvokableClass('buzzer', 'Buzzer'); // usually not under our
4 ↪ control
5
6 // as opposed to normal factory classes, a delegator factory is a
7 // service like any other, and must be registered:
8 $serviceManager->setInvokableClass('buzzer-delegator-factory', 'BuzzerDelegatorFactory
9 ↪ ');
10
11 // telling the service manager to use a delegator factory to handle service 'buzzer'
12 $serviceManager->addDelegator('buzzer', 'buzzer-delegator-factory');
13
14 // now, when fetching 'buzzer', we get a BuzzerDelegator instead
15 $buzzer = $serviceManager->get('buzzer');
16
17 $buzzer->buzz(); // "Stare at the art!\nBuzz!"

```

You can also call `$serviceManager->addDelegator()` multiple times, with the same or different delegator factory service names. Each call will add one decorator around the instantiation logic of that particular service.

Another way of configuring the service manager to use delegator factories is via configuration:

```

1 $config = array(
2     'invokables' => array(
3         'buzzer' => 'Buzzer',
4         'buzzer-delegator-factory' => 'BuzzerDelegatorFactory',
5     ),
6     'delegators' => array(
7         'buzzer' => array(
8             'buzzer-delegator-factory'
9             // eventually add more delegators here
10        ),
11    ),
12 );

```

Lazy Services

Zend\ServiceManager can use *delegator factories* to generate “lazy” references to your services.

Lazy services are *proxies* that get lazily instantiated, and keep a reference to the real instance of the proxied service.

Use cases

You may want to lazily initialize a service when it is instantiated very often, but not always used.

A typical example is a database connection: it is a dependency to many other elements in your application, but that doesn’t mean that every request will execute queries through it.

Additionally, instantiating a connection to the database may require some time and eat up resources.

Proxying the database connection would allow to delay that overhead until the object is really needed.

Setup

Zend\ServiceManager\Proxy\LazyServiceFactory is a *delegator factory* capable of generating lazy loading proxies for your services.

The LazyServiceFactory depends on *ProxyManager*, so be sure to install it before going through the following steps:

```
1 php composer.phar require ocradius/proxy-manager:0.3.*
```

Practical example

To demonstrate how a lazy service works, you may use the following *Buzzer* example class, which is designed to be slow at instantiation time for demonstration purposes:

```
1 namespace MyApp;
2
3 class Buzzer
4 {
5     public function __construct()
6     {
7         // deliberately halting the application for 5 seconds
8         sleep(5);
9     }
10
11     public function buzz()
12     {
13         return 'Buzz!';
14     }
15 }
```

You can then proceed and configure the service manager to generate proxies instead of real services:

```
1 $serviceManager = new \Zend\ServiceManager\ServiceManager();
2
3 $config = array(
4     'lazy_services' => array(
5         // mapping services to their class names is required
6         // since the ServiceManager is not a declarative DIC
7         'class_map' => array(
8             'buzzer' => 'MyApp\Buzzer',
9         ),
10    ),
11 );
12
13 $serviceManager->setService('Config', $config);
14 $serviceManager->setInvokableClass('buzzer', 'MyApp\Buzzer');
15 $serviceManager->setFactory('LazyServiceFactory',
16     ↪ '\Zend\ServiceManager\Proxy\LazyServiceFactoryFactory');
17 $serviceManager->addDelegator('buzzer', 'LazyServiceFactory');
```

This will tell the service manager to use the LazyServiceFactory delegator factory to instantiate the buzzer service.

As you may have noticed, the standard setup for the LazyServiceFactory requires you to define a Config service. That's because the functionality was thought to be easily integrated into Zend\Mvc.

You can now simply retrieve the buzzer:

```
1 $buzzer = $serviceManager->get('buzzer');
2
3 echo $buzzer->buzz();
```

To verify that the proxying occurred correctly, you can simply run the following code, which should delay the 5 seconds wait time hardcoded in Buzzer::__construct until Buzzer::buzz is invoked:

```
1 for ($i = 0; $i < 100; $i += 1) {
2     $buzzer = $serviceManager->create('buzzer');
3
4     echo "created buzzer $i\n";
5 }
6
7 echo $buzzer->buzz();
```

The setup above can also be represented via configuration in an MVC application's context:

```

1 return array(
2     'service_manager' => array(
3         'invokables' => array(
4             'buzzer' => 'MyApp\Buzzer',
5         ),
6         'delegators' => array(
7             'buzzer' => array(
8                 'LazyServiceFactory'
9             ),
10        ),
11        'factories' => array(
12            'LazyServiceFactory' =>
13            ↪ 'Zend\ServiceManager\Proxy\LazyServiceFactoryFactory',
14        ),
15        'lazy_services' => array(
16            'class_map' => array(
17                'buzzer' => 'MyApp\Buzzer',
18            ),
19        ),
20    );

```

Configuration

This is the config structure expected by `Zend\ServiceManager\Proxy\LazyServiceFactoryFactory`:

```

1 return array(
2     'lazy_services' => array(
3
4         // map of service names and their relative class names - this
5         // is required since the service manager cannot know the
6         // class name of defined services upfront
7         'class_map' => array(
8             // 'foo' => 'MyApplication\Foo',
9         ),
10
11        // directory where proxy classes will be written - default to system_get_tmp_
12        ↪ dir()
13        'proxies_target_dir' => null,
14
15        // namespace of the generated proxies, default to "ProxyManagerGeneratedProxy"
16        'proxies_namespace' => null,
17
18        // whether the generated proxy classes should be written to disk or
19        ↪ generated on-the-fly
20        'write_proxy_files' => false,
21    ),
22);

```


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`.

orphan

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.

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.
remember_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.

Basic Usage

A basic example is one like the following:

```

1 use Zend\Session\Config\StandardConfig;
2 use Zend\Session\SessionManager;
3
4 $config = new StandardConfig();
5 $config->setOptions(array(
6     'remember_me_seconds' => 1800,
7     'name'                 => 'zf2',
8 ));
9 $manager = new SessionManager($config);

```

orphan

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.

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.

Basic Usage

A basic example is one like the following:

```

1 use Zend\Session\Config\SessionConfig;
2 use Zend\Session\SessionManager;
3
4 $config = new SessionConfig();
5 $config->setOptions(array(
6     'phpSaveHandler' => 'redis',
7     'savePath' => 'tcp://127.0.0.1:6379?weight=1&timeout=1',
8 ));
9 $manager = new SessionManager($config);
    
```

Service Manager Factory

Zend\Session ships with a Service Manager factory which reads configuration data from the application configuration and injects a corresponding instance of Zend\Session\Config\SessionConfig into the session manager automatically.

To use this factory, you first need to register it with the Service Manager by adding the appropriate factory definition:

```

1 'service_manager' => array(
2     'factories' => array(
3         'Zend\Session\Config\ConfigInterface' =>
4         ↪ 'Zend\Session\Service\SessionConfigFactory',
5     ),
6 ),
    
```

Then place your application's session configuration in the root-level configuration key `session_config`:

```

1 'session_config' => array(
2     'phpSaveHandler' => 'redis',
3     'savePath' => 'tcp://127.0.0.1:6379?weight=1&timeout=1',
4 ),
    
```

Any of the configuration options defined in `zend.session.config.session-config.options` can be used there, as well as the following factory-specific configuration options:

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.

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`.

Basic Usage

```
1 use Zend\Session\Container;
2
3 $container = new Container('namespace');
4 $container->item = 'foo';
```

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.

```
1 use Zend\Session\Container;
2 use Zend\Session\SessionManager;
3
4 $manager = new SessionManager();
5 Container::setDefaultManager($manager);
```

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.

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:

```
1 return array(  
2     'session' => array(  
3         'config' => array(  
4             'class' => 'Zend\Session\Config\SessionConfig',  
5             'options' => array(  
6                 'name' => 'myapp',  
7             ),  
8         ),  
9         'storage' => 'Zend\Session\Storage\SessionArrayStorage',  
10        'validators' => array(  
11            'Zend\Session\Validator\RemoteAddr',  
12            'Zend\Session\Validator\HttpUserAgent',  
13        ),  
14    ),  
15 );
```

The following illustrates how you might utilize the above configuration to create the session manager:

```

1  use Zend\Session\SessionManager;
2  use Zend\Session\Container;
3
4  class Module
5  {
6      public function onBootstrap($e)
7      {
8          $eventManager      = $e->getApplication()->getEventManager();
9          $moduleRouteListener = new ModuleRouteListener();
10         $moduleRouteListener->attach($eventManager);
11         $this->bootstrapSession($e);
12     }
13
14     public function bootstrapSession($e)
15     {
16         $session = $e->getApplication()
17             ->getServiceManager()
18             ->get('Zend\Session\SessionManager');
19         $session->start();
20
21         $container = new Container('initialized');
22         if (!isset($container->init)) {
23             $serviceManager = $e->getApplication()->getServiceManager();
24             $request         = $serviceManager->get('Request');
25
26             $session->regenerateId(true);
27             $container->init      = 1;
28             $container->remoteAddr = $request->getServer()->get('REMOTE_ADDR');
29             $container->httpUserAgent = $request->getServer()->get('HTTP_USER_AGENT');
30
31             $config = $serviceManager->get('Config');
32             if (!isset($config['session'])) {
33                 return;
34             }
35
36             $sessionConfig = $config['session'];
37             if (isset($sessionConfig['validators'])) {
38                 $chain = $session->getValidatorChain();
39
40                 foreach ($sessionConfig['validators'] as $validator) {
41                     switch ($validator) {
42                         case 'Zend\Session\Validator\HttpUserAgent':
43                             $validator = new $validator($container->httpUserAgent);
44                             break;
45                         case 'Zend\Session\Validator\RemoteAddr':
46                             $validator = new $validator($container->remoteAddr);
47                             break;
48                         default:
49                             $validator = new $validator();
50                     }
51
52                     $chain->attach('session.validate', array($validator, 'isValid'));
53                 }
54             }
55         }
56     }
57 }

```



```

58     public function getServiceConfig()
59     {
60         return array(
61             'factories' => array(
62                 'Zend\Session\SessionManager' => function ($sm) {
63                     $config = $sm->get('config');
64                     if (isset($config['session'])) {
65                         $session = $config['session'];
66
67                         $sessionConfig = null;
68                         if (isset($session['config'])) {
69                             $class = isset($session['config']['class']) ? $session[
↪ 'config']['class'] : 'Zend\Session\Config\SessionConfig';
70                             $options = isset($session['config']['options']) ?
↪ $session['config']['options'] : array();
71                             $sessionConfig = new $class();
72                             $sessionConfig->setOptions($options);
73                         }
74
75                         $sessionStorage = null;
76                         if (isset($session['storage'])) {
77                             $class = $session['storage'];
78                             $sessionStorage = new $class();
79                         }
80
81                         $sessionSaveHandler = null;
82                         if (isset($session['save_handler'])) {
83                             // class should be fetched from service manager since it
↪ will require constructor arguments
84                             $sessionSaveHandler = $sm->get($session['save_handler']);
85                         }
86
87                         $sessionManager = new SessionManager($sessionConfig,
↪ $sessionStorage, $sessionSaveHandler);
88                     } else {
89                         $sessionManager = new SessionManager();
90                     }
91                     Container::setDefaultManager($sessionManager);
92                     return $sessionManager;
93                 },
94             ),
95         );
96     }
97 }
    
```

When you create a new `Zend\Session\Container` (see [Session Container](#) page) in a controller for example, it will use the session configured above.

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.

```
1 $cwd = getcwd();
2 chdir('/path/to/zf2-application');
3 require 'init_autoloader.php';
4 Zend\Mvc\Application::init(require 'config/application.config.php');
5 chdir($cwd);
6 session_start();
```

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`.

orphan

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.

Basic usage

A basic example is one like the following:

```
1 use Zend\Cache\StorageFactory;
2 use Zend\Session\SaveHandler\Cache;
3 use Zend\Session\SessionManager;
4
5 $cache = StorageFactory::factory(array(
6     'adapter' => array(
7         'name' => 'memcached',
8         'options' => array(
9             'server' => '127.0.0.1',
10        ),
11    ),
12 ));
13 $saveHandler = new Cache($cache);
14 $manager = new SessionManager();
15 $manager->setSaveHandler($saveHandler);
```

orphan

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.

Creating the database table

```
1 CREATE TABLE `session` (  
2     `id` char(32),  
3     `name` char(32),  
4     `modified` int,  
5     `lifetime` int,  
6     `data` text,  
7     PRIMARY KEY (`id`, `name`)  
8 );
```

Basic usage

A basic example is one like the following:

```
1 use Zend\Db\TableGateway\TableGateway;  
2 use Zend\Session\SaveHandler\DbTableGateway;  
3 use Zend\Session\SaveHandler\DbTableGatewayOptions;  
4 use Zend\Session\SessionManager;  
5  
6 $tableGateway = new TableGateway('session', $adapter);  
7 $saveHandler  = new DbTableGateway($tableGateway, new DbTableGatewayOptions());  
8 $manager      = new SessionManager();  
9 $manager->setSaveHandler($saveHandler);
```

orphan

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.

Basic Usage

A basic example is one like the following:

```
1 use Mongo;  
2 use Zend\Session\SaveHandler\MongoDB;  
3 use Zend\Session\SaveHandler\MongoDBOptions;  
4 use Zend\Session\SessionManager;
```

```
5
6 $mongo = new Mongo();
7 $options = new MongoClientOptions(array(
8     'database' => 'myapp',
9     'collection' => 'sessions',
10 ));
11 $saveHandler = new MongoDB($mongo, $options);
12 $manager     = new SessionManager();
13 $manager->setSaveHandler($saveHandler);
```

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.

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`.

orphan

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.

Basic Usage

A basic example is one like the following:

```
1 use Zend\Session\Storage\ArrayStorage;
2 use Zend\Session\SessionManager;
3
4 $populateStorage = array('foo' => 'bar');
5 $storage         = new ArrayStorage($populateStorage);
6 $manager         = new SessionManager();
7 $manager->setStorage($storage);
```

orphan

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.

Basic Usage

A basic example is one like the following:

```
1 use Zend\Session\Storage\SessionStorage;
2 use Zend\Session\SessionManager;
3
4 $manager = new SessionManager();
5 $manager->setStorage(new SessionStorage());
```

orphan

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.

Basic Usage

A basic example is one like the following:

```
1 use Zend\Session\Storage\SessionArrayStorage;
2 use Zend\Session\SessionManager;
3
4 $manager = new SessionManager();
5 $manager->setStorage(new SessionArrayStorage());
```

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.

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 browser's user agent may change during the request either by a web browser extension OR an upgrade that retains session cookies.

orphan

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.

Basic Usage

A basic example is one like the following:

```
1 use Zend\Session\Validator\HttpUserAgent;
2 use Zend\Session\SessionManager;
3
4 $manager = new SessionManager();
5 $manager->getValidatorChain()->attach('session.validate', array(new HttpUserAgent(),
    ↪ 'isValid'));
```

orphan

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.

Basic Usage

A basic example is one like the following:

```
1 use Zend\Session\Validator\RemoteAddr;
2 use Zend\Session\SessionManager;
3
4 $manager = new SessionManager();
5 $manager->getValidatorChain()->attach('session.validate', array(new RemoteAddr(),
    ↪ 'isValid'));
```

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`.

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\Wsd 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.

Zend\Soap\Server constructor

Zend\Soap\Server constructor should be used a bit differently for WSDL and non-WSDL modes.

Zend\Soap\Server constructor for the WSDL mode

Zend\Soap\Server constructor takes two optional parameters when it works in WSDL mode:

- `$wsdl`, which is an *URI* of a WSDL file¹.
- `$options`- options to create *SOAP* server object².

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.

¹ May be set later using `setWsd($wsdl)` method.

² Options may be set later using `setOptions($options)` method.

- ‘encoding’ - internal character encoding (UTF-8 is always used as an external encoding).
- ‘wsdl’ which is equivalent to `setWsdL($wsdlValue)` call.

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 object³.

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.

Methods to define Web Service API

There are two ways to define Web Service *API* when you 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*:

```
1  ...
2  class MyClass {
3      /**
4       * This method takes ...
5       *
6       * @param integer $inputParam
7       * @return string
8       */
9      public function method1($inputParam) {
10         ...
11     }
12
13     /**
14      * This method takes ...
15      *
16      * @param integer $inputParam1
17      * @param string $inputParam2
18      * @return float
19      */
20     public function method2($inputParam1, $inputParam2) {
21         ...
22     }
23
24     ...
```

³ Options may be set later using `setOptions($options)` method.

```
25 }
26 ...
27 $server = new Zend\Soap\Server(null, $options);
28 // Bind Class to Soap Server
29 $server->setClass('MyClass');
30 // Bind already initialized object to Soap Server
31 $server->setObject(new MyClass());
32 ...
33 $server->handle();
```

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:

```
1  ...
2  /**
3   * This function ...
4   *
5   * @param integer $inputParam
6   * @return string
7   */
8  function function1($inputParam) {
9      ...
10 }
11
12 /**
13 * This function ...
14 *
15 * @param integer $inputParam1
16 * @param string $inputParam2
17 * @return float
18 */
19 function function2($inputParam1, $inputParam2) {
20     ...
21 }
22 ...
23 $server = new Zend\Soap\Server(null, $options);
24 $server->addFunction('function1');
25 $server->addFunction('function2');
26 ...
27 $server->handle();
```

Request and response objects handling

Note: 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.

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:

```
1 ...
2 $server = new Zend\Soap\Server(...);
3 ...
4 // Set request using optional $request parameter
5 $server->handle($request);
6 ...
7 // Set request using setRequest() method
8 $server->setRequest();
9 $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:

```
1 ...
2 $server = new Zend\Soap\Server(...);
3 ...
4 $server->handle();
5 $request = $server->getLastRequest();
```

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 parameter⁴. Generated response is returned by `handle()` method in this case. Returned response can be a string or a `SoapFault` exception object.

Caution: Always check the returned response type to avoid returning `SoapFault` object as a string, which will be returned to the customer as a string with the exception stacktrace.

```
1 ...
2 $server = new Zend\Soap\Server(...);
3 ...
4 // Get a response as a return value of handle() method
5 // instead of emitting it to the standard output
6 $server->setReturnResponse(true);
7 ...
8 $response = $server->handle();
9 if ($response instanceof \SoapFault) {
```

⁴ Current state of the Return Response flag may be requested with `setReturnResponse()` method.

```

10     ...
11 } else {
12     ...
13 }
14 ...

```

Last response may be also retrieved by `getLastResponse()` method for some post-processing:

```

1 ...
2 $server = new Zend\Soap\Server(...);
3 ...
4 $server->handle();
5 $response = $server->getLastResponse();
6 if ($response instanceof \SoapFault) {
7     ...
8 } else {
9     ...
10 }
11 ...

```

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.

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.

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.

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:

```
1 // Accept response compression
2 $client = new Zend\Soap\Client("some.wsdl",
3     array('compression' => SOAP_COMPRESSION_ACCEPT));
4 ...
5
6 // Compress requests using gzip with compression level 5
7 $client = new Zend\Soap\Client("some.wsdl",
8     array('compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_GZIP | 5));
9 ...
10
11 // Compress requests using deflate compression
12 $client = new Zend\Soap\Client("some.wsdl",
13     array('compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_DEFLATE));
```

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:

```
1 //*****
2 //                               Server code
3 //*****
4 // class MyClass {
5 //     /**
6 //         * This method takes ...
7 //         *
8 //         * @param integer $inputParam
9 //         * @return string
10 //         */
11 //     public function method1($inputParam) {
12 //         ...
13 //     }
14 // }
15 //     /**
```

```
16 //      * This method takes ...
17 //      *
18 //      * @param integer $inputParam1
19 //      * @param string  $inputParam2
20 //      * @return float
21 //      */
22 //      public function method2($inputParam1, $inputParam2) {
23 //          ...
24 //      }
25 //
26 //      ...
27 // }
28 // ...
29 // $server = new Zend\Soap\Server(null, $options);
30 // $server->setClass('MyClass');
31 // ...
32 // $server->handle();
33 //
34 //*****
35 //      End of server code
36 //*****
37
38 $client = new Zend\Soap\Client("MyService.wsdl");
39 ...
40
41 // $result1 is a string
42 $result1 = $client->method1(10);
43 ...
44
45 // $result2 is a float
46 $result2 = $client->method2(22, 'some string');
```


CHAPTER 260

WSDL Accessor

Note: `Zend\Soap\Wsd1` 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\Wsd1` 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.

Zend\Soap\Wsd1 constructor

`Zend\Soap\Wsd1` 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).

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\Wsd1::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.

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.

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.

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.

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.

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.

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 *SOAPAction* 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.

addService() method

`addService($name, $port_name, $binding, $location)` method adds '`/definitions/service`' element to the WSDL document with the specified Web 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\Wsd 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.

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.

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.

Retrieving type information

`getType($type)` method may be used to get mapping for a specified *PHP* type:

```

1  ...
2  $wsdl = new Zend\Soap\WsdL('My_Web_Service', $myWebServiceUri);
3
4  ...
5  $soapIntType = $wsdl->getType('int');
6
7  ...
8  class MyClass {
9      ...

```

¹ 'tns:' namespace is defined as script URI ('<http://>' . \$_SERVER['HTTP_HOST'] . \$_SERVER['SCRIPT_NAME']).

² '<http://>' . \$_SERVER['HTTP_HOST'] . \$_SERVER['SCRIPT_NAME']

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.


```

10 }
11 ...
12 $soapMyClassType = $wsdl->getType('MyClass');

```

Adding complex type information

`addComplexType($type)` method is used to add complex types (PHP classes) to a WSDL document.

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\Wsd1\ComplexTypeStrategy` implementation as the third parameter of the constructor or using the `setComplexTypeStrategy($strategy)` function of `Zend\Soap\Wsd1`. The following detection strategies currently exist:

- Class `Zend\Soap\Wsd1\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\Wsd1\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\Wsd1\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\Wsd1\ComplexTypeStrategy\ArrayOfTypeComplex`: This strategy allows to detect very complex arrays of objects. Objects types are detected based on the `Zend\Soap\Wsd1\Strategy\DefaultComplexType` and an array is wrapped around that definition.
- Class `Zend\Soap\Wsd1\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\Wsd1\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.

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.

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.

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.

```

1 class MySoapServerClass {
2     ...
3 }
4
5 $autodiscover = new Zend\Soap\AutoDiscover();
6 $autodiscover->setClass('MySoapServerClass')
7     ->setUri('http://localhost/server.php')
8     ->setServiceName('MySoapService');
9 $wsdl = $autodiscover->generate();
10 echo $wsdl->toXml();
11 $wsdl->dump("/path/to/file.wsdl");
12 $dom = $wsdl->toDomDocument();

```

Note: Zend\Soap\AutoDiscover 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.

```

1 if (isset($_GET['wsdl'])) {
2     $autodiscover = new Zend\Soap\AutoDiscover();
3     $autodiscover->setClass('HelloWorldService')
4         ->setUri('http://example.com/soap.php');
5     echo $autodiscover->toXml();
6 } else {
7     // pointing to the current file here
8     $soap = new Zend\Soap\Server("http://example.com/soap.php?wsdl");
9     $soap->setClass('HelloWorldService');
10    $soap->handle();
11 }

```

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:

```

1 $autodiscover = new Zend\Soap\AutoDiscover();
2 $autodiscover->setClass('My_SoapServer_Class')
3     ->setUri('http://localhost/server.php')
4     ->setServiceName('MySoapService');
5 $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.

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:

```

1 $autodiscover = new Zend\Soap\AutoDiscover();
2 $autodiscover->addFunction('function1');
3 $autodiscover->addFunction('function2');
4 $autodiscover->addFunction('function3');
5 ...
6 $wsdl = $autodiscover->generate();

```

The same rules apply to generation as described in the class autodiscover section above.

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.

¹ `Zend\Soap\AutoDiscover` will be created with the `Zend\Soap\Wsd1\ComplexTypeStrategy\DefaultComplexType` class as detection algorithm for complex types. The first parameter of the `AutoDiscover` constructor takes any complex type strategy implementing `Zend\Soap\Wsd1\ComplexTypeStrategy\ComplexTypeStrategyInterface` or a string with the name of the class. See the [Zend\Soap\Wsd1 manual on adding complex types](#) for more information.

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 *setClass* or *addFunction* method on the *AutoDiscover* class.

```
1 $autodiscover = new Zend\Soap\AutoDiscover();
2 // Default is 'use' => 'encoded' and
3 // 'encodingStyle' => 'http://schemas.xmlsoap.org/soap/encoding/'
4 $autodiscover->setOperationBodyStyle(
5     array('use' => 'literal',
6           'namespace' => 'http://framework.zend.com')
7 );
8
9 // Default is 'style' => 'rpc' and
10 // 'transport' => 'http://schemas.xmlsoap.org/soap/http'
11 $autodiscover->setBindingStyle(
12     array('style' => 'document',
13           'transport' => 'http://framework.zend.com')
14 );
15 ...
16 $autodiscover->addFunction('myfunc1');
17 $wsdl = $autodiscover->generate();
```

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.

HydratorInterface

```
1 namespace Zend\Stdlib\Hydrator;
2
3 interface HydratorInterface
4 {
5     /**
6      * Extract values from an object
7      *
8      * @param object $object
9      * @return array
10     */
11     public function extract($object);
12
13     /**
14      * Hydrate $object with the provided $data.
15      *
16      * @param array $data
17      * @param object $object
18      * @return object
19     */
20     public function hydrate(array $data, $object);
21 }
```

Usage

Usage is quite simple: simply instantiate the hydrator, and then pass information to it.

```
1 use Zend\Stdlib\Hydrator;
2 $hydrator = new Hydrator\ArraySerializable();
3
4 $object = new ArrayObject(array());
5
6 $hydrator->hydrate($someData, $object);
7
8 // or, if the object has data we want as an array:
9 $data = $hydrator->extract($object);
```

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\DelegatingHydrator**

Composes a hydrator locator containing hydrators and will delegate `hydrate()` and `extract()` calls to the appropriate one based upon the class name of the object being operated on.

```
1 // Instantiate each hydrator you wish to delegate to
2 $albumHydrator = new Zend\Stdlib\Hydrator\ClassMethods;
3 $artistHydrator = new Zend\Stdlib\Hydrator\ClassMethods;
4
5 // Map the entity class name to the hydrator using the HydratorPluginManager
6 // In this case we have two entity classes, "Album" and "Artist"
7 $hydrators = new Zend\Stdlib\Hydrator\HydratorPluginManager;
8 $hydrators->setService('Album', $albumHydrator);
9 $hydrators->setService('Artist', $artistHydrator);
10
11 // Create the DelegatingHydrator and tell it to use our configured hydrator locator
12 $delegating = new Zend\Stdlib\Hydrator\DelegatingHydrator($hydrators);
13
14 // Now we can use $delegating to hydrate or extract any supported object
15 $array = $delegating->extract(new Artist);
16 $artist = $delegating->hydrate($data, new Artist);
```

- **Zend\Stdlib\Hydrator\ObjectProperty**

Any data key matching a publicly accessible property will be hydrated; any public properties will be used for extraction.

- **Zend\Stdlib\Hydrator\Reflection**

Similar to the `ObjectProperty` hydrator, but uses [PHP's reflection API](#) to hydrate or extract properties of any visibility. Any data key matching an existing property will be hydrated; any existing properties will be used for extraction.

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.

```
1 namespace Zend\Stdlib\Hydrator\Filter;
2
3 interface FilterInterface
4 {
5     /**
6      * Should return true, if the given filter
7      * does not match
8      *
9      * @param string $property The name of the property
10     * @return bool
11     */
12     public function filter($property);
13 }
```

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.

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

Remove filters

If you want to tell e.g. the `ClassMethods` hydrator, to not extract methods that start with `is`, you can do so:

```
1 $hydrator = new ClassMethods(false);
2 $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:

```
1 $this->filterComposite->addFilter("is", new IsFilter());
2 $this->filterComposite->addFilter("has", new HasFilter());
3 $this->filterComposite->addFilter("get", new GetFilter());
4 $this->filterComposite->addFilter("parameter", new NumberOfParameterFilter(),
  ↳FilterComposite::CONDITION_AND);
```

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`

Add filters

You can easily add filters to any hydrator, that extends the `AbstractHydrator`. You can use the `FilterInterface` or any callable:

```
1 $hydrator->addFilter("len", function($property) {
2     if (strlen($property) !== 3) {
3         return false;
4     }
5     return true;
6 });
```

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:

```
1 $hydrator->addFilter("len", function($property) {
2     if (strlen($property) !== 3) {
3         return false;
4     }
5     return true;
6 }, FilterComposite::CONDITION_AND);
```

Or you can add the shipped ones:

```
1 $hydrator->addFilter(
2     "servicemanager",
3     new MethodMatchFilter("getServiceManager"),
4     FilterComposite::CONDITION_AND
5 );
```

The example above will exclude the `getServiceManager` method or the key from the extraction, even if the `get` filter wants to add it.

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:

```
1 $composite = new FilterComposite();
2
3 $composite->addFilter("one", $condition1);
4 $composite->addFilter("two", $condition2);
5 $composite->addFilter("three", $condition3);
6 $composite->addFilter("four", $condition4, FilterComposite::CONDITION_AND);
7 $composite->addFilter("five", $condition5, FilterComposite::CONDITION_AND);
8
9 // This is what's happening internally
10 if (
11     (
12         $condition1
13         || $condition2
14         || $condition3
15     ) && (
16         $condition4
17         && $condition5
18     )
19 )
```

```
19     ) {  
20         //do extraction  
21     }
```

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:

```
1  $composite = new FilterComposite();  
2  $composite->addFilter(  
3      "servicemanager",  
4      new MethodMatchFilter("getServiceManager"),  
5      FilterComposite::CONDITION_AND  
6  );  
7  $composite->addFilter(  
8      "eventmanager",  
9      new MethodMatchFilter("getEventManager"),  
10     FilterComposite::CONDITION_AND  
11 );  
12  
13 $hydrator->addFilter("excludes", $composite, FilterComposite::CONDITION_AND);  
14  
15 // Internal  
16 if (  
17     ( // default composite inside the hydrator  
18         (  
19             $getFilter  
20             || $hasFilter  
21             || $isFilter  
22         ) && (  
23             $numberOfParameterFilter  
24         )  
25     ) && ( // new composite, added to the one above  
26         $serviceManagerFilter  
27         && $eventManagerFilter  
28     )  
29 ) {  
30     // do extraction  
31 }
```

If you perform this on the ClassMethods hydrator, all getters will get extracted, but not getServiceManager and getEventManager.

Using the provider interface

There is also a provider interface, that allows you to configure the behavior of the hydrator inside your objects.

```
1  namespace Zend\Stdlib\Hydrator\Filter;  
2  
3  interface FilterProviderInterface  
4  {  
5      /**  
6       * Provides a filter for hydration  
7       *  
8       * @return FilterInterface  
9       */
```

```

10     public function getFilter();
11 }

```

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:

```

1  Class Foo implements FilterProviderInterface
2  {
3      public function getFoo()
4      {
5          return "foo";
6      }
7
8      public function hasFoo()
9      {
10         return true;
11     }
12
13     public function getServiceManager()
14     {
15         return "servicemanager";
16     }
17
18     public function getEventManager()
19     {
20         return "eventmanager";
21     }
22
23     public function getFilter()
24     {
25         $composite = new FilterComposite();
26         $composite->addFilter("get", new GetFilter());
27
28         $exclusionComposite = new FilterComposite();
29         $exclusionComposite->addFilter(
30             "servicemanager",
31             new MethodMatchFilter("getServiceManager"),
32             FilterComposite::CONDITION_AND
33         );
34         $exclusionComposite->addFilter(
35             "eventmanager",
36             new MethodMatchFilter("getEventManager"),
37             FilterComposite::CONDITION_AND
38         );
39
40         $composite->addFilter("excludes", $exclusionComposite,
41     ↪FilterComposite::CONDITION_AND);
42
43         return $composite;
44     }
45
46     $hydrator = new ClassMethods(false);
47     $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.

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\StrategyEnabledInterface`) 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:

```
1 namespace Zend\Stdlib\Hydrator\Strategy;
2
3 interface StrategyInterface
4 {
5     /**
6      * Converts the given value so that it can be extracted by the hydrator.
7      *
8      * @param mixed $value The original value.
9      * @return mixed Returns the value that should be extracted.
10    */
11    public function extract($value);
12
13    /**
14     * Converts the given value so that it can be hydrated by the hydrator.
15     *
16     * @param mixed $value The original value.
17     * @return mixed Returns the value that should be hydrated.
18    */
19    public function hydrate($value);
20 }
```

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()`.

Adding strategies to the hydrators

To allow strategies within your hydrator, the `Zend\Stdlib\Hydrator\Strategy\StrategyEnabledInterface` provide the following methods:

```
1 namespace Zend\Stdlib\Hydrator;
2
3 use Zend\Stdlib\Hydrator\Strategy\StrategyInterface;
4
5 interface StrategyEnabledInterface
6 {
7     /**
8      * Adds the given strategy under the given name.
9      *
10     * @param string $name The name of the strategy to register.
11     * @param StrategyInterface $strategy The strategy to register.
12     * @return HydratorInterface
13     */
14     public function addStrategy($name, StrategyInterface $strategy);
15
16     /**
17     * Gets the strategy with the given name.
18     *
19     * @param string $name The name of the strategy to get.
20     * @return StrategyInterface
21     */
22     public function getStrategy($name);
23
24     /**
25     * Checks if the strategy with the given name exists.
26     *
27     * @param string $name The name of the strategy to check for.
28     * @return bool
29     */
30     public function hasStrategy($name);
31
32     /**
33     * Removes the strategy with the given name.
34     *
35     * @param string $name The name of the strategy to remove.
36     * @return HydratorInterface
37     */
38     public function removeStrategy($name);
39 }
```

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`.

Available implementations

- **Zend\Stdlib\Hydrator\Strategy\SerializableStrategy**

This is a 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\ClosureStrategy**

This is a strategy that allows you to pass in a `hydrate` callback to be called in the event of hydration, and an `extract` callback to be called in the event of extraction.

- **Zend\Stdlib\Hydrator\Strategy\DefaultStrategy**

This is a kind of dummy-implementation, that simply proxies everything through, without doing anything on the parameters.

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()`:

```

1  class Rot13Strategy implements StrategyInterface
2  {
3      public function extract($value)
4      {
5          return str_rot13($value);
6      }
7
8      public function hydrate($value)
9      {
10         return str_rot13($value);
11     }
12 }
```

This is the example class, we want to use for the hydrator example:

```

1  class Foo
2  {
3      protected $foo = null;
4      protected $bar = null;
5
6      public function getFoo()
7      {
8          return $this->foo;
9      }
10
11     public function setFoo($foo)
12     {
13         $this->foo = $foo;
14     }
15
16     public function getBar()
17     {
18         return $this->bar;
19     }
20
21     public function setBar($bar)
22     {
23         $this->bar = $bar;
24     }
25 }
```

Now, we want to add the rot13 strategy to the method `getFoo()` and `setFoo($foo)`:

```
1 $foo = new Foo();
2 $foo->setFoo("bar");
3 $foo->setBar("foo");
4
5 $hydrator = new ClassMethods();
6 $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:

```
1 $extractedArray = $hydrator->extract($foo);
2
3 // array(2) {
4 //     ["foo"]=>
5 //     string(3) "one"
6 //     ["bar"]=>
7 //     string(3) "foo"
8 // }
```

And the the way back:

```
1 $hydrator->hydrate($extractedArray, $foo)
2
3 // object(Foo)#2 (2) {
4 //     ["foo":protected]=>
5 //     string(3) "bar"
6 //     ["bar":protected]=>
7 //     string(3) "foo"
8 // }
```

AggregateHydrator

`Zend\Stdlib\Hydrator\Aggregate\AggregateHydrator` is an implementation of `Zend\Stdlib\Hydrator\HydratorInterface` that composes multiple hydrators via event listeners.

You typically want to use an aggregate hydrator when you want to hydrate or extract data from complex objects that implement multiple interfaces, and therefore need multiple hydrators to handle that in subsequent steps.

Installation requirements

The `AggregateHydrator` is based on the `Zend\EventManager` component, so be sure to have it installed before getting started:

```
php composer.phar require zendframework/zend-eventmanager:2.*
```

Basic usage

A simple use case may be hydrating a `BlogPost` object, which contains data for the user that created it, the time it was created, the current publishing status, etc:

```
1 use Zend\Stdlib\Hydrator\Aggregate\AggregateHydrator;
2
3 $hydrator = new AggregateHydrator();
4
5 // attach the various hydrators capable of handling simpler interfaces
6 $hydrator->add(new My\BlogPostHydrator());
7 $hydrator->add(new My\UserAwareObjectHydrator());
8 $hydrator->add(new My\TimestampedObjectHydrator());
9 $hydrator->add(new My\PublishableObjectHydrator());
10 // ...
11
12 // Now retrieve the BlogPost object
```

```
13 // ...
14
15 // you can now extract complex data from a blogpost
16 $data = $hydrator->extract($blogPost);
17
18 // or you can fill the object with complex data
19 $blogPost = $hydrator->hydrate($data, $blogPost);
```

Note: Hydrator priorities

`AggregateHydrator::add` has a second optional argument `$priority`. If you have two or more hydrators that conflict with each other for same data keys, you may decide which one has to be executed first or last by passing a higher or lower integer priority to the second argument of `AggregateHydrator::add`

In order to work with this logic, each of the hydrators that are attached should just ignore any unknown object type passed in, such as in following example:

```
1 namespace My;
2
3 use Zend\Stdlib\Hydrator\HydratorInterface
4
5 class BlogPostHydrator implements HydratorInterface
6 {
7     public function hydrate($data, $object)
8     {
9         if (!$object instanceof BlogPost) {
10             return $object;
11         }
12
13         // ... continue hydration ...
14     }
15
16     public function extract($object)
17     {
18         if (!$object instanceof BlogPost) {
19             return array();
20         }
21
22         // ... continue extraction ...
23     }
24 }
```

Advanced use cases

Since the `AggregateHydrator` is event-driven, you can use the `EventManager` API to tweak its behaviour.

Common use cases are:

- Removal of hydrated data keys (passwords/confidential information) depending on business rules
- Caching of the hydration/extraction process
- Transformations on extracted data, for compatibility with third-party APIs

In the following example, a cache listener will be introduced to speed up hydration, which can be very useful when the same data is requested multiple times:

```

1 use Zend\Stdlib\Hydrator\Aggregate\AggregateHydrator;
2 use Zend\Stdlib\Hydrator\Aggregate\ExtractEvent;
3 use Zend\Cache\Storage\Adapter\Memory;
4
5 $hydrator = new AggregateHydrator();
6
7 // attach the various hydrators
8 $hydrator->add(new My\BlogPostHydrator());
9 $hydrator->add(new My\UserAwareObjectHydrator());
10 $hydrator->add(new My\TimestampedObjectHydrator());
11 $hydrator->add(new My\PublishableObjectHydrator());
12 // ...
13
14 $cache          = new Memory();
15 $cacheReadListener = function (ExtractEvent $event) use ($cache) {
16     $object = $event->getExtractionObject();
17
18     if (!$object instanceof BlogPost) {
19         return;
20     }
21
22     if ($cache->hasItem($object->getId())) {
23         $event->setExtractedData($cache->getItem($object->getId()));
24         $event->stopPropagation();
25     }
26 };
27 $cacheWriteListener = function (ExtractEvent $event) use ($cache) {
28     $object = $event->getExtractionObject();
29
30     if (!$object instanceof BlogPost) {
31         return;
32     }
33
34     $cache->setItem($object->getId(), $event->getExtractedData());
35 };
36
37 // attaching a high priority listener executed before extraction logic
38 $hydrator->getEventManager()->attach(ExtractEvent::EVENT_EXTRACT, $cacheReadListener,
39     ↪ 1000);
40 // attaching a low priority listener executed after extraction logic
41 $hydrator->getEventManager()->attach(ExtractEvent::EVENT_EXTRACT, $cacheWriteListener,
42     ↪ -1000);
    
```

With an aggregate hydrator configured in this way, any `$hydrator->extract($blogPost)` operation will be cached

CompositeNamingStrategy

`Zend\Stdlib\Hydrator\NamingStrategy\CompositeNamingStrategy` allows you to specify which naming strategy should be used for each key encountered during hydration or extraction.

Basic Usage

When invoked, the following composite strategy will extract the property `bar` into array key `foo` (`MapNamingStrategy`) and property `barBat` into `bar_bat` (`UnderscoreNamingStrategy`):

```
1  class Foo
2  {
3      public $bar;
4      public $barBat;
5  }
6
7  $mapStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\MapNamingStrategy([
8      'foo' => 'bar'
9  ]);
10
11  $underscoreNamingStrategy = new
12  ↪ Zend\Stdlib\Hydrator\NamingStrategy\UnderscoreNamingStrategy();
13
14  $namingStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\CompositeNamingStrategy([
15      'bar' => $mapStrategy,
16      'barBat' => $underscoreNamingStrategy,
17  ]);
18
19  $hydrator = new Zend\Stdlib\Hydrator\ObjectProperty();
20  $hydrator->setNamingStrategy($namingStrategy);
21
22  $foo = new Foo();
23  $foo->bar = 123;
24  $foo->barBat = 42;
```

```
25 print_r($foo); // Foo Object ( [bar] => 123 [barBat] => 42 )
26 print_r($hydrator->extract($foo)); // Array ( [foo] => 123 [bar_bat] => 42 )
```

Unfortunately, this CompositeNamingStrategy can only be used for extraction as it will not know how to handle the keys necessary for hydration (foo and bar_bat, respectively). To rectify this we have to cover the keys for both hydration and extraction in our composite strategy:

```
1  class Foo
2  {
3      public $bar;
4      public $barBat;
5  }
6
7  $mapStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\MapNamingStrategy([
8      'foo' => 'bar'
9  ]);
10
11 $underscoreNamingStrategy = new
↳ Zend\Stdlib\Hydrator\NamingStrategy\UnderscoreNamingStrategy();
12
13 $namingStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\CompositeNamingStrategy([
14     // Define both directions for the foo => bar mapping
15     'bar' => $mapStrategy,
16     'foo' => $mapStrategy,
17     // Define both directions for the barBat => bar_bat mapping
18     'barBat' => $underscoreNamingStrategy,
19     'bar_bat' => $underscoreNamingStrategy,
20 ]);
21
22 $hydrator = new Zend\Stdlib\Hydrator\ObjectProperty();
23 $hydrator->setNamingStrategy($namingStrategy);
24
25 $foo = new Foo();
26 $foo->bar = 123;
27 $foo->barBat = 42;
28
29 $array = $hydrator->extract($foo);
30
31 print_r($foo); // Foo Object ( [bar] => 123 [barBat] => 42 )
32 print_r($array); // Array ( [foo] => 123 [bar_bat] => 42 )
33
34 $foo2 = new Foo();
35 $hydrator->hydrate($array, $foo2);
36
37 print_r($foo2); // Foo Object ( [bar] => 123 [barBat] => 42 )
```

IdentityNamingStrategy

`Zend\Stdlib\Hydrator\NamingStrategy\IdentityNamingStrategy` Will use the key it is given for hydration and extraction.

Basic Usage

```
1  $namingStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\IdentityNamingStrategy();
2
3  echo $namingStrategy->hydrate('foo'); // outputs: foo
4  echo $namingStrategy->extract('bar'); // outputs: bar
```

This strategy can be used in hydrators as well:

```
1  class Foo
2  {
3      public $foo;
4  }
5
6  $namingStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\IdentityNamingStrategy();
7  $hydrator = new Zend\Stdlib\Hydrator\ObjectProperty();
8  $hydrator->setNamingStrategy($namingStrategy);
9
10 $foo = new Foo();
11 $hydrator->hydrate(array('foo' => 123), $foo);
12
13 print_r($foo); // Foo Object ( [foo] => 123 )
14 print_r($hydrator->extract($foo)); // Array ( [foo] => 123 )
```

MapNamingStrategy

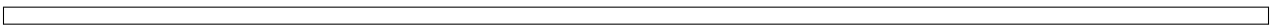
Zend\Stdlib\Hydrator\NamingStrategy\MapNamingStrategy Maps keys based on a given map.

Basic Usage

```
1  $namingStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\MapNamingStrategy(array(  
2      'foo' => 'bar',  
3      'baz' => 'bash'  
4  ));  
5  echo $namingStrategy->hydrate('foo'); // outputs: bar  
6  echo $namingStrategy->hydrate('baz'); // outputs: bash  
7  
8  echo $namingStrategy->extract('bar'); // outputs: foo  
9  echo $namingStrategy->extract('bash'); // outputs: baz
```

This strategy can be used in hydrators to dictate how keys should be mapped:

```
1  class Foo  
2  {  
3      public $bar;  
4  }  
5  
6  $namingStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\MapNamingStrategy(array(  
7      'foo' => 'bar',  
8      'baz' => 'bash'  
9  ));  
10 $hydrator = new Zend\Stdlib\Hydrator\ObjectProperty();  
11 $hydrator->setNamingStrategy($namingStrategy);  
12  
13 $foo = new Foo();  
14 $hydrator->hydrate(array('foo' => 123), $foo);  
15  
16 print_r($foo); // Foo Object ( [bar] => 123 )  
17 print_r($hydrator->extract($foo)); // Array ( [foo] => 123 )
```



UnderscoreNamingStrategy

Zend\Stdlib\Hydrator\NamingStrategy\UnderscoreNamingStrategy Converts snake case strings (e.g. foo_bar_baz) to studly case strings (e.g. fooBarBaz) and vice versa.

Basic Usage

```

1  $namingStrategy = new Zend\Stdlib\Hydrator\NamingStrategy\UnderscoreNamingStrategy();
2
3  echo $namingStrategy->extract('foo_bar'); // outputs: foo_bar
4  echo $namingStrategy->extract('Foo_Bar'); // outputs: foo_bar
5  echo $namingStrategy->extract('FooBar'); // outputs: foo_bar
6
7  echo $namingStrategy->hydrate('fooBar'); // outputs: foo_bar
8  echo $namingStrategy->hydrate('FooBar'); // outputs: foo_bar
9  echo $namingStrategy->hydrate('Foo_Bar'); // outputs: foo_bar

```

This strategy can be used in hydrators to dictate how keys should be mapped.

```

1  class Foo
2  {
3      public $fooBar;
4  }
5
6  $hydrator = new Zend\Stdlib\Hydrator\ObjectProperty();
7  $hydrator->setNamingStrategy(new
↳ Zend\Stdlib\Hydrator\NamingStrategy\UnderscoreNamingStrategy());
8
9  $foo = new Foo();
10 $hydrator->hydrate(array('foo_bar' => 123), $foo);
11
12 print_r($foo); // Foo Object ( [fooBar] => 123 )
13 print_r($hydrator->extract($foo)); // Array ( [foo_bar] => 123 )

```

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.

```
1 // Create the item list
2 $list = new Zend\Tag\ItemList();
3
4 // Assign tags to it
5 $list[] = new Zend\Tag\Item(array('title' => 'Code', 'weight' => 50));
6 $list[] = new Zend\Tag\Item(array('title' => 'Zend Framework', 'weight' => 1));
7 $list[] = new Zend\Tag\Item(array('title' => 'PHP', 'weight' => 5));
8
9 // Spread absolute values on the items
10 $list->spreadWeightValues(array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
11
12 // Output the items with their absolute values
13 foreach ($list as $item) {
14     printf("%s: %d\n", $item->getTitle(), $item->getParam('weightValue'));
15 }
```

This will output the three items “Code”, “Zend Framework” and “PHP” with the absolute values 10, 1 and 2.

```
1 Code: 10
2 Zend Framework: 1
3 PHP: 2
```

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 271.1: 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 <code>decorator</code> and optionally an array under the key <code>options</code> , 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 <code>decorator</code> and optionally an array under the key <code>options</code> , 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.

```

1 // Create the cloud and assign static tags to it
2 $cloud = new Zend\Tag\Cloud(array(
3     'tags' => array(
4         array(
5             'title' => 'Code',
6             'weight' => 50,
7             'params' => array('url' => '/tag/code'),
8         ),
9         array(
10            'title' => 'Zend Framework',
11            'weight' => 1,
12            'params' => array('url' => '/tag/zend-framework'),
13        ),
14        array(
15            'title' => 'PHP',
16            'weight' => 5,
17            'params' => array('url' => '/tag/php'),
18        ),
19    ),
20 ));
21
22 // Render the cloud
23 echo $cloud;

```

This will output the tag cloud with the three tags, spread with the default font-sizes:

```

1 <ul class="zend-tag-cloud">
2     <li>
3         <a href="/tag/code" style="font-size: 20px;">
4             Code
5         </a>
6     </li>
7     <li>
8         <a href="/tag/zend-framework" style="font-size: 10px;">
9             Zend Framework
10        </a>
11    </li>
12    <li>
13        <a href="/tag/php" style="font-size: 11px;">
14            PHP
15        </a>
16    </li>
17 </ul>

```

Note: The HTML code examples are preformatted for a better visualization in the documentation.

You can define a output separator for the *HTML Cloud decorator*.

The following example shows how create the **same** tag cloud from a `Zend\Config\Config` object.

```

1 # An example tags.ini file
2 tags.1.title = "Code"
3 tags.1.weight = 50
4 tags.1.params.url = "/tag/code"
5 tags.2.title = "Zend Framework"
6 tags.2.weight = 1

```

```

7 tags.2.params.url = "/tag/zend-framework"
8 tags.3.title = "PHP"
9 tags.3.weight = 2
10 tags.3.params.url = "/tag/php"

1 // Create the cloud from a Zend\Config\Config object
2 $config = Zend\Config\Factory::fromFile('tags.ini');
3 $cloud = new Zend\Tag\Cloud($config);
4
5 // Render the cloud
6 echo $cloud;

```

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.

HTML Tag decorator

The *HTML* tag decorator will by default render every tag in an anchor element, surrounded by a `` 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 surrounds 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 271.2: HTML Tag decorator Options

Option	Default	Description
fontSizeUnit	Unit	Defines the font-size unit used for all font-sizes. The possible values are: em, ex, px, in, cm, mm, pt, pc and %.
minFontSize	12	The minimum font-size distributed through the tags (must be numeric).
maxFontSize	22	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 <i>HTML</i> 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.

```

1 $cloud = new Zend\Tag\Cloud(array(
2     'tagDecorator' => array(
3         'decorator' => 'htmltag',
4         'options'   => array(

```

```
5         'minFontSize' => '20',
6         'maxFontSize' => '50',
7         'htmlTags'     => array(
8             'li' => array('class' => 'my_custom_class'),
9         ),
10    ),
11),
12'tags' => array(
13    array(
14        'title'  => 'Code',
15        'weight' => 50,
16        'params' => array('url' => '/tag/code'),
17    ),
18    array(
19        'title'  => 'Zend Framework',
20        'weight' => 1,
21        'params' => array('url' => '/tag/zend-framework'),
22    ),
23    array(
24        'title'  => 'PHP',
25        'weight' => 5,
26        'params' => array('url' => '/tag/php')
27    ),
28),
29));
30
31// Render the cloud
32echo $cloud;
```

The output:

```
1 <ul class="zend-tag-cloud">
2   <li class="my_custom_class">
3     <a href="/tag/code" style="font-size: 50px;">Code</a>
4   </li>
5   <li class="my_custom_class">
6     <a href="/tag/zend-framework" style="font-size: 20px;">Zend Framework</a>
7   </li>
8   <li class="my_custom_class">
9     <a href="/tag/php" style="font-size: 23px;">PHP</a>
10  </li>
11 </ul>
```

HTML Cloud decorator

By default the *HTML* cloud decorator will surround the *HTML* tags with a `` 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 271.3: HTML Cloud decorator Options

Option	Default	Description
separator	<code>array(' ')</code> (a whitespace)	Defines the separator which is placed between all tags.
htmlTags	<code>array('ul' => array('class' => 'zend-tag-cloud'))</code>	An array of <i>HTML</i> 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.

```

1 // Create the cloud and assign static tags to it
2 $cloud = new Zend\Tag\Cloud(array(
3     'cloudDecorator' => array(
4         'decorator' => 'htmlcloud',
5         'options' => array(
6             'separator' => "\n\n",
7             'htmlTags' => array(
8                 'ul' => array(
9                     'class' => 'my_custom_class',
10                    'id' => 'tag-cloud',
11                ),
12            ),
13        ),
14    ),
15    'tags' => array(
16        array(
17            'title' => 'Code',
18            'weight' => 50,
19            'params' => array('url' => '/tag/code'),
20        ),
21        array(
22            'title' => 'Zend Framework',
23            'weight' => 1,
24            'params' => array('url' => '/tag/zend-framework'),
25        ),
26        array(
27            'title' => 'PHP',
28            'weight' => 5,
29            'params' => array('url' => '/tag/php'),
30        ),
31    ),
32 ));
33
34 // Render the cloud
35 echo $cloud;

```

The output:

```

1 <ul class="my_custom_class" id="tag-cloud"><li><a href="/tag/code" style="font-size: 20px;">Code</a></li>
2
3 <li><a href="/tag/zend-framework" style="font-size: 10px;">Zend Framework</a></li>
4
5 <li><a href="/tag/php" style="font-size: 11px;">PHP</a></li></ul>

```


CHAPTER 272

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.

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 :

```
1  <?php
2
3  namespace ApplicationTest\Controller;
4
5  use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;
6
7  class IndexControllerTest extends AbstractHttpControllerTestCase
8  {
9      public function setUp()
10     {
11         $this->setApplicationConfig(
12             include '/path/to/application/config/test/application.config.php'
13         );
14         parent::setUp();
15     }
16
17     public function testIndexActionCanBeAccessed()
18     {
19         $this->dispatch('/');
20         $this->assertResponseStatusCode(200);
21
22         $this->assertModuleName('application');
23         $this->assertControllerName('application_index');
24         $this->assertControllerClass('IndexController');
25         $this->assertMatchedRouteName('home');
26     }
27 }
```

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.

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 :

```
1 public function setUp()  
2 {  
3     $this->setApplicationConfig(  
4         include '/path/to/application/config/test/application.config.php'  
5     );  
6     parent::setUp();  
7 }
```

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 :

```
1 <?php  
2  
3 namespace ApplicationTest\Controller;  
4  
5 use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;  
6  
7 class IndexControllerTest extends AbstractHttpControllerTestCase  
8 {  
9     protected $traceError = true;  
10 }
```

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`:

```
1 public function testIndexAction()  
2 {  
3     $this->dispatch('/');  
4 }
```

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:

```
1 public function testIndexAction()  
2 {  
3     $this->getRequest()  
4         ->setMethod('POST')  
5         ->setPost(new Parameters(array('argument' => 'value')));  
6     $this->dispatch('/');  
7 }
```

You can populate GET or POST variables directly with the `dispatch` method :

```

1 public function testIndexAction()
2 {
3     $this->dispatch('/', 'POST', array('argument' => 'value'));
4 }

```

You can use directly yours query args in the url :

```

1 public function testIndexAction()
2 {
3     $this->dispatch('/tests?foo=bar&baz=foo');
4 }

```

Now that the request is made, it's time to start making assertions against it.

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.

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.

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.)

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)`

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.

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.

- `assertResponseStatusCode($code)`: assert that the response resulted in the given HTTP response code.
- `assertResponseHeader($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.

Introduction

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.

Basic Usage

This example illustrates the basic use of `Zend\Text\Figlet` to create a simple FIGlet text:

```
1 $figlet = new Zend\Text\Figlet\Figlet();
2 echo $figlet->render('Zend');
```

Assuming you are using a monospace font, this would look as follows:

1				
2	_ _ _ //	_ _ _	_ \	_ _ _ \ \
3	/ / //	_	'	\
4	/ / / _	_	.	\
5	/ _ _ _	_ _ _	_ \	_ _ _ //
6	/ _ _ _ \	_ _ _ \	_ \	_ _ _ \

Introduction

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 with 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.`

Basic Usage

This example illustrates the basic use of `Zend\Text\Table` to create a simple table:

```
1 $table = new Zend\Text\Table\Table(array('columnWidths' => array(10, 20)));
2
3 // Either simple
4 $table->appendRow(array('Zend', 'Framework'));
5
6 // Or verbose
7 $row = new Zend\Text\Table\Row();
8
9 $row->appendColumn(new Zend\Text\Table\Column('Zend'));
10 $row->appendColumn(new Zend\Text\Table\Column('Framework'));
11
12 $table->appendRow($row);
13
14 echo $table;
```

This will result in the following output:

```
1 -----
2 |Zend          |Framework          |
3 |-----|-----|
4 |Zend          |Framework          |
5 |-----|-----|
```

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.

URIs 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.

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()

```
1 // To create a new URI from scratch, pass only the scheme
2 // followed by a colon.
3 $uri = Zend\Uri\UriFactory::factory('http:');
4
5 // $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.

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

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.

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.

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

```
1 // Create a new 'ftp' URI based on a custom class
2 use Zend\Uri\UriFactory
3
4 UriFactory::registerScheme('ftp', 'MyNamespace\MyClass');
5
6 $ftpUri = UriFactory::factory(
7     'ftp://user@ftp.example.com/path/file'
8 );
9
10 // $ftpUri is an instance of MyLibrary\MyClass, which implements
11 // Zend\Uri\UriInterface
```

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()`

```
1 // To manipulate an existing URI, pass it in.
2 $uri = Zend\Uri\UriFactory::factory('http://www.zend.com');
3
4 // $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.

Common Instance Methods

The `Zend\Uri\UriInterface` defines several instance methods that are useful for working with any kind of *URI*.

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

```
1 $uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');
2
3 $scheme = $uri->getScheme(); // "mailto"
```

The `getScheme()` instance method returns only the scheme part of the *URI* object.

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

```
1 $uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');
2
3 $scheme = $uri->getUserinfo(); // "john.doe"
```

The `getUserinfo()` method returns only the userinfo part of the *URI* object.

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

```
1 $uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');
2
3 $scheme = $uri->getHost(); // "example.com"
```

The `getHost()` method returns only the host part of the *URI* object.

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

```
1 $uri = Zend\Uri\UriFactory::factory('http://example.com:8080');
2
3 $scheme = $uri->getPort(); // "8080"
```

Getting a default port from a Zend\Uri\UriInterface Object

```
1 $uri = Zend\Uri\UriFactory::factory('http://example.com');
2
3 $scheme = $uri->getPort(); // "80"
```

The `getHost()` method returns only the port part of the *URI* object.

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

```
1 $uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');
2
3 $scheme = $uri->getPath(); // "/my/path"
```

The `getPath()` method returns only the path of the *URI* object.

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

```
1 $uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');
2
3 $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

```
1 $uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');
2
3 $scheme = $uri->getQueryAsArray();
4 // array(
5 //     'a' => 'b',
```

```

6 // 'c' => 'd',
7 // )

```

The query-part often contains key=value pairs and therefore can be split into an associative array. This array can be retrieved using `getQueryAsArray()`

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

```

1 $uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');
2
3 $scheme = $uri->getFragment(); // "token"

```

The `getFragment()` method returns only the fragment-part of the *URI* object.

Getting the Entire URI

Getting the Entire URI from a Zend\Uri\UriInterface Object

```

1 $uri = Zend\Uri\UriFactory::factory('http://www.zend.com');
2
3 echo $uri->toString(); // "http://www.zend.com"
4
5 // Alternate method:
6 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.

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

```

1 $uri = Zend\Uri\UriFactory::factory('http://www.zend.com');
2
3 $isValid = $uri->isValid(); // TRUE

```

The `isValid()` instance method provides a means to check that the *URI* object is still valid.

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.

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.

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:

```
1 $validator = new Zend\Validator\EmailAddress();
2
3 if ($validator->isValid($email)) {
4     // email appears to be valid
5 } else {
6     // email is invalid; print the reasons
7     foreach ($validator->getMessages() as $messageId => $message) {
8         echo "Validation failure '$messageId': $message\n";
9     }
10 }
```

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.

```
1 $validator = new Zend\Validator\StringLength(8);
2
3 $validator->setMessage(
4     'The string \'%value%\' is too short; it must be at least %min% ' .
5     'characters',
6     Zend\Validator\StringLength::TOO_SHORT);
7
8 if (!$validator->isValid('word')) {
9     $messages = $validator->getMessages();
10    echo current($messages);
11
12    // "The string 'word' is too short; it must be at least 8 characters"
13 }
```

You can set multiple messages using the `setMessages()` method. Its argument is an array containing key/message pairs.

```
1 $validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
2
3 $validator->setMessages( array(
4     Zend\Validator\StringLength::TOO_SHORT =>
5         'The string \'%value%\' is too short',
6     Zend\Validator\StringLength::TOO_LONG =>
```

```

7     'The string \'%value%\' is too long'
8 ));

```

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.

```

1 $validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
2
3 if (!$validator->isValid('word')) {
4     echo 'Word failed: '
5         . $validator->value
6         . '; its length is not between '
7         . $validator->min
8         . ' and '
9         . $validator->max
10        . "\n";
11 }

```

Translating messages

Note: In versions 2.0 - 2.1, `Zend\Validator\AbstractValidator` implemented `Zend\I18n\Translator\TranslatorAwareInterface` and accepted instances of `Zend\I18n\Translator\Translator`. Starting in version 2.2.0, `Zend\Validator` now defines a translator interface, `Zend\Validator\Translator\TranslatorInterface`, as well as its own -aware variant, `Zend\Validator\Translator\TranslatorAwareInterface`. This was done to reduce dependencies for the component, and follows the principal of Separated Interfaces.

The upshot is that if you are migrating from a pre-2.2 version, and receiving errors indicating that the translator provided does not implement `Zend\Validator\Translator\TranslatorInterface`, you will need to make a change to your code.

An implementation of `Zend\Validator\Translator\TranslatorInterface` is provided in `Zend\Mvc\I18n\Translator`, which also extends `Zend\I18n\Translator\Translator`. This version can be instantiated and used just as the original `Zend\I18n` version.

A new service has also been registered with the MVC, `MvcTranslator`, which will return this specialized, bridge instance.

Most users should see no issues, as `Zend\Validator\ValidatorPluginManager` has been modified to use the `MvcTranslator` service internally, which is how most developers were getting the translator instance into validators in the first place. You will only need to change code if you were manually injecting the instance previously.

Validator classes provide a `setTranslator()` method with which you can specify an instance of `Zend\Validator\Translator\TranslatorInterface` which will translate the messages in case of a validation failure. The `getTranslator()` method returns the translator instance. `Zend\Mvc\I18n\Translator` provides an implementation compatible with the validator component.

```

1 $validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
2 $translate = new Zend\Mvc\Translator\Translator();
3 // configure the translator...
4
5 $validator->setTranslator($translate);

```

With the static `setDefaultTranslator()` method you can set an instance of `Zend\Validator\Translator\TranslatorInterface` 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.

```
1 $translate = new Zend\Mvc\I18n\Translator();
2 // configure the translator...
3
4 Zend\Validator\AbstractValidator::setDefaultTranslator($translate);
```

Sometimes it is necessary to disable the translator within a validator. To achieve this you can use the `setDisableTranslator()` method, which accepts a boolean parameter, and `isTranslatorDisabled()` to get the set value.

```
1 $validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
2 if (!$validator->isTranslatorDisabled()) {
3     $validator->setDisableTranslator();
4 }
```

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.

Standard Validation Classes

Zend Framework comes with a standard set of validation classes, which are ready for you to use.

Included Validators

- *Alnum*
- *Alpha*
- *Barcode*
- *Between*
- *Callback*
- *CreditCard*
- *Date*
- *Db\RecordExists and Db\NoRecordExists*
- *Digits*
- *EmailAddress*
- *File Validation Classes*
- *GreaterThan*
- *Hex*
- *Hostname*
- *Iban*
- *Identical*
- *InArray*
- *Ip*

- *Isbn*
- *IsFloat*
- *IsInt*
- LessThan
- NotEmpty
- PostCode
- *Regex*
- *Sitemap*
- *Step*
- StringLength
- *Timezone*
- *Uri*

Deprecated Validators

Ccnum

The `Ccnum` validator has been deprecated in favor of the `CreditCard` validator. For security reasons you should use `CreditCard` instead of `Ccnum`.

`Zend\I18n\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.

Supported Options

The following options are supported for `Zend\I18n\Validator\Alnum`:

- **`allowWhiteSpace`**: If whitespace characters are allowed. This option defaults to `FALSE`

Basic usage

A basic example is the following one:

```
1 $validator = new Zend\I18n\Validator\Alnum();
2 if ($validator->isValid('Abcd12')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

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()`.

```
1 $validator = new Zend\I18n\Validator\Alnum(array('allowWhiteSpace' => true));
2 if ($validator->isValid('Abcd and 12')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

Using different languages

There are actually 3 languages which are not accepted in their own script. These languages are **korean**, **japanese** and **chinese** because these languages are using an alphabet where a single character is built by using multiple characters.

In the case you are using these languages, the input will only be validated by using the english alphabet.

`Zend\I18n\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\I18n\Validator\Alnum` validator with the exception that it does not accept digits.

Supported options for `Zend\I18n\Validator\Alpha`

The following options are supported for `Zend\I18n\Validator\Alpha`:

- **`allowWhiteSpace`**: If whitespace characters are allowed. This option defaults to `FALSE`

Basic usage

A basic example is the following one:

```
1 $validator = new Zend\I18n\Validator\Alpha();
2 if ($validator->isValid('Abcd')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

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()`.

```
1 $validator = new Zend\I18n\Validator\Alpha(array('allowWhiteSpace' => true));
2 if ($validator->isValid('Abcd and efg')) {
3     // value contains only allowed chars
4 } else {
5     // false
6 }
```

Using different languages

When using `Zend\I18n\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 281

Barcode Validator

`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.

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.

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:

```
1 $valid = new Zend\Validator\Barcode('EAN13');
2 if ($valid->isValid($input)) {
3     // input appears to be valid
4 } else {
5     // input is invalid
6 }
```

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.

```
1 $valid = new Zend\Validator\Barcode(array(
2     'adapter' => 'EAN13',
3     'checksum' => false,
4 ));
5 if ($valid->isValid($input)) {
6     // input appears to be valid
7 } else {
8     // input is invalid
9 }
```

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.

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.

```
1 class My\Barcode\MyBar extends Zend\Validator\Barcode\AbstractAdapter
2 {
3     protected $length      = 'even';
4     protected $characters  = '0123456789ABCDE';
5     protected $checksum    = 'mod66';
6
7     protected function mod66($barcode)
8     {
9         // do some validations and return a boolean
10    }
11 }
12
13 $valid = new Zend\Validator\Barcode('My\Barcode\MyBar');
14 if ($valid->isValid($input)) {
15     // input appears to be valid
16 } else {
17     // input is invalid
18 }
```

Between Validator

`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.

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.

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.

```
1 $valid = new Zend\Validator\Between(array('min' => 0, 'max' => 10));
2 $value = 10;
3 $result = $valid->isValid($value);
4 // 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`.

Validation exclusive the border values

Sometimes it is useful to validate a value by excluding the border values. See the following example:

```
1 $valid = new Zend\Validator\Between(  
2     array(  
3         'min' => 0,  
4         'max' => 10,  
5         'inclusive' => false  
6     )  
7 );  
8 $value = 10;  
9 $result = $valid->isValid($value);  
10 // 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.

Zend\Validator\Callback allows you to provide a callback with which to validate a given value.

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.

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.

```
1 function myMethod($value)
2 {
3     // some validation
4     return true;
5 }
```

To use it within Zend\Validator\Callback you just have to call it this way:

```
1 $valid = new Zend\Validator\Callback('myMethod');
2 if ($valid->isValid($input)) {
3     // input appears to be valid
4 } else {
5     // input is invalid
6 }
```

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:

```
1 $valid = new Zend\Validator\Callback(function($value) {
2     // some validation
3     return true;
4 });
5
6 if ($valid->isValid($input)) {
7     // input appears to be valid
8 } else {
9     // input is invalid
10 }
```

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:

```
1 class MyClass
2 {
3     public function myMethod($value)
4     {
5         // some validation
6         return true;
7     }
8 }
```

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:

```
1 $object = new MyClass;
2 $valid = new Zend\Validator\Callback(array($object, 'myMethod'));
3 if ($valid->isValid($input)) {
4     // input appears to be valid
5 } else {
6     // input is invalid
7 }
```

You may also define a static method as a callback. Consider the following class definition and validator usage:

```
1 class MyClass
2 {
3     public static function test($value)
4     {
5         // some validation
6         return true;
7     }
8 }
9
10 $valid = new Zend\Validator\Callback(array('MyClass', 'test'));
11 if ($valid->isValid($input)) {
12     // input appears to be valid
13 } else {
```

```

14     // input is invalid
15 }

```

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:

```

1 class MyClass
2 {
3     public function __invoke($value)
4     {
5         // some validation
6         return true;
7     }
8 }
9
10 $object = new MyClass();
11 $valid = new Zend\Validator\Callback($object);
12 if ($valid->isValid($input)) {
13     // input appears to be valid
14 } else {
15     // input is invalid
16 }

```

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:

```

1 class MyClass
2 {
3     function myMethod($value, $option)
4     {
5         // some validation
6         return true;
7     }
8
9     //if a context is present
10    function myMethod($value, $context, $option)
11    {
12        // some validation
13        return true;
14    }
15
16 }

```

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 “callbackOptions”:

```

1 $valid = new Zend\Validator\Callback(array(
2     'callback' => array('MyClass', 'myMethod'),
3     'callbackOptions' => $options,

```

```
4    ));
5
6    if ($valid->isValid($input)) {
7        // input appears to be valid
8    } else {
9        // input is invalid
10   }
```

Otherwise, you may pass them to the validator after instantiation:

```
1  $valid = new Zend\Validator\Callback(array('MyClass', 'myMethod'));
2  $valid->setOptions($options);
3
4  if ($valid->isValid($input)) {
5      // input appears to be valid
6  } else {
7      // input is invalid
8  }
```

When there are additional values given to `isValid()` then these values will be added immediately after `$value`.

```
1  $valid = new Zend\Validator\Callback(array('MyClass', 'myMethod'));
2  $valid->setOptions($options);
3
4  if ($valid->isValid($input, $additional)) {
5      // input appears to be valid
6  } else {
7      // input is invalid
8  }
```

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.

`Zend\Validator\Date` allows you to validate if a given value contains a date.

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.

Default date validation

The easiest way to validate a date is by using the default date format. It is used when no format has been given.

```
1 $validator = new Zend\Validator\Date();  
2  
3 $validator->isValid('2000-10-10'); // returns true  
4 $validator->isValid('10.10.2000'); // returns false
```

The default date format for `Zend\Validator\Date` is 'yyyy-MM-dd'.

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()`.

```
1 $validator = new Zend\Validator\Date(array('format' => 'Y'));  
2
```

```
3 $validator->isValid('2010'); // returns true
4 $validator->isValid('May');  // returns false
```

Db\RecordExists and Db\NoRecordExists Validators

`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.

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.

Basic usage

An example of basic usage of the validators:

```
1 //Check that the email address exists in the database
2 $validator = new Zend\Validator\Db\RecordExists(
3     array(
4         'table'    => 'users',
5         'field'    => 'emailaddress',
```

```

6         'adapter' => $dbAdapter
7     )
8 );
9
10 if ($validator->isValid($emailaddress)) {
11     // email address appears to be valid
12 } else {
13     // email address is invalid; print the reasons
14     foreach ($validator->getMessages() as $message) {
15         echo "$message\n";
16     }
17 }

```

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.

```

1 //Check that the username is not present in the database
2 $validator = new Zend\Validator\Db\NoRecordExists(
3     array(
4         'table'    => 'users',
5         'field'    => 'username',
6         'adapter' => $dbAdapter
7     )
8 );
9 if ($validator->isValid($username)) {
10     // username appears to be valid
11 } else {
12     // username is invalid; print the reason
13     $messages = $validator->getMessages();
14     foreach ($messages as $message) {
15         echo "$message\n";
16     }
17 }

```

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.

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)

```

1 //Check no other users have the username
2 $user_id = $user->getId();
3 $validator = new Zend\Validator\Db\NoRecordExists(
4     array(
5         'table' => 'users',
6         'field' => 'username',
7         'exclude' => array(
8             'field' => 'id',
9             'value' => $user_id
10         )
11     )
12 );

```

```

11     )
12 );
13
14 if ($validator->isValid($username)) {
15     // username appears to be valid
16 } else {
17     // username is invalid; print the reason
18     $messages = $validator->getMessages();
19     foreach ($messages as $message) {
20         echo "$message\n";
21     }
22 }

```

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.

```

1 $email      = 'user@example.com';
2 $clause     = $dbAdapter->quoteIdentifier('email') . ' = ' . $dbAdapter->quoteValue(
3     ↪$email);
4 $validator = new Zend\Validator\Db\RecordExists(
5     array(
6         'table'    => 'users',
7         'field'    => 'username',
8         'adapter'  => $dbAdapter,
9         'exclude'  => $clause
10    )
11 );
12
13 if ($validator->isValid($username)) {
14     // username appears to be valid
15 } else {
16     // username is invalid; print the reason
17     $messages = $validator->getMessages();
18     foreach ($messages as $message) {
19         echo "$message\n";
20     }
21 }

```

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.

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:

```

1 $validator = new Zend\Validator\Db\RecordExists(
2     array(
3         'table'    => 'users',
4         'schema'   => 'my',
5         'field'    => 'id'
6     )
7 );

```

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.

```
1 $select = new Zend\Db\Sql\Select();
2 $select->from('users')
3     ->where->equalTo('id', $user_id)
4     ->where->equalTo('email', $email);
5
6 $validator = new Zend\Validator\Db\RecordExists($select);
7
8 // We still need to set our database adapter
9 $validator->setAdapter($dbAdapter);
10
11 // Validation is then performed as usual
12 if ($validator->isValid($username)) {
13     // username appears to be valid
14 } else {
15     // username is invalid; print the reason
16     $messages = $validator->getMessages();
17     foreach ($messages as $message) {
18         echo "$message\n";
19     }
20 }
```

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.

`Zend\Validator\Digits` validates if a given value contains only digits.

Supported options for `Zend\Validator\Digits`

There are no additional options for `Zend\Validator\Digits`:

Validating digits

To validate if a given value contains only digits and no other characters, simply call the validator like shown in this example:

```
1 $validator = new Zend\Validator\Digits();
2
3 $validator->isValid("1234567890"); // returns true
4 $validator->isValid(1234);          // returns true
5 $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`.

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.

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

```
1 // Does file have the given hash?
2 $validator = new \Zend\Validator\File\Crc32('3b3652f');
3
4 // Or, check file against multiple hashes
5 $validator = new \Zend\Validator\File\Crc32(array('3b3652f', 'e612b69'));
```

```
6 // Perform validation with file path
7
8 if ($validator->isValid('./myfile.txt')) {
9     // file is valid
10 }
```

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.

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.

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.

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

```

1 // Only allow files that exist in ~both~ directories
2 $validator = new \Zend\Validator\File\Exists('/tmp,/var/tmp');
3
4 // ...or with array notation
5 $validator = new \Zend\Validator\File\Exists(array('/tmp', '/var/tmp'));
6
7 // Perform validation
8 if ($validator->isValid('/tmp/myfile.txt')) {
9     // file is valid
10 }

```

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.

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

```

1 // Allow files with 'php' or 'exe' extensions
2 $validator = new \Zend\Validator\File\Extension('php,exe');
3
4 // ...or with array notation
5 $validator = new \Zend\Validator\File\Extension(array('php', 'exe'));
6
7 // Test with case-sensitivity on
8 $validator = new \Zend\Validator\File\Extension(array('php', 'exe'), true);
9
10 // Perform validation
11 if ($validator->isValid('./myfile.php')) {
12     // file is valid
13 }

```

Public Methods

addExtension (*string|array \$options*)

Adds extension(s) via a comma-delimited string or an array.

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

```
1 // Does file have the given hash?
2 $validator = new \Zend\Validator\File\Hash('3b3652f', 'crc32');
3
4 // Or, check file against multiple hashes
5 $validator = new \Zend\Validator\File\Hash(array('3b3652f', 'e612b69'), 'crc32');
6
7 // Perform validation with file path
8 if ($validator->isValid('./myfile.txt')) {
9     // file is valid
10 }
```

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.

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

```

1 // Is image size between 320x200 (min) and 640x480 (max)?
2 $validator = new \Zend\Validator\File\ImageSize(320, 200, 640, 480);
3
4 // ...or with array notation
5 $validator = new \Zend\Validator\File\ImageSize(array(
6     'minWidth' => 320, 'minHeight' => 200,
7     'maxWidth' => 640, 'maxHeight' => 480,
8 ));
9
10 // Is image size equal to or larger than 320x200?
11 $validator = new \Zend\Validator\File\ImageSize(array(
12     'minWidth' => 320, 'minHeight' => 200,
13 ));
14
15 // Is image size equal to or smaller than 640x480?
16 $validator = new \Zend\Validator\File\ImageSize(array(
17     'maxWidth' => 640, 'maxHeight' => 480,
18 ));
19
20 // Perform validation with file path
21 if ($validator->isValid('./myfile.jpg')) {
22     // file is valid
23 }
```

Public Methods

getImageMin ()

Returns the minimum dimensions (width and height)

Return type array

getImageMax ()

Returns the maximum dimensions (width and height)

Return type array

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

```
1 $validator = new \Zend\Validator\File\IsCompressed();
2 if ($validator->isValid('./myfile.zip')) {
3     // file is valid
4 }
```

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

```
1 $validator = new \Zend\Validator\File\IsImage();
2 if ($validator->isValid('./myfile.jpg')) {
3     // file is valid
4 }
```

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

```

1 // Does file have the given hash?
2 $validator = new \Zend\Validator\File\Md5('3b3652f336522365223');
3
4 // Or, check file against multiple hashes
5 $validator = new \Zend\Validator\File\Md5(array(
6     '3b3652f336522365223', 'eb3365f3365ddc65365'
7 ));
8
9 // Perform validation with file path
10 if ($validator->isValid('./myfile.txt')) {
11     // file is valid
12 }

```

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.

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:

- **contentType (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 mimeType extensions can not be found.

Usage Examples

```
1 // Only allow 'gif' or 'jpg' files
2 $validator = new \Zend\Validator\File\MimeType('image/gif,image/jpg');
3
4 // ...or with array notation
5 $validator = new \Zend\Validator\File\MimeType(array('image/gif', 'image/jpg'));
6
7 // ...or restrict an entire group of types
8 $validator = new \Zend\Validator\File\MimeType(array('image', 'audio'));
9
10 // Use a different magicFile
11 $validator = new \Zend\Validator\File\MimeType(array(
12     'image/gif', 'image/jpg',
13     'magicFile' => '/path/to/magicfile.mgx'
14 ));
15
16 // Use the HTTP information for the file type
17 $validator = new \Zend\Validator\File\MimeType(array(
18     'image/gif', 'image/jpg',
19     'enableHeaderCheck' => true
20 ));
21
22 // Perform validation
23 if ($validator->isValid('./myfile.jpg')) {
24     // file is valid
25 }
```

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.

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

```

1 // Only allow files that do not exist in ~either~ directories
2 $validator = new \Zend\Validator\File\NotExists('/tmp,/var/tmp');
3
4 // ...or with array notation
5 $validator = new \Zend\Validator\File\NotExists(array('/tmp', '/var/tmp'));
6
7 // Perform validation
8 if ($validator->isValid('/home/myfile.txt')) {
9     // file is valid
10 }

```

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.

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

```

1 // Does file have the given hash?
2 $validator = new \Zend\Validator\File\Sha1('3b3652f336522365223');
3
4 // Or, check file against multiple hashes
5 $validator = new \Zend\Validator\File\Sha1(array(
6     '3b3652f336522365223', 'eb3365f3365ddc65365'
7 ));
8
9 // Perform validation with file path
10 if ($validator->isValid('./myfile.txt')) {
11     // file is valid
12 }

```

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.

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

```
1 // Limit the file size to 40000 bytes
2 $validator = new \Zend\Validator\File\Size(40000);
3
4 // Limit the file size to between 10kB and 4MB
5 $validator = new \Zend\Validator\File\Size(array(
6     'min' => '10kB', 'max' => '4MB'
7 ));
8
9 // Perform validation with file path
10 if ($validator->isValid('./myfile.txt')) {
11     // file is valid
12 }
```


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

```

1 use Zend\Http\PhpEnvironment\Request;
2
3 $request = new Request();
4 $files   = $request->getFiles();
5 // i.e. $files['my-upload']['error'] == 0
6
7 $validator = new \Zend\Validator\File\UploadFile();
8 if ($validator->isValid($files['my-upload'])) {
9     // file is valid
10 }

```

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

```

1 // Limit the amount of words to a maximum of 2000
2 $validator = new \Zend\Validator\File\WordCount(2000);
3
4 // Limit the amount of words to between 100 and 5000
5 $validator = new \Zend\Validator\File\WordCount(100, 5000);
6
7 // ... or with array notation
8 $validator = new \Zend\Validator\File\WordCount(array(
9     'min' => 1000, 'max' => 5000
10 ));
11
12 // Perform validation with file path
13 if ($validator->isValid('./myfile.txt')) {
14     // file is valid
15 }

```

Hex Validator

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.

```
1 $validator = new Zend\Validator\Hex();
2 if ($validator->isValid('123ABC')) {
3     // value contains only hex chars
4 } else {
5     // false
6 }
```

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.

Supported options for Zend\Validator\Hex

There are no additional options for Zend\Validator\Hex:

Hostname Validator

`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.

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 *TLDs* are validated. This option defaults to `TRUE`.

Basic usage

A basic example of usage is below:

```
1 $validator = new Zend\Validator\Hostname();
2 if ($validator->isValid($hostname)) {
3     // hostname appears to be valid
4 } else {
5     // hostname is invalid; print the reasons
6     foreach ($validator->getMessages() as $message) {
7         echo "$message\n";
8     }
9 }
```

This will match the hostname `$hostname` and on failure populate `getMessages()` with useful error messages.

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:

```
1 $validator = new Zend\Validator\Hostname(Zend\Validator\Hostname::ALLOW_IP);
2 if ($validator->isValid($hostname)) {
3     // hostname appears to be valid
4 } else {
5     // hostname is invalid; print the reasons
6     foreach ($validator->getMessages() as $message) {
7         echo "$message\n";
8     }
9 }
```

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:

```
1 $validator = new Zend\Validator\Hostname(Zend\Validator\Hostname::ALLOW_DNS |
2                                           Zend\Validator\Hostname::ALLOW_IP);
```

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.

```
1 $validator =  
2     new Zend\Validator\Hostname(  
3         array(  
4             'allow' => Zend\Validator\Hostname::ALLOW_DNS,  
5             'useIdnCheck' => false  
6         )  
7     );
```

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 *IDNs* are only validated if you allow *DNS* hostnames to be validated.

Validating Top Level Domains

By default a hostname will be checked against a list of known *TLDs*. 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.

```
1 $validator =  
2     new Zend\Validator\Hostname(  
3         array(  
4             'allow' => Zend\Validator\Hostname::ALLOW_DNS,  
5             'useIdnCheck' => true,  
6             'useTldCheck' => false  
7         )  
8     );
```

Alternatively you can either pass TRUE or FALSE to `setValidateTld()` to enable or disable *TLD* validation.

Note: TLD validation

Please note *TLDs* are only validated if you allow *DNS* hostnames to be validated.

`Zend\Validator\Iban` validates if a given value could be a *IBAN* number. *IBAN* is the abbreviation for “International Bank Account Number”.

Supported options for `Zend\Validator\Iban`

The following options are supported for `Zend\Validator\Iban`:

- **country_code**: Sets the country code which is used to get the *IBAN* format for validation.

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 country code. By knowing this we already know how to use `Zend\Validator\Iban`.

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.

```
1 $validator = new Zend\Validator\Iban(array('country_code' => false));
2 // Note: you can also set a FALSE as single parameter
3
4 if ($validator->isValid('AT611904300234573201')) {
5     // IBAN appears to be valid
6 } else {
7     // IBAN is not valid
8 }
```

So **any** *IBAN* number will be valid. Note that this should not be done when you accept only accounts from a single country.

Region aware IBAN validation

To validate against a defined country, you just need to give the wished country code. You can do this by the option `country_code` and also afterwards by using `setCountryCode()`.

```
1 $validator = new Zend\Validator\Iban(array('country_code' => 'AT'));
2
3 if ($validator->isValid('AT611904300234573201')) {
4     // IBAN appears to be valid
5 } else {
6     // IBAN is not valid
7 }
```

`Zend\Validator\Identical` allows you to validate if a given value is identical with a set token.

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.
- **literal**: If set to `TRUE`, the validation will skip the lookup for elements in the form context, and validate the token just the way it was provided. The default value is `FALSE`.

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.

```
1 $valid = new Zend\Validator\Identical('origin');  
2 if ($valid->isValid($value)) {  
3     return true;  
4 }
```

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.

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.

```
1 $valid = new Zend\Validator\Identical(123);
2 if ($valid->isValid($input)) {
3     // input appears to be valid
4 } else {
5     // input is invalid
6 }
```

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`.

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:

```
1 $form->add(array(
2     'name' => 'elementOne',
3     'type' => 'Password',
4 ));
5 $form->add(array(
6     'name'      => 'elementTwo',
7     'type'      => 'Password',
8     'validators' => array(
9         array(
10            'name'      => 'Identical',
11            'options' => array(
12                'token' => 'elementOne',
13            ),
14        ),
15    ),
16 ));
```

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.

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.

```
1 use Zend\Form\Element;
2 use Zend\Form\Fieldset;
3 use Zend\Form\Form;
4 use Zend\InputFilter\Input;
```

```

5 use Zend\InputFilter\InputFilter;
6
7 $userFieldset = new Fieldset('user'); // (1)
8 $userFieldset->add(array(
9     'name' => 'email', // (2)
10    'type' => 'Email',
11));
12
13 // Let's add one fieldset inside the 'user' fieldset,
14 // so we can see how to manage the token in a different deepness
15 $deeperFieldset = new Fieldset('deeperFieldset'); // (3)
16 $deeperFieldset->add(array(
17     'name'      => 'deeperFieldsetInput', // (4)
18     'type'      => 'Text',
19     'options'   => array(
20         'label' => 'What validator are we testing?',
21     ),
22));
23 $userFieldset->add($deeperFieldset);
24
25 $signUpForm = new Form('signUp');
26 $signUpForm->add($userFieldset);
27 // Add an input that will validate the 'email' input from 'user' fieldset
28 $signUpForm->add(array(
29     'name' => 'confirmEmail', // (5)
30     'type' => 'Email',
31));
32 // Add an input that will validate the 'deeperFieldsetInput' from 'deeperFieldset'
33 // that lives inside the 'user' fieldset
34 $signUpForm->add(array(
35     'name' => 'confirmTestingValidator', // (6)
36     'type' => 'Text',
37));
38
39 $inputFilter = new InputFilter();
40 // This will ensure the user enter the same email in 'email' (2) and 'confirmEmail'
41 ↪ (5)
42 $inputFilter->add(array(
43     'name' => 'confirmEmail', // references (5)
44     'validators' => array(
45         array(
46             'name' => 'Identical',
47             'options' => array(
48                 // 'user' key references 'user' fieldset (1), and 'email' references
49                 ↪ 'email' element inside
50                 // 'user' fieldset (2)
51                 'token' => array('user' => 'email'),
52             ),
53         ),
54     ),
55));
56 // This will ensure the user enter the same string in 'deeperFieldsetInput' (4)
57 // and 'confirmTestingValidator' (6)
58 $inputFilter->add(array(
59     'name' => 'confirmTestingValidator', // references (6)
60     'validators' => array(
61         array(
62             'name' => 'Identical',

```

```

61         'options' => array(
62             'token' => array(
63                 'user' => array( // references 'user' fieldset (1)
64                     // 'deeperFieldset' key references 'deeperFieldset' fieldset_
65                     // (3)
66                     // 'deeperFieldsetInput' references 'deeperFieldsetInput' _
67                     // element (4)
68                     'deeperFieldset' => 'deeperFieldsetInput'
69                 )
70             ),
71         ),
72     );
73
74 $signupForm->setInputFilter($inputFilter);

```

Note: Always 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.

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`.

```

1 $valid = new Zend\Validator\Identical(array('token' => 123, 'strict' => FALSE));
2 $input = '123';
3 if ($valid->isValid($input)) {
4     // input appears to be valid
5 } else {
6     // input is invalid
7 }

```

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()`.

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 understand this situation.

```

1 // This will not validate array('token' => 123), it will actually validate the_
2 // integer 123
3 $valid = new Zend\Validator\Identical(array('token' => 123));
4 if ($valid->isValid($input)) {
5     // input appears to be valid
6 } else {
7 }

```

```
6     // input is invalid
7 }
```

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.

```
1 // Unlike the previous example, this will validate array('token' => 123)
2 $valid = new Zend\Validator\Identical(array('token' => array('token' => 123)));
3 if ($valid->isValid($input)) {
4     // input appears to be valid
5 } else {
6     // input is invalid
7 }
```

If the array you are willing to validate does not have a 'token' key, you do not need to wrap it.

`Zend\Validator\Ip` allows you to validate if a given value is an IP address. It supports the IPv4, IPv6 and IPvFuture definitions.

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`.
- **allowipfuture**: 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`.

Basic usage

A basic example of usage is below:

```
1 $validator = new Zend\Validator\Ip();
2 if ($validator->isValid($ip)) {
3     // ip appears to be valid
4 } else {
5     // ip is invalid; print the reasons
6 }
```

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 *URLs* 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.

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.

```
1 $validator = new Zend\Validator\Ip(array('allowipv6' => false));
2 if ($validator->isValid($ip)) {
3     // ip appears to be valid ipv4 address
4 } else {
5     // ip is no ipv4 address
6 }
```

Note: Default behaviour

The default behaviour which `Zend\Validator\Ip` follows is to allow both standards.

Zend\Validator\Isbn allows you to validate an *ISBN-10* or *ISBN-13* value.

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](#).

Basic usage

A basic example of usage is below:

```
1 $validator = new Zend\Validator\Isbn();
2 if ($validator->isValid($isbn)) {
3     // isbn is valid
4 } else {
5     // isbn is not valid
6 }
```

This will validate any *ISBN-10* and *ISBN-13* without separator.

Setting an explicit ISBN validation type

An example of an *ISBN* type restriction is below:

```
1 $validator = new Zend\Validator\Isbn();
2 $validator->setType(Zend\Validator\Isbn::ISBN13);
3 // OR
4 $validator = new Zend\Validator\Isbn(array(
5     'type' => Zend\Validator\Isbn::ISBN13,
6 ));
7
8 if ($validator->isValid($isbn)) {
9     // this is a valid ISBN-13 value
10 } else {
11     // this is an invalid ISBN-13 value
12 }
```

The above will validate only *ISBN-13* values.

Valid types include:

- `Zend\Validator\Isbn::AUTO` (default)
- `Zend\Validator\Isbn::ISBN10`
- `Zend\Validator\Isbn::ISBN13`

Specifying a separator restriction

An example of separator restriction is below:

```
1 $validator = new Zend\Validator\Isbn();
2 $validator->setSeparator('-');
3 // OR
4 $validator = new Zend\Validator\Isbn(array(
5     'separator' => '-',
6 ));
7
8 if ($validator->isValid($isbn)) {
9     // this is a valid ISBN with separator
10 } else {
11     // this is an invalid ISBN with separator
12 }
```

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)

IsInstanceOf Validator

`Zend\Validator\IsInstanceOf` allows you to validate whether a given object is an instance of a specific class or interface.

Supported options

The following options are supported for `Zend\Validator\IsInstanceOf`:

- **className**: Defines the fully-qualified class name which objects must be an instance of.

Basic usage

A basic example of usage is below:

```
1 $validator = new Zend\Validator\IsInstanceOf([
2     'className' => 'Zend\Validator\Digits'
3 ]);
4 $object = new Zend\Validator\Digits();
5
6 if ($validator->isValid($object)) {
7     // $object is an instance of Zend\Validator\Digits
8 } else {
9     // false. You can use $validator->getMessages() to retrieve error messages
10 }
```

If a string argument is passed to the constructor of `Zend\Validator\IsInstanceOf` then that value will be used as the class name:

```
1 $validator = new Zend\Validator\IsInstanceOf('Zend\Validator\Digits');
2 $object = new Zend\Validator\Digits();
3
4 if ($validator->isValid($object)) {
```

```
5      // $object is an instance of Zend\Validator\Digits
6  } else {
7      // false. You can use $validator->getMessages() to retrieve error messages
8  }
```

`Zend\I18n\Validator\IsFloat` allows you to validate if a given value contains a floating-point value. This validator validates also localized input.

Supported options for `Zend\I18n\Validator\IsFloat`

The following options are supported for `Zend\I18n\Validator\IsFloat`:

- **locale**: Sets the locale which will be used to validate localized float values.

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:

```
1 $validator = new Zend\I18n\Validator\IsFloat();
2
3 $validator->isValid(1234.5); // returns true
4 $validator->isValid('10a01'); // returns false
5 $validator->isValid('1,234.5'); // returns true
```

In the above example we expected that our environment is set to “en” as locale.

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\IsFloat is able to validate such notations. However, it is limited to the locale you set. See the following code:

```
1 $validator = new Zend\I18n\Validator\IsFloat(array('locale' => 'de'));
2
3 $validator->isValid(1234.5); // returns true
4 $validator->isValid("1 234,5"); // returns false
5 $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()`.

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `float` is a reserved keyword, which required renaming the `Float` validator. If you were using the `Float` validator directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `IsFloat` class instead.

Users pulling their `Float` validator instance from the validator plugin manager receive an `IsFloat` instance instead starting in 2.4.0.

`Zend\I18n\Validator\IsInt` validates if a given value is an integer. Also localized integer values are recognised and can be validated.

Supported Options

The following options are supported for `Zend\I18n\Validator\IsInt`:

- **locale**: Sets the locale which will be used to validate localized integers.

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:

```
1 $validator = new Zend\I18n\Validator\IsInt();
2
3 $validator->isValid(1234);    // returns true
4 $validator->isValid(1234.5); // returns false
5 $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.

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\IsInt 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:

```
1 $validator = new Zend\I18n\Validator\IsInt(array('locale' => 'de'));
2
3 $validator->isValid(1234); // returns true
4 $validator->isValid("1,234"); // returns false
5 $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()`.

Migration from 2.0-2.3 to 2.4+

Version 2.4 adds support for PHP 7. In PHP 7, `int` is a reserved keyword, which required renaming the `Int` validator. If you were using the `Int` validator directly previously, you will now receive an `E_USER_DEPRECATED` notice on instantiation. Please update your code to refer to the `IsInt` class instead.

Users pulling their `Int` validator instance from the validator plugin manager receive an `IsInt` instance instead starting in 2.4.0.

This validator allows you to validate if a given string conforms a defined regular expression.

Supported options for Zend\Validator\Regex

The following options are supported for Zend\Validator\Regex:

- **pattern**: Sets the regular expression pattern for this validator.

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:

```
1 $validator = new Zend\Validator\Regex(array('pattern' => '/^Test/'));
2
3 $validator->isValid("Test"); // returns true
4 $validator->isValid("Testing"); // returns true
5 $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](#).

Pattern handling

It is also possible to set a different pattern afterwards by using `setPattern()` and to get the actual set pattern with `getPattern()`.

```
1 $validator = new Zend\Validator\Regex(array('pattern' => '/^Test/'));
2 $validator->setPattern('ing$/');
3
4 $validator->isValid("Test"); // returns false
5 $validator->isValid("Testing"); // returns true
6 $validator->isValid("Pest"); // returns false
```

Sitemap Validators

The following validators conform to the [Sitemap XML](#) protocol.

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.

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

```
1 $validator = new Zend\Validator\Sitemap\Lastmod();
2
3 $validator->isValid('1999-11-11T22:23:52-02:00'); // true
4 $validator->isValid('2008-05-12T00:42:52+02:00'); // true
5 $validator->isValid('1999-11-11'); // true
6 $validator->isValid('2008-05-12'); // true
7
8 $validator->isValid('1999-11-11t22:23:52-02:00'); // false
9 $validator->isValid('2008-05-12T00:42:60+02:00'); // false
10 $validator->isValid('1999-13-11'); // false
```

```
11 $validator->isValid('2008-05-32'); // false
12 $validator->isValid('yesterday'); // false
```

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](#).

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

```
1 $validator = new Zend\Validator\Sitemap\Priority();
2
3 $validator->isValid('0.1'); // true
4 $validator->isValid('0.789'); // true
5 $validator->isValid(0.8); // true
6 $validator->isValid(1.0); // true
7
8 $validator->isValid('1.1'); // false
9 $validator->isValid('-0.4'); // false
10 $validator->isValid(1.00001); // false
11 $validator->isValid(0xFF); // false
12 $validator->isValid('foo'); // false
```

Supported options for `Zend\Validator\Sitemap_*`

There are no supported options for any of the Sitemap validators.

`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).

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

Basic usage

A basic example is the following one:

```
1 $validator = new Zend\Validator\Step();
2
3 if ($validator->isValid(1)) {
4     // value is a valid step value
5 } else {
6     // false
7 }
```

Using floating-point values

This validator also supports floating-point base value and step value. Here is a basic example of this feature:

```
1 $validator = new Zend\Validator\Step(  
2     array(  
3         'baseValue' => 1.1,  
4         'step'      => 2.2,  
5     )  
6 );  
7  
8 echo $validator->isValid(1.1); // prints true  
9 echo $validator->isValid(3.3); // prints true  
10 echo $validator->isValid(3.35); // prints false  
11 echo $validator->isValid(2.2); // prints false
```

Timezone Validator

Zend\Validator\Timezone allows validating if an input string represents a timezone.

Supported validation types

The Zend\Validator\Timezone validator is capable of validating the abbreviation (e.g. “ewt”) as well as the location string (e.g. “America/Los_Angeles”). These options are stored in the validator as `LOCATION`, `ABBREVIATION`, and `ALL` class constants.

Basic Usage

The default validation type will check again abbreviations as well as the location string.

```
1 $validator = new Zend\Validator\Timezone();
2
3 $validator->isValid('America/Los_Angeles'); // returns true
4 $validator->isValid('ewt'); // returns true
5 $validator->isValid('Foobar'); // returns false
```

To validate against only the location string you can set the type:

```
1 $validator = new Zend\Validator\Timezone();
2 $validator->setType(Zend\Validator\Timezone::LOCATION);
3
4 $validator->isValid('America/Los_Angeles'); // returns true
5 $validator->isValid('ewt'); // returns false
6 $validator->isValid('Foobar'); // returns false
```


`Zend\Validator\Uri` allows you to validate a uri using the `Zend\Uri\Uri` handler to parse to uri. The validator allows for both validation of absolute and/or relative uris. There is the possibility to exchange the handler for another one in case the parsing of the uri should be done differently.

Supported options

The following options are supported for `Zend\Validator\Uri`:

- **uriHandler**: Defines the handler to be used to parse the uri. This options defaults to a new instance of `Zend\Uri\Uri`.
- **allowRelative**: Defines if relative paths are allowed. This option defaults to `TRUE`.
- **allowAbsolute**: Defines if absolute paths are allowed. This option defaults to `TRUE`.

Basic usage

A basic example of usage is below:

```
1 $validator = new Zend\Validator\Uri();
2 $uri = 'http://framework.zend.com/manual';
3
4 if ($validator->isValid($uri)) {
5     // $uri was valid
6 } else {
7     // false. You can use $validator->getMessages() to retrieve error messages
8 }
```


Overview

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:

```
1 // Create a validator chain and add validators to it
2 $validatorChain = new Zend\Validator\ValidatorChain();
3 $validatorChain->attach(
4     new Zend\Validator\StringLength(array('min' => 6,
5                                           'max' => 12))
6     ->attach(new Zend\I18n\Validator\Alnum());
7
8 // Validate the username
9 if ($validatorChain->isValid($username)) {
10     // username passed validation
11 } else {
12     // username failed validation; print reasons
13     foreach ($validatorChain->getMessages() as $message) {
14         echo "$message\n";
15     }
16 }
```

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:

```
1 $validatorChain->attach(  
2     new Zend\Validator\StringLength(array('min' => 6,  
3                                         'max' => 12)),  
4     true)  
5     ->attach(new Zend\Validator\Alnum());
```

Any object that implements `Zend\Validator\ValidatorInterface` may be used in a validator chain.

Setting Validator Chain Order

For each validator added to the `ValidatorChain`, you can set a priority to define the chain order. The default value is 1. The more the priority is high, the earlier it is checked.

In the following example, the username is first checked to ensure that its length is between 7 and 9 characters, and then it is checked to ensure that its length is between 3 and 5 characters.

```
1 $username = 'ABCDFE';  
2  
3 // Create a validator chain and add validators to it  
4 $validatorChain = new Zend\Validator\ValidatorChain();  
5 $validatorChain->attach(  
6     new Zend\Validator\StringLength(array('min' => 3, 'max' => 5)),  
7     true,  
8     1  
9 );  
10 $validatorChain->attach(  
11     new Zend\Validator\StringLength(array('min' => 7, 'max' => 9)),  
12     true, // Break on failure  
13     2  
14 );  
15  
16 // Validate the username  
17 if ($validatorChain->isValid($username)) {  
18     // username passed validation  
19     echo "Success";  
20 } else {  
21     // username failed validation; print reasons  
22     foreach ($validatorChain->getMessages() as $message) {  
23         echo "$message\n";  
24     }  
25 }  
26  
27 // This first example will display: The input is less than 7 characters long
```

Overview

`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 *ZendInputFilter*.

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.

```

1 class MyValid\Float extends Zend\Validator\AbstractValidator
2 {
3     const FLOAT = 'float';
4
5     protected $messageTemplates = array(
6         self::FLOAT => "'%value%' is not a floating point value"
7     );
8
9     public function isValid($value)
10    {
11        $this->setValue($value);
12
13        if (!is_float($value)) {
14            $this->error(self::FLOAT);
15            return false;
16        }
17
18        return true;
19    }
20 }

```

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:

```

1 class MyValid\NumericBetween extends Zend\Validator\AbstractValidator
2 {
3     const MSG_NUMERIC = 'msgNumeric';
4     const MSG_MINIMUM = 'msgMinimum';
5     const MSG_MAXIMUM = 'msgMaximum';
6
7     public $minimum = 0;
8     public $maximum = 100;
9
10    protected $messageVariables = array(
11        'min' => 'minimum',
12        'max' => 'maximum'
13    );
14
15    protected $messageTemplates = array(
16        self::MSG_NUMERIC => "'%value%' is not numeric",
17        self::MSG_MINIMUM => "'%value%' must be at least '%min%'",

```



```

18         self::MSG_MAXIMUM => "'%value%' must be no more than '%max%'"
19     );
20
21     public function isValid($value)
22     {
23         $this->setValue($value);
24
25         if (!is_numeric($value)) {
26             $this->error(self::MSG_NUMERIC);
27             return false;
28         }
29
30         if ($value < $this->minimum) {
31             $this->error(self::MSG_MINIMUM);
32             return false;
33         }
34
35         if ($value > $this->maximum) {
36             $this->error(self::MSG_MAXIMUM);
37             return false;
38         }
39
40         return true;
41     }
42 }

```

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:

```

1 class MyValid\PasswordStrength extends Zend\Validator\AbstractValidator
2 {
3     const LENGTH = 'length';
4     const UPPER  = 'upper';

```

```
5     const LOWER = 'lower';
6     const DIGIT = 'digit';
7
8     protected $messageTemplates = array(
9         self::LENGTH => "'%value%' must be at least 8 characters in length",
10        self::UPPER => "'%value%' must contain at least one uppercase letter",
11        self::LOWER => "'%value%' must contain at least one lowercase letter",
12        self::DIGIT => "'%value%' must contain at least one digit character"
13    );
14
15    public function isValid($value)
16    {
17        $this->setValue($value);
18
19        $isValid = true;
20
21        if (strlen($value) < 8) {
22            $this->error(self::LENGTH);
23            $isValid = false;
24        }
25
26        if (!preg_match('/[A-Z]/', $value)) {
27            $this->error(self::UPPER);
28            $isValid = false;
29        }
30
31        if (!preg_match('/[a-z]/', $value)) {
32            $this->error(self::LOWER);
33            $isValid = false;
34        }
35
36        if (!preg_match('/\d/', $value)) {
37            $this->error(self::DIGIT);
38            $isValid = false;
39        }
40
41        return $isValid;
42    }
43 }
```

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()`.

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:

```
1 protected $messageTemplates = array(  
2     self::NOT_GREATER => "'%value%' is not greater than '%min%'",  
3 );
```

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.

```
1 $validator = new Zend\Validator\GreaterThan();  
2 $messages  = $validator->getMessageTemplates();
```

Using the `setMessage()` method you can set another message to be returned in case of the specified failure.

```
1 $validator = new Zend\Validator\GreaterThan();  
2 $validator->setMessage(  
3     'Please enter a lower value',  
4     Zend\Validator\GreaterThan::NOT_GREATER  
5 );
```

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.

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.

```
1 $translator = new Zend\Mvc\I18n\Translator();
2 $translator->addTranslationFile(
3     'phpArray',
4     'resources/languages/en/Zend_Validate.php', //or Zend_Captcha
5     'default',
6     'en_US'
7 );
8 Zend\Validator\AbstractValidator::setDefaultTranslator($translator);
```

Note: Supported languages

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.

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.

```
1 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`.

Getting the Zend Framework Version

Overview

`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

```
1 // returns -1, 0 or 1
2 $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

```
1 // returns 1.11.0 (or a later version)
2 echo Zend\Version\Version::getLatest();
```


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.

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.

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 autoloading configurations within the `config/autoload/` directory.

```

1  return array(
2      'view_manager' => array(
3          // The TemplateMapResolver allows you to directly map template names
4          // to specific templates. The following map would provide locations
5          // for a home page template ("application/index/index"), as well as for
6          // the layout ("layout/layout"), error pages ("error/index"), and
7          // 404 page ("error/404"), resolving them to view scripts.
8          'template_map' => array(
9              'application/index/index' => __DIR__ . '/../view/application/index/index.
10             ↪.phtml',
11              'site/layout'              => __DIR__ . '/../view/layout/layout.phtml',
12              'error/index'              => __DIR__ . '/../view/error/index.phtml',
13              'error/404'                => __DIR__ . '/../view/error/404.phtml',
14          ),
15          // The TemplatePathStack takes an array of directories. Directories
16          // are then searched in LIFO order (it's a stack) for the requested
17          // view script. This is a nice solution for rapid application
18          // development, but potentially introduces performance expense in
19          // production due to the number of static calls necessary.
20          //
21          // The following adds an entry pointing to the view directory
22          // of the current module. Make sure your keys differ between modules
23          // to ensure that they are not overwritten -- or simply omit the key!
24          'template_path_stack' => array(
25              'application' => __DIR__ . '/../view',
26          ),
27          // This will be used as the default suffix for template scripts resolving, it
28          ↪defaults to 'phtml'.
29          'default_template_suffix' => 'php',
30          // Set the template name for the site's layout.
31          //
32          // By default, the MVC's default Rendering Strategy uses the
33          // template name "layout/layout" for the site's layout.
34          // Here, we tell it to use the "site/layout" template,
35          // which we mapped via the TemplateMapResolver above.
36          'layout' => 'site/layout',
37          // By default, the MVC registers an "exception strategy", which is
38          // triggered when a requested action raises an exception; it creates

```



```

41     // a custom view model that wraps the exception, and selects a
42     // template. We'll set it to "error/index".
43     //
44     // Additionally, we'll tell it that we want to display an exception
45     // stack trace; you'll likely want to disable this by default.
46     'display_exceptions' => true,
47     'exception_template' => 'error/index',
48
49     // Another strategy the MVC registers by default is a "route not
50     // found" strategy. Basically, this gets triggered if (a) no route
51     // matches the current request, (b) the controller specified in the
52     // route match cannot be found in the service locator, (c) the controller
53     // specified in the route match does not implement the DispatchableInterface
54     // interface, or (d) if a response from a controller sets the
55     // response status to 404.
56     //
57     // The default template used in such situations is "error", just
58     // like the exception strategy. Here, we tell it to use the "error/404"
59     // template (which we mapped via the TemplateMapResolver, above).
60     //
61     // You can opt in to inject the reason for a 404 situation; see the
62     // various `Application\::ERROR_*` constants for a list of values.
63     // Additionally, a number of 404 situations derive from exceptions
64     // raised during routing or dispatching. You can opt-in to display
65     // these.
66     'display_not_found_reason' => true,
67     'not_found_template'       => 'error/404',
68 },
69 );

```

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.

```

1  namespace Foo\Controller;
2
3  use Zend\Mvc\Controller\AbstractActionController;
4  use Zend\View\Model\ViewModel;
5
6  class BazBatController extends AbstractActionController
7  {
8      public function doSomethingCrazyAction()
9      {
10         $view = new ViewModel(array(
11             'message' => 'Hello world',
12         ));
13         $view->setTemplate('foo/baz-bat/do-something-crazy');
14         return $view;
15     }
16 }

```

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:

```
1 namespace Foo\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4
5 class BazBatController extends AbstractActionController
6 {
7     public function doSomethingCrazyAction()
8     {
9         return array(
10             'message' => 'Hello world',
11         );
12     }
13 }
```

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.

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:

```
1 namespace Content\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\View\Model\ViewModel;
5
6 class ArticleController extends AbstractActionController
7 {
8     public function viewAction()
9     {
10         // get the article from the persistence layer, etc...
11
12         $view = new ViewModel();
13     }
14 }
```

```

14      // this is not needed since it matches "module/controller/action"
15      $view->setTemplate('content/article/view');
16
17      $articleView = new ViewModel(array('article' => $article));
18      $articleView->setTemplate('content/article');
19
20      $primarySidebarView = new ViewModel();
21      $primarySidebarView->setTemplate('content/main-sidebar');
22
23      $secondarySidebarView = new ViewModel();
24      $secondarySidebarView->setTemplate('content/secondary-sidebar');
25
26      $sidebarBlockView = new ViewModel();
27      $sidebarBlockView->setTemplate('content/block');
28
29      $secondarySidebarView->addChild($sidebarBlockView, 'block');
30
31      $view->addChild($articleView, 'article')
32          ->addChild($primarySidebarView, 'sidebar_primary')
33          ->addChild($secondarySidebarView, 'sidebar_secondary');
34
35      return $view;
36  }
37 }
    
```

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:

```

1  <?php // "content/article/view" template ?>
2  <!-- This is from the $view View Model, and the "content/article/view" template -->
3  <div class="row content">
4      <?php echo $this->article ?>
5
6      <?php echo $this->sidebar_primary ?>
7
8      <?php echo $this->sidebar_secondary ?>
9  </div>
    
```

```

1  <?php // "content/article" template ?>
2  <!-- This is from the $articleView View Model, and the "content/article"
3  template -->
4  <article class="span8">
5      <?php echo $this->escapeHtml('article') ?>
6  </article>
    
```

```

1  <?php // "content/main-sidebar" template ?>
2  <!-- This is from the $primarySidebarView View Model, and the
3  "content/main-sidebar" template -->
    
```

```

4     <div class="span2 sidebar">
5         sidebar content...
6     </div>

```

```

1 <?php // "content/secondary-sidebar template ?>
2 <!-- This is from the $secondarySidebarView View Model, and the
3      "content/secondary-sidebar" template -->
4 <div class="span2 sidebar pull-right">
5     <?php echo $this->block ?>
6 </div>

```

```

1 <?php // "content/block template ?>
2 <!-- This is from the $sidebarBlockView View Model, and the
3      "content/block" template -->
4 <div class="block">
5     block content...
6 </div>

```

And here is the aggregate, generated content:

```

1 <!-- This is from the $view View Model, and the "content/article/view" template -->
2 <div class="row content">
3     <!-- This is from the $articleView View Model, and the "content/article"
4          template -->
5     <article class="span8">
6         Lorem ipsum ....
7     </article>
8
9     <!-- This is from the $primarySidebarView View Model, and the
10          "content/main-sidebar" template -->
11     <div class="span2 sidebar">
12         sidebar content...
13     </div>
14
15     <!-- This is from the $secondarySidebarView View Model, and the
16          "content/secondary-sidebar" template -->
17     <div class="span2 sidebar pull-right">
18         <!-- This is from the $sidebarBlockView View Model, and the
19              "content/block" template -->
20         <div class="block">
21             block content...
22         </div>
23     </div>
24 </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.

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:

```

1 <html>
2   <head>
3     <title><?php echo $this->headTitle() ?></title>
4   </head>
5   <body>
6     <?php echo $this->content; ?>
7   </body>
8 </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:

```

1 namespace Foo\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\View\Model\ViewModel;
5
6 class BazBatController extends AbstractActionController
7 {
8     public function doSomethingCrazyAction()
9     {
10         $view = new ViewModel(array(
11             'message' => 'Hello world',
12         ));
13
14         // Capture to the layout view's "article" variable
15         $view->setCaptureTo('article');
16
17         return $view;
18     }
19 }

```

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.

```

1 namespace Foo\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\View\Model\ViewModel;
5
6 class BazBatController extends AbstractActionController
7 {
8     public function doSomethingCrazyAction()
9     {

```

```
10     $view = new ViewModel(array(
11         'message' => 'Hello world',
12     ));
13
14     // Disable layouts; `MvcEvent` will use this View Model instead
15     $view->setTerminal(true);
16
17     return $view;
18 }
19 }
```

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.

```
1 namespace Content\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4 use Zend\View\Model\ViewModel;
5
6 class ArticleController extends AbstractActionController
7 {
8     public function viewAction()
9     {
10         // get the article from the persistence layer, etc...
11
12         // Get the "layout" view model and inject a sidebar
13         $layout = $this->layout();
14         $sidebarView = new ViewModel();
15         $sidebarView->setTemplate('content/sidebar');
16         $layout->addChild($sidebarView, 'sidebar');
17
18         // Create and return a view model for the retrieved article
19         $view = new ViewModel(array('article' => $article));
20         $view->setTemplate('content/article');
21         return $view;
22     }
23 }
```

You could also use this technique to select a different layout, by simply calling the `setTemplate()` method of the layout View Model:

```
1 //In a controller
2 namespace Content\Controller;
3
4 use Zend\Mvc\Controller\AbstractActionController;
5 use Zend\View\Model\ViewModel;
6
7 class ArticleController extends AbstractActionController
8 {
9     public function viewAction()
10    {
11        // get the article from the persistence layer, etc...
12
13        // Get the "layout" view model and set an alternate template
14        $layout = $this->layout();
15        $layout->setTemplate('article/layout');
```

```

16
17     // Create and return a view model for the retrieved article
18     $view = new ViewModel(array('article' => $article));
19     $view->setTemplate('content/article');
20     return $view;
21 }
22 }

```

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.

```

1 //In a view script
2
3 // Change the layout:
4 $this->layout('alternate/layout'); // OR
5 $this->layout()->setTemplate('alternate/layout');
6
7 // Access a layout variable.
8 // Since access to the base view model is relatively easy, it becomes a
9 // reasonable place to store things such as API keys, which other view scripts
10 // may need.
11 $layout = $this->layout();
12 $disqusApiKey = false;
13 if (isset($layout->disqusApiKey)) {
14     $disqusApiKey = $layout->disqusApiKey;
15 }
16
17 // Set a layout variable
18 $this->layout()->footer = $this->render('article/footer');

```

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.

```

1 namespace Content;
2
3 class Module
4 {
5     /**
6      * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
7      * @return void
8      */
9     public function onBootstrap($e)
10     {
11         // Register a dispatch event
12         $app = $e->getParam('application');
13         $app->getEventManager()->attach('dispatch', array($this, 'setLayout'));
14     }
15
16     /**
17      * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
18      * @return void
19      */
20     public function setLayout($e)

```

```

21 {
22     $matches = $e->getRouteMatch();
23     $controller = $matches->getParam('controller');
24     if (false === strpos($controller, '__NAMESPACE__')) {
25         // not a controller from this module
26         return;
27     }
28
29     // Set the layout template
30     $viewModel = $e->getViewModel();
31     $viewModel->setTemplate('content/layout');
32 }
33 }

```

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, “render” 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:

```

1 namespace Application;
2
3 class Module
4 {
5     /**
6      * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
7      * @return void
8      */
9     public function onBootstrap($e)
10     {
11         // Register a "render" event, at high priority (so it executes prior
12         // to the view attempting to render)
13         $app = $e->getApplication();
14         $app->getEventManager()->attach('render', array($this, 'registerJsonStrategy
    →'), 100);

```



```

15     }
16
17     /**
18      * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
19      * @return void
20      */
21     public function registerJsonStrategy($e)
22     {
23         $app          = $e->getTarget();
24         $locator       = $app->getServiceManager();
25         $view          = $locator->get('Zend\View\View');
26         $jsonStrategy = $locator->get('ViewJsonStrategy');
27
28         // Attach strategy, which is a listener aggregate, at high priority
29         $view->getEventManager()->attach($jsonStrategy, 100);
30     }
31 }

```

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:

```

1 namespace Content;
2
3 class Module
4 {
5     /**
6      * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
7      * @return void
8      */
9     public function onBootstrap($e)
10     {
11         // Register a render event
12         $app = $e->getParam('application');
13         $app->getEventManager()->attach('render', array($this, 'registerJsonStrategy
14 ↪'), 100);
15     }
16
17     /**
18      * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
19      * @return void
20      */
21     public function registerJsonStrategy($e)
22     {
23         $matches = $e->getRouteMatch();
24         $controller = $matches->getParam('controller');
25         if (false == strpos($controller, __NAMESPACE__)) {
26             // not a controller from this module
27             return;
28         }
29
30         // Potentially, you could be even more selective at this point, and test
31         // for specific controller classes, and even specific actions or request
32         // methods.
33
34         // Set the JSON strategy when controllers from this module are selected

```

```

34     $app          = $e->getTarget();
35     $locator      = $app->getServiceManager();
36     $view         = $locator->get('Zend\View\View');
37     $jsonStrategy = $locator->get('ViewJsonStrategy');
38
39     // Attach strategy, which is a listener aggregate, at high priority
40     $view->getEventManager()->attach($jsonStrategy, 100);
41 }
42 }

```

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.

```

1  namespace Content\View;
2
3  use Zend\EventManager\EventManagerInterface;
4  use Zend\EventManager\ListenerAggregateInterface;
5  use Zend\Feed\Writer\Feed;
6  use Zend\View\Renderer\FeedRenderer;
7  use Zend\View\Renderer\JsonRenderer;
8  use Zend\View\Renderer\PhpRenderer;
9
10 class AcceptStrategy implements ListenerAggregateInterface
11 {
12     protected $feedRenderer;
13     protected $jsonRenderer;
14     protected $listeners = array();
15     protected $phpRenderer;
16
17     public function __construct(
18         PhpRenderer $phpRenderer,
19         JsonRenderer $jsonRenderer,
20         FeedRenderer $feedRenderer
21     ) {
22         $this->phpRenderer = $phpRenderer;
23         $this->jsonRenderer = $jsonRenderer;
24         $this->feedRenderer = $feedRenderer;
25     }
26
27     public function attach(EventManagerInterface $events, $priority = null)
28     {
29         if (null === $priority) {
30             $this->listeners[] = $events->attach('renderer', array($this,
31 ↪ 'selectRenderer'));
32             $this->listeners[] = $events->attach('response', array($this,
33 ↪ 'injectResponse'));
34         } else {
35             $this->listeners[] = $events->attach('renderer', array($this,
36 ↪ 'selectRenderer'), $priority);
37             $this->listeners[] = $events->attach('response', array($this,
38 ↪ 'injectResponse'), $priority);
39         }
40     }
41
42     public function detach(EventManagerInterface $events)

```

```

39 {
40     foreach ($this->listeners as $index => $listener) {
41         if ($events->detach($listener)) {
42             unset($this->listeners[$index]);
43         }
44     }
45 }
46
47 /**
48  * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
49  * @return \Zend\View\Renderer\RendererInterface
50  */
51 public function selectRenderer($e)
52 {
53     $request = $e->getRequest();
54     $headers = $request->getHeaders();
55
56     // No Accept header? return PhpRenderer
57     if (!$headers->has('accept')) {
58         return $this->phpRenderer;
59     }
60
61     $accept = $headers->get('accept');
62     foreach ($accept->getPrioritized() as $mediaType) {
63         if (0 === strpos($mediaType, 'application/json')) {
64             return $this->jsonRenderer;
65         }
66         if (0 === strpos($mediaType, 'application/rss+xml')) {
67             $this->feedRenderer->setFeedType('rss');
68             return $this->feedRenderer;
69         }
70         if (0 === strpos($mediaType, 'application/atom+xml')) {
71             $this->feedRenderer->setFeedType('atom');
72             return $this->feedRenderer;
73         }
74     }
75
76     // Nothing matched; return PhpRenderer. Technically, we should probably
77     // return an HTTP 415 Unsupported response.
78     return $this->phpRenderer;
79 }
80
81 /**
82  * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
83  * @return void
84  */
85 public function injectResponse($e)
86 {
87     $renderer = $e->getRenderer();
88     $response = $e->getResponse();
89     $result    = $e->getResult();
90
91     if ($renderer === $this->jsonRenderer) {
92         // JSON Renderer; set content-type header
93         $headers = $response->getHeaders();
94         $headers->addHeaderLine('content-type', 'application/json');
95     } elseif ($renderer === $this->feedRenderer) {
96         // Feed Renderer; set content-type header, and export the feed if

```

```
97         // necessary
98         $feedType = $this->feedRenderer->getFeedType();
99         $headers = $response->getHeaders();
100         $mediatype = 'application/'
101             . (('rss' == $feedType) ? 'rss' : 'atom')
102             . '+xml';
103         $headers->addHeaderLine('content-type', $mediatype);
104
105         // If the $result is a feed, export it
106         if ($result instanceof Feed) {
107             $result = $result->export($feedType);
108         }
109     } elseif ($renderer !== $this->phpRenderer) {
110         // Not a renderer we support, therefore not our strategy. Return
111         return;
112     }
113
114     // Inject the content
115     $response->setContent($result);
116 }
117 }
```

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.

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.

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:

```
1 use Zend\View\Renderer\PhpRenderer;
2
3 $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\RelativeFallbackResolver`, which allows using short template name into partial rendering. It is used as wrapper for each of two aforesaid resolvers. For example, this allows usage of partial template paths such as `my/module/script/path/my-view/some/partial.phtml`, while rendering template `my/module/script/path/my-view` by short name `some/partial`.

- `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:

```
1 use Zend\View\Renderer\PhpRenderer;
2 use Zend\View\Resolver;
3
4 $renderer = new PhpRenderer();
5
6 $resolver = new Resolver\AggregateResolver();
7
8 $renderer->setResolver($resolver);
9
10 $map = new Resolver\TemplateMapResolver(array(
11     'layout' => __DIR__ . '/view/layout.phtml',
12     'index/index' => __DIR__ . '/view/index/index.phtml',
13 ));
14 $stack = new Resolver\TemplatePathStack(array(
15     'script_paths' => array(
16         __DIR__ . '/view',
17         $someOtherPath
18     )
19 ));
20
21 $resolver->attach($map)    // this will be consulted first
22     ->attach($stack)
23     ->attach(new Resolver\RelativeFallbackResolver($map) // this allows using_
    ↪ short template name
24     ->attach(new Resolver\RelativeFallbackResolver($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:

```
1 return array(
2     'di' => array(
3         'instance' => array(
4             'Zend\View\Resolver\AggregateResolver' => array(
5                 'injections' => array(
6                     'Zend\View\Resolver\TemplateMapResolver',
7                     'Zend\View\Resolver\TemplatePathStack',
8                 ),
9             ),
10         ),
11         'Zend\View\Resolver\TemplateMapResolver' => array(
12             'parameters' => array(
13                 'map' => array(
14                     'layout' => __DIR__ . '/view/layout.phtml',
15                     'index/index' => __DIR__ . '/view/index/index.phtml',
16                 ),
17             ),
18         ),
19         'Zend\View\Resolver\TemplatePathStack' => array(
20             'parameters' => array(
```

```

21         'paths' => array(
22             'application' => __DIR__ . '/view',
23             'elsewhere' => $someOtherPath,
24         ),
25     ),
26 ),
27 'Zend\View\Renderer\PhpRenderer' => array(
28     'parameters' => array(
29         'resolver' => 'Zend\View\Resolver\AggregateResolver',
30     ),
31 ),
32 ),
33 ),
34 );
    
```

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:

```

1  use Zend\View\Model\ViewModel;
2  use Zend\View\Renderer\PhpRenderer;
3
4  $renderer = new PhpRenderer();
5
6  $model     = new ViewModel();
7  $model->setVariable('foo', 'bar');
8  // or
9  $model = new ViewModel(array('foo' => 'bar'));
10
11 $model->setTemplate($templateName);
12 $renderer->render($model);
    
```

Now, let's render something. As a simple example, let us say you have a list of book data.

```

1  // use a model to get the data for book authors and titles.
2  $data = array(
3      array(
4          'author' => 'Hernando de Soto',
5          'title' => 'The Mystery of Capitalism'
6      ),
7      array(
8          'author' => 'Henry Hazlitt',
9          'title' => 'Economics in One Lesson'
10     ),
11     array(
12         'author' => 'Milton Friedman',
13         'title' => 'Free to Choose'
14     )
15 );
    
```

```
16 // now assign the book data to a renderer instance
17 $renderer->books = $data;
18
19
20 // and render the template "booklist"
21 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.

```
1 namespace Bookstore\Controller;
2
3 use Zend\Mvc\Controller\AbstractActionController;
4
5 class BookController extends AbstractActionController
6 {
7     public function listAction()
8     {
9         // do some work...
10
11         // Assume $data is the list of books from the previous example
12         $model = new ViewModel(array('books' => $data));
13
14         // Optionally specify a template; if we don't, by default it will be
15         // auto-determined based on the module name, controller name and this action.
16         // In this example, the template would resolve to "bookstore/book/list",
17         // and thus the file "bookstore/book/list.phtml"; the following overrides
18         // that to set the template to "booklist", and thus the file "booklist.phtml"
19         // (note the lack of directory preceding the filename).
20         $model->setTemplate('booklist');
21
22         return $model;
23     }
24 }
```

This will then be rendered as if the following were executed:

```
1 $renderer->render($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:

```
1 <?php if ($this->books): ?>
2
3     <!-- A table of some books. -->
4     <table>
5         <tr>
6             <th>Author</th>
7             <th>Title</th>
8         </tr>
9
10        <?php foreach ($this->books as $key => $val): ?>
11        <tr>
12            <td><?php echo $this->escapeHtml($val['author']) ?></td>
13            <td><?php echo $this->escapeHtml($val['title']) ?></td>
14        </tr>
```



```

15         <?php endforeach; ?>
16
17     </table>
18
19 <?php else: ?>
20
21     <p>There are no books to display.</p>
22
23 <?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.

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`

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* `$nameOrModel`, *array* `$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`

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.

```
1 <?php if ($this->books): ?>
2
3     <!-- A table of some books. -->
4     <table>
5         <tr>
6             <th>Author</th>
7             <th>Title</th>
8         </tr>
9
10        <?php foreach ($this->books as $key => $val): ?>
11            <tr>
12                <td><?php echo $this->escapeHtml($val['author']) ?></td>
13                <td><?php echo $this->escapeHtml($val['title']) ?></td>
14            </tr>
15        <?php endforeach; ?>
```

```
16     </table>
17
18
19 <?php else: ?>
20
21     <p>There are no books to display.</p>
22
23 <?php endif; ?>
```

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.

```
1 // bad view-script practice:
2 echo $this->variable;
3
4 // good view-script practice:
5 echo $this->escapeHtml($this->variable);
6
7 // and remember context is always relevant!
8 <script type="text/javascript">
9     var foo = "<?php echo $this->escapeJs($variable) ?>";
10 </script>
```

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()`

Order of events

The following events are triggered, in the following order:

Table 309.1: 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.

ViewEvent::EVENT_RENDERER

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 309.2: 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 309.3: 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 309.4: ViewEvent::EVENT_RENDERER Listeners for FeedStrategy

Class	Priority	Method Called	Description
Zend\View\Strategy\FeedStrategy	1	selectRenderer	Return a FeedRenderer

Triggerers

This event is triggered by the following classes:

Table 309.5: ViewEvent::EVENT_RENDERER Triggerers

Class	In Method	Description
Zend\View\ViewRenderer		It has a short circuit callback that stops propagation once one result return an instance of a Renderer.

ViewEvent::EVENT_RENDERER_POST

Listeners

There are currently no built-in listeners for this event.

Triggerers

This event is triggered by the following classes:

Table 309.6: ViewEvent::EVENT_RENDERER_POST Triggerers

Class	In Method	Description
Zend\View\ViewRenderer		This event is triggered after ViewEvent::EVENT_RENDERER and before ViewEvent::EVENT_RESPONSE.

ViewEvent::EVENT_RESPONSE

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 309.7: ViewEvent::EVENT_RESPONSE Listeners for PhpStrategy

Class	Prior-ity	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 309.8: `ViewEvent::EVENT_RESPONSE` Listeners for `JsonStrategy`

Class	Prior-ity	Method Called	Description
<code>Zend\View\Strategy\JsonStrategy</code>	by	<code>injectResponse</code>	Populate the Response object from the view.

For FeedStrategy

This listener is added when the strategy used for rendering is `FeedStrategy`:

Table 309.9: `ViewEvent::EVENT_RESPONSE` Listeners for `FeedStrategy`

Class	Prior-ity	Method Called	Description
<code>Zend\View\Strategy\FeedStrategy</code>	by	<code>injectResponse</code>	Populate the Response object from the view.

Triggerers

This event is triggered by the following classes:

Table 309.10: `ViewEvent::EVENT_RESPONSE` Triggerers

Class	In Method	Description
<code>Zend\View\View</code>	<code>render</code>	This event is triggered after <code>ViewEvent::EVENT_RENDERER</code> and <code>ViewEvent::EVENT_RENDERER_POST</code> .

Introduction

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:

```
1 // $view is a PhpRenderer instance
2
3 // Via the plugin manager:
4 $pluginManager = $view->getHelperPluginManager();
5 $helper        = $pluginManager->get('lowercase');
6
7 // Retrieve the helper instance, via the method "plugin",
8 // which proxies to the plugin manager:
9 $helper = $view->plugin('lowercase');
10
11 // If the helper does not define __invoke(), the following also retrieves it:
12 $helper = $view->lowercase();
13
14 // If the helper DOES define __invoke, you can call the helper
15 // as if it is a method:
16 $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*.

Included Helpers

Zend Framework comes with an initial set of helper classes. In particular, there are helpers for creating route-based *URLs* 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*
- *FlashMessenger*
- *Gravatar*
- *HeadLink*
- *HeadMeta*
- *HeadScript*
- *HeadStyle*
- *HeadTitle*
- *HtmlList*
- *HTML Object Plugins*
- *Identity*
- *InlineScript*
- *JSON*
- *Partial*
- *Placeholder*
- *Url*

Note: View helpers related to **Internationalization** are documented in the *118n View Helpers* chapter.

Note: View helpers related to **form** are documented in the *Form View Helpers* chapter.

Note: View helpers related to **navigation** are documented in the *Navigation View Helpers* chapter.

Note: View helpers related to **paginator** are documented in the *Paginator Usage* chapter.

Note: For documentation on writing **custom view helpers** see the *Advanced usage* chapter.

View Helper - BasePath

Introduction

While most *URLs* generated by the framework have the base *URL* prepended automatically, developers will need to prepend the base *URL* to their own *URLs* (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.

Basic Usage

Usage of the `basePath()` helper is straightforward:

```
1  /*
2   * The following assume that the base URL of the page/application is "/mypage".
3   */
4
5  /*
6   * Prints:
7   * <base href="/mypage/" />
8   */
9  <base href="<?php echo $this->basePath(); ?>" />
10
11 /*
12  * Prints:
13  * <link rel="stylesheet" type="text/css" href="/mypage/css/base.css" />
14  */
15 <link rel="stylesheet" type="text/css"
16     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.

Introduction

The `Cycle` helper is used to alternate a set of values.

Basic Usage

To add elements to cycle just specify them in constructor:

```
1 <table>
2   <?php foreach ($this->books as $book): ?>
3       <tr style="background-color: <?php echo $this->cycle(array('#F0F0F0', '#FFF'))
4           ->next() ?>">
5           <td><?php echo $this->escapeHtml($book['author']) ?></td>
6       </tr>
7   <?php endforeach ?>
8 </table>
```

The output:

```
1 <table>
2   <tr style="background-color: #F0F0F0">
3       <td>First</td>
4   </tr>
5   <tr style="background-color: #FFF">
6       <td>Second</td>
7   </tr>
8 </table>
```

Or use `assign(array $data)` method and moving in backwards order:

```
1 <?php $this->cycle()->assign(array('#F0F0F0', '#FFF')) ?>
2
3 <table>
4     <?php foreach ($this->books as $book): ?>
5         <tr style="background-color: <?php echo $this->cycle()->prev() ?>">
6             <td><?php echo $this->escapeHtml($book['author']) ?></td>
7         </tr>
8     <?php endforeach ?>
9 </table>
```

The output:

```
1 <table>
2     <tr style="background-color: #FFF">
3         <td>First</td>
4     </tr>
5     <tr style="background-color: #F0F0F0">
6         <td>Second</td>
7     </tr>
8 </table>
```

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', '#FFF'), 'cycle2')`

```
1 <table>
2     <?php foreach ($this->books as $book): ?>
3         <tr style="background-color: <?php echo $this->cycle(array('#F0F0F0', '#FFF'))
4             <?php echo $this->cycle(array(1, 2, 3), 'number')->next() ?>">
5             <td><?php echo $this->cycle(array(1, 2, 3), 'number')->next() ?></td>
6             <td><?php echo $this->escapeHtml($book['author']) ?></td>
7         </tr>
8     <?php endforeach ?>
9 </table>
```

You can also use `assign($data, $name)` and `setName($name)` methods:

```
1 <?php
2 $this->cycle()->assign(array('#F0F0F0', '#FFF'), 'colors');
3 $this->cycle()->assign(array(1, 2, 3), 'numbers');
4 ?>
5 <table>
6     <?php foreach ($this->books as $book): ?>
7         <tr style="background-color: <?php echo $this->cycle()->setName('colors')->
8             <?php echo $this->cycle()->setName('numbers')->next() ?>">
9             <td><?php echo $this->cycle()->setName('numbers')->next() ?></td>
10            <td><?php echo $this->escapeHtml($book['author']) ?></td>
11        </tr>
12    <?php endforeach ?>
</table>
```


Introduction

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_RDFS`
- `XHTML1_RDFS11`
- `XHTML_BASIC1`
- `XHTML5`
- `HTML4_STRICT`
- `HTML4_LOOSE`
- `HTML4_FRAMESET`
- `HTML5`
- `CUSTOM_XHTML`
- `CUSTOM`

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*.

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:

```
1 $doctypeHelper = new Zend\View\Helper\Doctype();
2 $doctypeHelper->doctype('XHTML1_STRICT');
```

And then print it out on top of your layout script:

```
1 <?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.

```
1 $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:

```
1 if ($view->doctype()->isXhtml()) {
2     // do something differently
3 }
```

You can also check if the doctype represents an *HTML5* document.

```
1 if ($view->doctype()->isHtml5()) {
2     // do something differently
3 }
```

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.

```
1 $doctypeHelper = new Zend\View\Helper\Doctype();
2 $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:

```
1 <?php echo $this->doctype('XHTML1_RDFa'); ?>
2 <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:og="http://opengraphprotocol.org/schema/">
4 <head>
5     <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:

```
1 <?php echo $this->doctype() ?>
2 <html xmlns="http://www.w3.org/1999/xhtml"
3     <?php if ($view->doctype()->isRdfa()): ?>
4     xmlns:og="http://opengraphprotocol.org/schema/"
5     xmlns:fb="http://www.facebook.com/2008/fbml"
6     <?php endif; ?>
7 >
```

Zend MVC View Manager

If you're running a ZendMvc application, you should specify doctype via the *ViewManager* service.

FlashMessenger Helper

Introduction

The `FlashMessenger` helper is used to render the messages of the *FlashMessenger MVC plugin*.

Basic Usage

When only using the default namespace for the `FlashMessenger` all you need to do is this:

```
1 // Usable in any of your .phtml files
2 echo $this->flashMessenger()->render();
```

The first argument of the `render()`-function is the namespace. If no namespace is defined, the default `Zend\Mvc\Controller\Plugin\FlashMessenger::NAMESPACE_DEFAULT` will be used, which translates to default.

```
1 // Usable in any of your .phtml files
2 echo $this->flashMessenger()->render('error');
3
4 // Alternatively use one of the pre-defined namespaces
5 // (aka: use Zend\Mvc\Controller\Plugin\FlashMessenger;)
6 echo $this->flashMessenger()->render(FlashMessenger::NAMESPACE_SUCCESS);
```

CSS Layout

The `FlashMessenger` default rendering adds a CSS class to the generated HTML, that matches the defined namespace that should be rendered. While it may work well for the default cases, every so often you may want to add specific CSS classes to the HTML output. This can be done while making use of the second parameter of the `render()` function.

```
1 // Usable in any of your .phtml files
2 echo $this->flashMessenger()->render('error', array('alert', 'alert-danger'));
```

The output of this example, using the default HTML rendering settings, would look like this:

```
1 <ul class="alert alert-danger">
2     <li>Some FlashMessenger Content</li>
3     <li>You, the developer, are AWESOME!</li>
4 </ul>
```

HTML Layout

Aside from modifying the rendered CSS classes of the FlashMessenger, you are furthermore able to modify the generated HTML as a whole to create even more distinct visuals for your FlashMessages. The default output format is defined within the source code of the FlashMessenger view helper itself.

```
1 // Zend/View/Helper/FlashMessenger.php#L41-L43
2 protected $messageCloseString      = '</li></ul>';
3 protected $messageOpenFormat      = '<ul%s><li>';
4 protected $messageSeparatorString = '</li><li>';
```

These defaults exactly match what we're trying to do. The placeholder %s will be filled with the CSS classes output.

To change this, all we need to do is call the respective setter methods of these variables and give them new strings; for example:

```
1 // In any of your .phtml files:
2 echo $this->flashMessenger()
3     ->setMessageOpenFormat('<div%s><p>')
4     ->setMessageSeparatorString('</p><p>')
5     ->setMessageCloseString('</p></div>')
6     ->render('success');
```

The above code sample then would then generate the following output:

```
1 <div class="success">
2     <p>Some FlashMessenger Content</p>
3     <p>You, who's reading the docs, are AWESOME!</p>
4 </div>
```

Sample Modification for Twitter Bootstrap 3

Taking all the above knowledge into account, we can create a nice, highly usable and user-friendly rendering strategy using the Bootstrap front-end framework version 3 layouts:

```
1 // In any of your .phtml files:
2 $flash = $this->flashMessenger();
3 $flash->setMessageOpenFormat('<div%s>
4     <button type="button" class="close" data-dismiss="alert" aria-hidden="true">
5         &times;
6     </button>
7     <ul><li>'
8     ->setMessageSeparatorString('</li><li>')
```

```

9         ->setMessageCloseString('</li></ul></div>');
10
11     echo $flash->render('error',    array('alert', 'alert-dismissible', 'alert-danger'));
12     echo $flash->render('info',     array('alert', 'alert-dismissible', 'alert-info'));
13     echo $flash->render('default',  array('alert', 'alert-dismissible', 'alert-warning'));
14     echo $flash->render('success',  array('alert', 'alert-dismissible', 'alert-success'));

```

The output of the above example would create dismissable FlashMessages with the following HTML markup. The example only covers one type of FlashMessenger output; if you would have several FlashMessages available in each of the rendered namespaces, then you would receive the same output multiple times only having different CSS classes applied.

```

1 <div class="alert alert-dismissable alert-success">
2     <button type="button" class="close" data-dismiss="alert" aria-hidden="true"><times></button>
3     <ul>
4         <li>Some FlashMessenger Content</li>
5         <li>You, who's reading the docs, are AWESOME!</li>
6     </ul>
7 </div>

```

Alternative Configuration of the ViewHelper Layout

The FlashMessengerFactory checks the Configuration of the application. Therefore it is possible to set up the FlashMessenger strings through your *module.config.php*, too. The next example will set up the output to be identical with the above Twitter Bootstrap 3 Example

```

1 'view_helper_config' => array(
2     'flashmessenger' => array(
3         'message_open_format'      => '<div%s><button type="button" class="close"
4         data-dismiss="alert" aria-hidden="true">&times;</button><ul><li>',
5         'message_close_string'     => '</li></ul></div>',
6         'message_separator_string' => '</li><li>'
7     )
8 )

```


Introduction

The Gravatar helper is useful for rendering image HTML markup returned from the `gravatar.com` service.

Basic Usage

You can use the Gravatar helper wherever in view scripts per the following example:

```
1 //This could be inside any of your .phtml file
2 echo $this->gravatar('email@example.com')->getImgTag();
```

The first (and only, in this example) argument passed to the Gravatar helper is an e-mail for which you want grab an avatar from `gravatar.com`. For convenience, this e-mail will be automatically hashed via the md5 algorithm.

This will render the HTML below:

```

```

As you can see, the helper already provides URL defaults for you.

Custom Settings

You can customize the request for a `gravatar.com` image by using setter methods on the view helper:

```
1 $gravatar = $this->gravatar();
2
3 // Set the email instead of passing it via helper invocation
4 $gravatar->setEmail('email@example.com');
```

```

5
6 // Set the image size you want gravatar.com to return, in pixels
7 $gravatar->setImgSize(40);
8
9 // Set the default avatar image to use if gravatar.com does not find a match
10 $gravatar->setDefaultImg( \Zend\View\Helper\Gravatar::DEFAULT_MM );
11
12 // Set the avatar "rating" threshold (often used to omit NSFW avatars)
13 $gravatar->setRating( \Zend\View\Helper\Gravatar::RATING_G );
14
15 // Indicate that a secure URI should be used for the image source
16 $gravatar->setSecure(true);
17
18 // Render the <img> tag with the email you've set previously
19 echo $gravatar->getImgTag();

```

Alternately, you can pass an array as the second argument on invocation, with the following keys:

```

1 $settings = array(
2     'img_size' => 40,
3     'default_img' => \Zend\View\Helper\Gravatar::DEFAULT_MM,
4     'rating' => \Zend\View\Helper\Gravatar::RATING_G,
5     'secure' => null,
6 );
7 $email = 'email@example.com';
8 echo $this->gravatar($email, $settings);

```

Note: Passing null for the secure setting will cause the view helper to choose a schema that matches the current request to your application. This is the default behavior.

As you can see in the above examples, there are predefined settings for the default image and rating.

The Gravatar helper defines the following constants for ratings:

- RATING_G
- RATING_PG
- RATING_R
- RATING_X

The helper defines the following constants for the default image: * DEFAULT_404 * DEFAULT_MM * DEFAULT_IDENTICON * DEFAULT_MONSTERID * DEFAULT_WAVATAR

You may also provide custom attributes for the generated img tag. To do this, pass an attributes array to the `setAttributes()` method:

```

1 $gravatar = $this->gravatar('email@example.com');
2
3 // Suppose that I want to add the class attribute with a value of
4 // "gravatarccls" to the rendered <img> tag:
5 $attr = array(
6     'class' => 'gravatarccls',
7 );
8 echo $gravatar->setAttributes($attr)->getImgTag();

```

Alternately, you can pass this array as the third argument during helper invocation:

```
1 $email = 'email@example.com';
2 $settings = array(
3     'default_img' => \Zend\View\Helper\Gravatar::DEFAULT_MM,
4 );
5 $attr = array(
6     'class' => 'gravatar-image',
7     'id'    => 'gravatar',
8 );
9
10 echo $this->gravatar($email, $settings, $attr);
```


Introduction

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*.

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.

```
1 <?php
2 // setting links in a view script:
3 $this->headLink(array('rel' => 'icon', 'href' => '/img/favicon.ico'), 'PREPEND')
4     ->appendStylesheet('/styles/basic.css')
5     ->prependStylesheet(
6         '/styles/moz.css',
7         'screen',
8         true,
9         array('id' => 'my_stylesheet')
10    );
11
12 // rendering the links from the layout:
13 echo $this->headLink();
14 ?>
```

Output:

```
1 <link href="/styles/moz.css" media="screen" rel="stylesheet" type="text/css" id="my_
2 ↪stylesheet">
3 <link href="/img/favicon.ico" rel="icon">
4 <link href="/styles/basic.css" media="screen" rel="stylesheet" type="text/css">
```

Introduction

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*.

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:

```
1 // setting meta keywords
2 $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:

```
1 // disabling client-side cache
2 $this->headMeta()->appendHttpEquiv('expires',
3                                     'Wed, 26 Feb 1997 08:21:57 GMT')
4     ->appendHttpEquiv('pragma', 'no-cache')
5     ->appendHttpEquiv('Cache-Control', 'no-cache');
```

Another popular use for meta tags is setting the content type, character set, and language:

```
1 // setting content type and character set
2 $this->headMeta()->appendHttpEquiv('Content-Type',
3                                     'text/html; charset=UTF-8')
4     ->appendHttpEquiv('Content-Language', 'en-US');
```

If you are serving an *HTML5* document, you should provide the character set like this:

```
1 // setting character set in HTML5
2 $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":

```
1 // setting a meta refresh for 3 seconds to a new url:
2 $this->headMeta()->appendHttpEquiv('Refresh',
3                                     '3;URL=http://www.some.org/some.html');
```

When you're ready to place your meta tags in the layout, simply echo the helper:

```
1 <?php echo $this->headMeta() ?>
```


Usage with XHTML1_RDFS doctype

Enabling the RDFS 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:

```
1 $this->doctype(Zend\View\Helper\Doctype::XHTML1_RDFS);
2 $this->headMeta()->setProperty('og:title', 'my article title');
3 $this->headMeta()->setProperty('og:type', 'article');
4 echo $this->headMeta();
5
6 // output is:
7 // <meta property="og:title" content="my article title" />
8 // <meta property="og:type" content="article" />
```


Introduction

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

```
1 // adding scripts
2 $this->headScript()->appendFile(
3     '/js/prototype.js',
4     'text/javascript',
5     array('conditional' => 'lt IE 7')
6 );
```

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 than `'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 a jQuery template with the headScript

```
1 // jquery template
2 $template = '<div class="book">{{:title}}</div>';
3 $this->headScript()->appendScript(
4     $template,
5     'text/x-jquery-tmpl',
6     array('id' => 'tmpl-book', 'noescape' => true)
7 );
```

`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:

```
1 $this->headScript()->setAllowArbitraryAttributes(true);
```

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.

```
1 // adding scripts
2 $this->headScript()->appendFile('/js/prototype.js')
3     ->appendScript($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:

```
1 // Putting scripts in order
2
3 // place at a particular offset to ensure loaded last
4 $this->headScript()->offsetSetFile(100, '/js/myfuncs.js');
5
6 // use scriptaculous effects (append uses next index, 101)
7 $this->headScript()->appendFile('/js/scriptaculous.js');
8
9 // but always have base prototype script load first:
10 $this->headScript()->prependFile('/js/prototype.js');
```

When you're finally ready to output all scripts in your layout script, simply echo the helper:

```
1 <?php echo $this->headScript() ?>
```

Capturing Scripts

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:

```
1 <?php $this->headScript()->captureStart() ?>
2 var action = '<?php echo $this->baseUrl ?>';
3 $('foo_form').action = action;
4 <?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()`.

Introduction

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

```
1 // adding scripts
2 $this->headStyle()->appendStyle($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 to 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:

```
1 // module/MyModule/View/Renderer/MyRenderer.php
2 <?php
3
4 namespace MyModule\View\Renderer;
5
6 // Since we just want to implement the getEncoding() method, we can extend the Zend_
  ↳native renderer
7 use Zend\View\Renderer\PhpRenderer;
8
9 class MyRenderer extends PhpRenderer
10 {
11     /**
12      * @var string
13      */
14     protected $encoding;
15
16     /**
17      * Constructor
18      *
19      * @param string $encoding The encoding to be used
20      */
21     public function __construct($encoding)
22     {
23         parent::__construct();
```



```

24     $this->encoding = $encoding;
25 }
26
27 /**
28  * Sets the encoding
29  *
30  * @param string $encoding The encoding to be used
31  */
32 public function setEncoding($encoding)
33 {
34     $this->encoding = $encoding;
35 }
36
37 /**
38  * Gets the encoding
39  *
40  * @return string The encoding being used
41  */
42 public function getEncoding()
43 {
44     return $this->encoding;
45 }
46 }

```

Now we make some configuration in the module class:

```

1  // module/MyModule.php
2  <?php
3
4  namespace MyModule;
5
6  use MyModule\View\Renderer\MyRenderer;
7  use Zend\Mvc\MvcEvent;
8  use Zend\View\Strategy\PhpRendererStrategy;
9
10 class Module
11 {
12     public function getConfig() { /* ... */ }
13
14     public function getAutoloaderConfig() { /* ... */ }
15
16     public function getServiceConfig()
17     {
18         return array(
19             'factories' => array(
20                 // Register our custom renderer in the service manager
21                 'MyCustomRenderer' => function ($serviceManager) {
22                     $myRenderer = new MyRenderer('ISO-8859-1');
23                     return $myRenderer;
24                 },
25                 'MyCustomStrategy' => function ($serviceManager) {
26                     // As stated before, we just want to implement the getEncoding()  

↳method, so we can use
27                     // Zend\View\Strategy\PhpRendererStrategy and just provide our custom  

↳renderer to it.
28                     $myRenderer = $serviceManager->get('MyCustomRenderer');
29                     $strategy = new PhpRendererStrategy($myRenderer);
30                     return $strategy;

```

```
31         }
32     },
33 );
34 }
35
36 public function onBootstrap(MvcEvent $e)
37 {
38     // Register a render event
39     $app = $e->getParam('application');
40     $app->getEventManager()->attach('render', array($this, 'registerMyStrategy'),
↪100);
41 }
42
43 public function registerMyStrategy(MvcEvent $e)
44 {
45     $app      = $e->getTarget();
46     $locator  = $app->getServiceManager();
47     $view     = $locator->get('Zend\View\View');
48     $myStrategy = $locator->get('MyCustomStrategy');
49
50     // Attach strategy, which is a listener aggregate, at high priority
51     $view->getEventManager()->attach($myStrategy, 100);
52 }
53 }
```

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.

Basic Usage

You may specify a new style tag at any time:

```
1 // adding styles
2 $this->headStyle()->appendStyle($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:

```
1 // Putting styles in order
2
3 // place at a particular offset:
4 $this->headStyle()->offsetSetStyle(100, $customStyles);
5
6 // place at end:
7 $this->headStyle()->appendStyle($finalStyles);
8
9 // place at beginning
10 $this->headStyle()->prependStyle($firstStyles);
```

When you're finally ready to output all style declarations in your layout script, simply echo the helper:

```
1 <?php echo $this->headStyle() ?>
```

Capturing Style Declarations

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:

```
1 <?php $this->headStyle()->captureStart() ?>
2 body {
3     background-color: <?php echo $this->bgColor ?>;
4 }
5 <?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()`.

View Helper - HeadTitle

Introduction

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.

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.

```
1 // module/MyModule/Module.php
2 <?php
3
4 namespace MyModule;
5
6 class Module
7 {
8     /**
9      * @param  \Zend\Mvc\MvcEvent $e The MvcEvent instance
10     * @return void
```

```
11  */
12  public function onBootstrap($e)
13  {
14      // Register a render event
15      $app = $e->getParam('application');
16      $app->getEventManager()->attach('render', array($this, 'setLayoutTitle'));
17  }
18
19  /**
20   * @param \Zend\Mvc\MvcEvent $e The MvcEvent instance
21   * @return void
22   */
23  public function setLayoutTitle($e)
24  {
25      $matches = $e->getRouteMatch();
26      $action = $matches->getParam('action');
27      $controller = $matches->getParam('controller');
28      $module = __NAMESPACE__;
29      $siteName = 'Zend Framework';
30
31      // Getting the view helper manager from the application service manager
32      $viewHelperManager = $e->getApplication()->getServiceManager()->get(
33      ↪ 'viewHelperManager');
34
35      // Getting the headTitle helper from the view helper manager
36      $headTitleHelper = $viewHelperManager->get('headTitle');
37
38      // Setting a separator string for segments
39      $headTitleHelper->setSeparator(' - ');
40
41      // Setting the action, controller, module and site name as title segments
42      $headTitleHelper->append($action);
43      $headTitleHelper->append($controller);
44      $headTitleHelper->append($module);
45      $headTitleHelper->append($siteName);
46  }
```

When you're finally ready to render the title in your layout script, simply echo the helper:

```
1 <?php echo $this->headTitle() ?>
```

Output:

```
1 <title>action - controller - module - Zend Framework</title>
```

In case you want the title without the `<title>` and `</title>` tags you can use the `renderTitle()` method:

```
1 <?php echo $this->headTitle()->renderTitle() ?>
```

Output:

```
1 action - controller - module - Zend Framework
```

Introduction

`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.

Basic Usage

Unordered list

```
1 $items = array(  
2     'Level one, number one',  
3     array(  
4         'Level two, number one',  
5         'Level two, number two',  
6         array(  
7             'Level three, number one'  
8         ),  
9         'Level two, number three',  
10    ),  
11    'Level one, number two',  
12 );  
13  
14 echo $this->htmlList($items);
```

Output:

```
1 <ul>  
2     <li>Level one, number one
```

```
3         <ul>
4             <li>Level two, number one</li>
5             <li>Level two, number two
6                 <ul>
7                     <li>Level three, number one</li>
8                 </ul>
9             </li>
10            <li>Level two, number three</li>
11        </ul>
12    </li>
13    <li>Level one, number two</li>
14 </ul>
```

Ordered list

```
1 echo $this->htmlList($items, true);
```

Output:

```
1 <ol>
2     <li>Level one, number one
3         <ol>
4             <li>Level two, number one</li>
5             <li>Level two, number two
6                 <ol>
7                     <li>Level three, number one</li>
8                 </ol>
9             </li>
10            <li>Level two, number three</li>
11        </ol>
12    </li>
13    <li>Level one, number two</li>
14 </ol>
```

HTML attributes

```
1 $attribs = array(
2     'class' => 'foo',
3 );
4
5 echo $this->htmlList($items, false, $attribs);
```

Output:

```
1 <ul class="foo">
2     <li>Level one, number one
3         <ul class="foo">
4             <li>Level two, number one</li>
5             <li>Level two, number two
6                 <ul class="foo">
7                     <li>Level three, number one</li>
8                 </ul>
9             </li>
10            <li>Level two, number three</li>
```



```

11     </ul>
12     </li>
13     <li>Level one, number two</li>
14 </ul>

```

Escape Output

```

1 $items = array(
2     'Level one, number <strong>one</strong>',
3     'Level one, number <em>two</em>',
4 );
5
6 // Escape output (default)
7 echo $this->htmlList($items);
8
9 // Don't escape output
10 echo $this->htmlList($items, false, false, false);

```

Output:

```

1 <!-- Escape output (default) -->
2 <ul class="foo">
3     <li>Level one, number &lt;strong&gt;one&lt;/strong&gt;</li>
4     <li>Level one, number &lt;em&gt;two&lt;/em&gt;</li>
5 </ul>
6
7 <!-- Don't escape output -->
8 <ul class="foo">
9     <li>Level one, number <strong>one</strong></li>
10    <li>Level one, number <em>two</em></li>
11 </ul>

```

View Helper - HTML Object

Introduction

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*.

```
1 <?php echo $this->htmlFlash('/path/to/flash.swf'); ?>
```

This outputs the following *HTML*:

```
1 <object data="/path/to/flash.swf"
2     type="application/x-shockwave-flash"
3     classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
4     codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab
5     ">
6 </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.

```
1 echo $this->htmlObject(  
2     '/path/to/file.ext',  
3     'mime/type',  
4     array(  
5         'attr1' => 'aval1',  
6         'attr2' => 'aval2'  
7     ),  
8     array(  
9         'param1' => 'pval1',  
10        'param2' => 'pval2'  
11    ),  
12    'some content'  
13 );
```

This would output:

```
1 <object data="/path/to/file.ext" type="mime/type"  
2     attr1="aval1" attr2="aval2">  
3     <param name="param1" value="pval1" />  
4     <param name="param2" value="pval2" />  
5     some content  
6 </object>
```

Introduction

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.

Basic Usage

```
1 <?php
2     if ($user = $this->identity()) {
3         echo 'Logged in as ' . $this->escapeHtml($user->getUsername());
4     } else {
5         echo 'Not logged in';
6     }
7 ?>
```

Using with ServiceManager

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:

```
1 // In a configuration file...
2 return array(
3     'service_manager' => array(
4         'aliases' => array(
```

```
5         'Zend\Authentication\AuthenticationService' => 'my_auth_service',
6     ),
7     'invokables' => array(
8         'my_auth_service' => 'Zend\Authentication\AuthenticationService',
9     ),
10 ),
11 );
```

Introduction

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.

Basic Usage

Add to the layout script:

```
1 <body>
2   <!-- Content -->
3
4   <?php
5     echo $this->inlineScript()->prependFile($this->basePath('js/vendor/foundation.min.
   ↳js'))
6                                     ->prependFile($this->basePath('js/vendor/jquery.js'));
7   ?>
8 </body>
```

Output:

```
1 <body>
2     <!-- Content -->
3
4     <script type="text/javascript" src="/js/vendor/jquery.js"></script>
5     <script type="text/javascript" src="/js/vendor/foundation.min.js"></script>
6 </body>
```

Capturing Scripts

Add in your view scripts:

```
1 $this->inlineScript()->captureStart();
2 echo <<<JS
3     $('select').change(function(){
4         location.href = $(this).val();
5     });
6 JS;
7 $this->inlineScript()->captureEnd();
```

Output:

```
1 <body>
2     <!-- Content -->
3
4     <script type="text/javascript" src="/js/vendor/jquery.js"></script>
5     <script type="text/javascript" src="/js/vendor/foundation.min.js"></script>
6     <script type="text/javascript">
7         <!--
8         $('select').change(function(){
9             location.href = $(this).val();
10        });
11        -->
12    </script>
13 </body>
```


Introduction

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:

```
1 Content-Type: application/json
```

Most *AJAX* libraries look for this header when parsing responses to determine how to handle the content.

Basic Usage

Usage of the *JSON* helper is very straightforward:

```
1 <?php echo $this->json($this->data) ?>
```

Note: Enabling encoding using Zend\Json\Expr

The *JSON* helper accepts an array of options that will be passed to `Zend\Json\Json::encode()` and used internally to encode data.

`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:

```
1 <?php echo $this->json($this->data, array(  
2     'enableJsonExprFinder' => true,  
3 )) ?>
```


Introduction

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.

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 gives access to the current position of the array within the view script via `$this->partialLoop()->getPartialCounter()`. This provides an easy way to have alternating colors on table rows for example.

Basic Usage

Basic usage of partials is to render a template fragment in its own view scope. Consider the following partial script:

```
1 <?php // partial.phtml ?>
2 <ul>
3     <li>From: <?php echo $this->escapeHtml($this->from) ?></li>
4     <li>Subject: <?php echo $this->escapeHtml($this->subject) ?></li>
5 </ul>
```

You would then call it from your view script using the following:

```
1 <?php echo $this->partial('partial.phtml', array(
2     'from' => 'Team Framework',
3     'subject' => 'view partials')); ?>
```

Which would then render:

```
1 <ul>
2   <li>From: Team Framework</li>
3   <li>Subject: view partials</li>
4 </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 and 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:

```
1 // Tell partial to pass objects as 'model' variable
2 $view->partial()->setObjectKey('model');
3
4 // Tell partial to pass objects from partialLoop as 'model' variable
5 // in final partial view script:
6 $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:

```
1 <?php // partialLoop.phtml ?>
2 <dt><?php echo $this->key ?></dt>
3 <dd><?php echo $this->value ?></dd>
```

And the following “model”:

```
1 $model = array(
2     array('key' => 'Mammal', 'value' => 'Camel'),
3     array('key' => 'Bird', 'value' => 'Penguin'),
4     array('key' => 'Reptile', 'value' => 'Asp'),
```

```
5     array('key' => 'Fish', 'value' => 'Flounder'),  
6 );
```

In your view script, you could then invoke the `PartialLoop` helper:

```
1 <dl>  
2 <?php echo $this->partialLoop('partialLoop.phtml', $model) ?>  
3 </dl>
```

```
1 <dl>  
2     <dt>Mammal</dt>  
3     <dd>Camel</dd>  
4  
5     <dt>Bird</dt>  
6     <dd>Penguin</dd>  
7  
8     <dt>Reptile</dt>  
9     <dd>Asp</dd>  
10  
11    <dt>Fish</dt>  
12    <dd>Flounder</dd>  
13 </dl>
```


Introduction

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

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.

```
1 <?php $this->placeholder('foo')->set("Some text for later") ?>
2
3 <?php
4     echo $this->placeholder('foo');
5     // outputs "Some text for later"
6 ?>
```

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.

```
1 <!-- first view script -->
2 <?php $this->placeholder('foo')->exchangeArray($this->data) ?>
```

```
1 <!-- later view script -->
2 <?php
3 $this->placeholder('foo')->setPrefix("<ul>\n    <li>")
4                               ->setSeparator("</li><li>\n")
5                               ->setIndent(4)
6                               ->setPostfix("</li></ul>\n");
7
8
9 <?php
10     echo $this->placeholder('foo');
11     // outputs as unordered list with pretty indentation
12 ?>
```

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.

```
1 <?php $this->placeholder('foo')->bar = $this->data ?>
2 <?php echo $this->placeholder('foo')->bar ?>
3
4 <?php
5 $foo = $this->placeholder('foo');
6 echo $foo['bar'];
7 ?>
```

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.

```

1 <!-- Default capture: append -->
2 <?php $this->placeholder('foo')->captureStart();
3 foreach ($this->data as $datum): ?>
4 <div class="foo">
5     <h2><?php echo $datum->title ?></h2>
6     <p><?php echo $datum->content ?></p>
7 </div>
8 <?php endforeach; ?>
9 <?php $this->placeholder('foo')->captureEnd() ?>
10
11 <?php echo $this->placeholder('foo') ?>

```

```

1 <!-- Capture to key -->
2 <?php $this->placeholder('foo')->captureStart('SET', 'data');
3 foreach ($this->data as $datum): ?>
4 <div class="foo">
5     <h2><?php echo $datum->title ?></h2>
6     <p><?php echo $datum->content ?></p>
7 </div>
8 <?php endforeach; ?>
9 <?php $this->placeholder('foo')->captureEnd() ?>
10
11 <?php echo $this->placeholder('foo')->data ?>

```

Concrete 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](#)

View Helper - URL

The URL view helper is used to create a string representation of the routes that you define within your application. The syntax for the view helper is `$this->url($name, $params, $options, $reuseMatchedParameters)`, using the following definitions for the helper arguments:

- `$name`: The name of the route you want to output.
- `$params`: An array of parameters that is defined within the respective route configuration.
- `$options`: An array of options that will be used to create the URL.
- `$reuseMatchedParams`: A flag indicating if the currently matched route parameters should be used when generating the new URL.

Let's take a look at how this view helper is used in real-world applications.

Basic Usage

The following example shows a simple configuration for a news module. The route is called `news` and it has two **optional** parameters called `action` and `id`.

```
1 // In a configuration array (e.g. returned by some module's module.config.php)
2 'router' => array(
3     'routes' => array(
4         'news' => array(
5             'type' => 'segment',
6             'options' => array(
7                 'route' => '/news[:action][:id]',
8                 'constraints' => array(
9                     'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
10                ),
11                 'defaults' => array(
12                     'controller' => 'news',
13                     'action' => 'index',
14                ),
15            ),
16        ),
17    ),
18 );
```

```
15         )
16     )
17 )
18 ),
```

First, let's use the view helper to create the output for the URL `/news` without any of the optional parameters being used:

```
1 <a href="<?php echo $this->url('news'); ?>">News Index</a>
```

This will render the output:

```
1 <a href="/news">News Index</a>
```

Now let's assume we want to get a link to display the detail page of a single news entry. For this task, the optional parameters `action` and `id` need to have values assigned. This is how you do it:

```
1 <a href="<?php echo $this->url('news', array('action' => 'details', 'id' => 42)); ?>">
2     Details of News #42
3 </a>
```

This will render the output:

```
1 <a href="/news/details/42">News Index</a>
```

Query String Arguments

Most SEO experts agree that pagination parameters should not be part of the URL path; for example, the following URL would be considered a bad practice: `/news/archive/page/13`. Pagination is more correctly accomplished using a query string arguments, such as `/news/archive?page=13`. To achieve this, you'll need to make use of the `$options` argument from the view helper.

We will use the same route configuration as defined above:

```
1 // In a configuration array (e.g. returned by some module's module.config.php)
2 'router' => array(
3     'routes' => array(
4         'news' => array(
5             'type' => 'segment',
6             'options' => array(
7                 'route' => '/news[/:action][/id]',
8                 'constraints' => array(
9                     'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
10                ),
11                 'defaults' => array(
12                     'controller' => 'news',
13                     'action' => 'index',
14                ),
15            ),
16        ),
17    ),
18 ),
```

To generate query string arguments from the view helper, you need to assign them as the third argument using the query key like this:

```

1 <?php
2 $url = $this->url(
3     'news',
4     array('action' => 'archive'),
5     array(
6         'query' => array(
7             'page' => 13,
8         )
9     )
10 );
11 ?>
12 <a href="<?php echo $url; ?>">News Archive Page #13</a>

```

The above code sample would output:

```

1 <a href="/news/archive?page=13">News Archive Page #13</a>

```

Fragments

Another possible entry within the `$options` array is the assignment of URL fragments (typically used to link to in-page anchors), denoted with using the `fragment` key. Let's assume we want to enter a link for users to directly jump to the comment section of a details page:

```

1 <?php
2 $url = $this->url(
3     'news',
4     array('action' => 'details', 'id' => 42),
5     array(
6         'fragment' => 'comments',
7     )
8 );
9 ?>
10 <a href="<?php echo $url; ?>">Comment Section of News #42</a>

```

The above code sample would output:

```

1 <a href="/news/details/42#comments">Comment Section of News #42</a>

```

You can use `fragment` and `query` options at the same time!

```

1 <?php
2 $url = $this->url(
3     'news',
4     array('action' => 'details', 'id' => 42),
5     array(
6         'query' => array(
7             'commentPage' => 3,
8         ),
9         'fragment' => 'comments',
10    )
11 );
12 ?>
13 <a href="<?php echo $url; ?>">Comment Section of News #42</a>

```

The above code sample would output:

```
1 <a href="/news/details/42?commentPage=3#comments">Comment Section of News #42</a>
```

Fully Qualified Domain Name

Another possible entry within the `$options` array is to output a fully qualified domain name (absolute URL), denoted using the `force_canonical` key:

```
1 <?php
2 $url = $this->url(
3     'news',
4     array(),
5     array(
6         'force_canonical' => true,
7     )
8 );
9 ?>
10 <a href="<?php echo $url; ?>">News Index</a>
```

The above code sample would output:

```
1 <a href="http://www.example.com/news">News Index</a>
```

Reusing Matched Parameters

When you're on a route that has many parameters, often times it makes sense to reuse currently matched parameters instead of assigning them new explicitly. In this case, the argument `$reuseMatchedParams` will come in handy.

As an example, we will imagine being on a detail page for our “news” route. We want to display links to the edit and delete actions without having to assign the ID again. This is how you would do it:

```
1 // Currently url /news/details/777
2
3 <a href="<?php echo $this->url('news', array('action' => 'edit'), null, true); ?>">
4 ↪Edit Me</a>
5
6 <a href="<?php echo $this->url('news', array('action' => 'delete'), null, true); ?>">
7 ↪Delete Me</a>
```

Notice the `true` argument in the fourth position. This tells the view helper to use the matched `id` (777) when creating the new URL:

```
1 <a href="/news/edit/777">Edit Me</a>
2 <a href="/news/delete/777">Edit Me</a>
```

Shorthand

Due to the fact that reusing parameters is a use case that can happen when no route options are set, the third argument for the URL view helper will be checked against its type; when a boolean is passed, the helper uses it to set the value of the `$reuseMatchedParams` flag:

```
1 $this->url('news', array('action' => 'archive'), null, true);  
2 // is equal to  
3 $this->url('news', array('action' => 'archive'), true);
```

Advanced usage of helpers

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:

```
1 // $view is an instance of PhpRenderer
2 $pluginManager = $view->getHelperPluginManager();
3
4 // Register as an invokable class:
5 $pluginManager->setInvokableClass('lowercase', 'MyModule\View\Helper\LowerCase');
6
7 // Register as a factory:
8 $pluginManager->setFactory('lowercase', function ($pluginManager) {
9     $lowercaseHelper = new MyModule\View\Helper\LowerCase;
10
11     // ...do some configuration or dependency injection...
12
13     return $lowercaseHelper;
14 });
```

Within an MVC application, you will typically simply pass a map of plugins to the class via your configuration.

```
1 // From within a configuration file
2 return array(
3     'view_helpers' => array(
4         'invokables' => array(
5             'lowercase' => 'MyModule\View\Helper\LowerCase',
6             'uppercase' => 'MyModule\View\Helper\UpperCase',
```

```
7         ),
8     ),
9 );
```

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).

```
1 namespace MyModule;
2
3 class Module
4 {
5     public function getAutoloaderConfig(){ /*common code*/ }
6     public function getConfig(){ /*common code*/ }
7
8     public function getViewHelperConfig()
9     {
10         return array(
11             'invokables' => array(
12                 'lowercase' => 'MyModule\View\Helper\LowerCase',
13                 'uppercase' => 'MyModule\View\Helper\UpperCase',
14             ),
15         );
16     }
17 }
```

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!

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:

```
1 namespace Zend\View\Helper;
2
3 use Zend\View\Renderer\RendererInterface as Renderer;
4
5 interface HelperInterface
6 {
7     /**
8      * Set the View object
9      *
10     * @param Renderer $view
11     * @return HelperInterface
12     */
13     public function setView(Renderer $view);
14
15     /**
16     * Get the View object
17     *
18     * @return Renderer
19     */
20     public function getView();
21 }
```

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”

```
1 // /module/src/MyModule/View/Helper/SpecialPurpose.php
2 namespace MyModule\View\Helper;
3
4 use Zend\View\Helper\AbstractHelper;
5
6 class SpecialPurpose extends AbstractHelper
7 {
8     protected $count = 0;
9
10    public function __invoke()
11    {
12        $this->count++;
13        $output = sprintf("I have seen 'The Jerk' %d time(s).", $this->count);
14        return htmlspecialchars($output, ENT_QUOTES, 'UTF-8');
15    }
16 }
```

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.

```
1 // remember, in a view script, $this refers to the Zend\View\Renderer\PhpRenderer_
   ↳ instance.
2 echo $this->specialPurpose();
3 echo $this->specialPurpose();
4 echo $this->specialPurpose();
```

The output would look something like this:

```
1 I have seen 'The Jerk' 1 time(s) .
2 I have seen 'The Jerk' 2 time(s) .
3 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:

```
1 namespace MyModule\View\Helper;
2
3 use Zend\View\Helper\AbstractHelper;
4
5 class SpecialPurpose extends AbstractHelper
6 {
7     protected $count = 0;
8
9     public function __invoke()
10    {
```

```
11     $this->count++;
12     $output = sprintf("I have seen 'The Jerk' %d time(s).", $this->count);
13     $escaper = $this->getView()->plugin('escapehtml');
14     return $escaper($output);
15 }
16 }
```

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.

```
1 // $view is a PhpRenderer instance
2
3 $helper = new MyModule\View\Helper\LowerCase;
4 // ...do some configuration or dependency injection...
5
6 $view->getHelperPluginManager()->setService('lowercase', $helper);
```

The plugin manager will validate the helper/plugin, and if the validation passes, the helper/plugin will be registered.

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.

Quick Start

To show how easy is to create *XML-RPC* services with `Zend\XmlRpc\Server`, take a look at the following example:

```
1 class Greeter
2 {
3
4     /**
5      * Say hello to someone.
6      *
7      * @param string $name Who to greet
8      * @return string
9      */
10    public function sayHello($name='Stranger')
11    {
12        return sprintf("Hello %s!", $name);
13    }
14 }
15
16 $server = new Zend\XmlRpc\Server;
17 // Our Greeter class will be called
18 // greeter from the client
19 $server->setClass('Greeter', 'greeter');
20 $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:

```
1 $client = new Zend\XmlRpc\Client('http://example.com/xmlrpcserver.php');
2
3 echo $client->call('greeter.sayHello');
4 // will output "Hello Stranger!"
5
6 echo $client->call('greeter.sayHello', array('Dude'));
7 // will output "Hello Dude!"
```

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.

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

```
1 $client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');
2
3 echo $client->call('test.sayHello');
4
5 // 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

```
1 $client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');
2
3 $arg1 = 1.1;
4 $arg2 = 'foo';
5
6 $result = $client->call('test.sayHello', array($arg1, $arg2));
7
8 // $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.

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).

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 331.1: 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:

```
1 $client->setSkipSystemLookup(true);
2 $result = $client->call('foo.bar', array(array()));
```

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 exist 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 331.2: 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\Text
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
date-Time.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`.

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

```
1 $client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');
2
3 $service = $client->getProxy();           // Proxy the default namespace
4
5 $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

```
1 $client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');
2
3 $test = $client->getProxy('test');        // Proxy the "test" namespace
4
5 $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()`.

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.

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

```
1 $client = new Zend\XmlRpc\Client('http://foo/404');
2
3 try {
```

```

4      $client->call('bar', array($arg1, $arg2));
5
6  } catch (Zend\XmlRpc\Client\Exception\HttpException $e) {
7
8      // $e->getCode() returns 404
9      // $e->getMessage() returns "Not Found"
10
11  }
12

```

Regardless of how the *XML-RPC* client is used, the `Zend\XmlRpc\Client\Exception\HttpException` will be thrown whenever an *HTTP* error occurs.

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

```

1  $client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');
2
3  try {
4
5      $client->call('badMethod');
6
7  } catch (Zend\XmlRpc\Client\Exception\FaultException $e) {
8
9      // $e->getCode() returns 1
10     // $e->getMessage() returns "Unknown method"
11
12 }

```

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`.

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.

```
1 $client = new Zend\XmlRpc\Client('http://example.com/xmlrpcserver.php');
2 $introspector = $client->getIntrospector();
3 foreach ($introspector->listMethods() as $method) {
4     echo "Method: " . $method . "\n";
5 }
```

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

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

```
1 $client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');
2
3 $request = new Zend\XmlRpc\Request();
4 $request->setMethod('test.sayHello');
5 $request->setParams(array('foo', 'bar'));
6
7 $client->doRequest($request);
8
9 // $client->getLastRequest() returns instanceof Zend\XmlRpc\Request
10 // $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.

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.

Introduction

Zend\XmlRpc\Server is intended as a fully-featured *XML-RPC* server, following [the specifications outlined at www.xmlrpc.com](http://www.xmlrpc.com). Additionally, it implements the `system.multicall()` method, allowing boxcarring of requests.

Basic Usage

An example of the most basic use case:

```
1 $server = new Zend\XmlRpc\Server();  
2 $server->setClass('My\Service\Class');  
3 echo $server->handle();
```

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\Faultobject`. These objects both have `__toString()` methods that create valid *XML-RPC XML* responses, allowing them to be directly echoed.

Anatomy of a webservice

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\Mvc\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/`.

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.

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:

```
1  /**
2   * This is a sample function
3   *
4   * @param base64 $val1 Base64-encoded data
5   * @param dateTime.iso8601 $val2 An ISO date
6   * @param struct $val3 An associative array
7   * @return struct
8   */
9  function myFunc($val1, $val2, $val3)
```



```

10 {
11 }

```

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:

```

1  /**
2   * This is a sample function
3   *
4   * @param string|base64 $val1 String or base64-encoded data
5   * @param string|dateTime.iso8601 $val2 String or an ISO date
6   * @param array|struct $val3 Normal indexed array or an associative array
7   * @return boolean|struct
8   */
9  function myFunc($val1, $val2, $val3)
10 {
11 }

```

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.

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:

```

1  // All public methods in My_Service_Class will be accessible as
2  // myservice.METHODNAME
3  $server->setClass('My\Service\Class', 'myservice');
4
5  // Function 'somefunc' will be accessible as funcs.somefunc
6  $server->addFunction('somefunc', 'funcs');

```

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.

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()`.

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'.

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:

```

1  use Zend\XmlRpc\Server as XmlRpcServer;
2
3  // Register the "My\Services" namespace
4  $loader = new Zend\Loader\StandardAutoloader();
5  $loader->registerNamespace('My\Services', 'path to My/Services');
6  $loader->register();
7
8  $cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
9  $server = new XmlRpcServer();
10
11 if (!XmlRpcServer\Cache::get($cacheFile, $server)) {
12
13     $server->setClass('My\Services\Glue', 'glue'); // glue. namespace
14     $server->setClass('My\Services\Paste', 'paste'); // paste. namespace
15     $server->setClass('My\Services\Tape', 'tape'); // tape. namespace
16
17     XmlRpcServer\Cache::save($cacheFile, $server);
18 }
19
20 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.

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.

```

1  /**
2   * Return the MD5 sum of a value
3   *
4   * @param string $value Value to md5sum
5   * @return string MD5 sum of value
6   */
7  function md5Value($value)
8  {
9      return md5($value);
10 }
11
12 $server = new Zend\XmlRpc\Server();
13 $server->addFunction('md5Value');
14 echo $server->handle();
    
```

Attaching a class

The example below illustrates attaching a class' public methods as dispatchable *XML-RPC* methods.

```
1 require_once 'Services/Comb.php';
2
3 $server = new Zend\XmlRpc\Server();
4 $server->setClass('Services\Comb');
5 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.

```
1 namespace Services;
2
3 class PricingService
4 {
5     /**
6      * Calculate current price of product with $productId
7      *
8      * @param ProductRepository $productRepository
9      * @param PurchaseRepository $purchaseRepository
10     * @param integer $productId
11     */
12     public function calculate(ProductRepository $productRepository,
13                             PurchaseRepository $purchaseRepository,
14                             $productId)
15     {
16         ...
17     }
18 }
19
20 $server = new Zend\XmlRpc\Server();
21 $server->setClass('Services\PricingService',
22                 'pricing',
23                 new ProductRepository(),
24                 new PurchaseRepository());
```

The arguments passed at `setClass()` at server construction time are injected into the method call `pricing.calculate()` on remote invocation. 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`.

```
1 class Services\PricingService2
2 {
3     /**
4      * @param ProductRepository $productRepository
5      * @param PurchaseRepository $purchaseRepository
6      */
```

```

7     public function __construct(ProductRepository $productRepository,
8                               PurchaseRepository $purchaseRepository)
9     {
10         ...
11     }
12
13     /**
14      * Calculate current price of product with $productId
15      *
16      * @param integer $productId
17      * @return double
18      */
19     public function calculate($productId)
20     {
21         ...
22     }
23 }
24
25 $server = new Zend\XmlRpc\Server();
26 $server->sendArgumentsToAllMethods(false);
27 $server->setClass('Services\PricingService2',
28                 'pricing',
29                 new ProductRepository(),
30                 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.

```

1  require_once 'Services/Comb.php';
2  require_once 'Services/Brush.php';
3  require_once 'Services/Pick.php';
4
5  $server = new Zend\XmlRpc\Server();
6  $server->setClass('Services\Comb', 'comb'); // methods called as comb.*
7  $server->setClass('Services\Brush', 'brush'); // methods called as brush.*
8  $server->setClass('Services\Pick', 'pick'); // methods called as pick.*
9  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.

```

1  require_once 'Services/Exception.php';
2  require_once 'Services/Comb.php';
3  require_once 'Services/Brush.php';
4  require_once 'Services/Pick.php';

```

```
5
6 // Allow Services_Exceptions to report as fault responses
7 Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');
8
9 $server = new Zend\XmlRpc\Server();
10 $server->setClass('Services\Comb', 'comb'); // methods called as comb.*
11 $server->setClass('Services\Brush', 'brush'); // methods called as brush.*
12 $server->setClass('Services\Pick', 'pick'); // methods called as pick.*
13 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.

```
1 require_once 'Services/Request.php';
2 require_once 'Services/Exception.php';
3 require_once 'Services/Comb.php';
4 require_once 'Services/Brush.php';
5 require_once 'Services/Pick.php';
6
7 // Allow Services_Exceptions to report as fault responses
8 Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');
9
10 $server = new Zend\XmlRpc\Server();
11 $server->setClass('Services\Comb', 'comb'); // methods called as comb.*
12 $server->setClass('Services\Brush', 'brush'); // methods called as brush.*
13 $server->setClass('Services\Pick', 'pick'); // methods called as pick.*
14
15 // Create a request object
16 $request = new Services\Request();
17
18 echo $server->handle($request);
```

Specifying a custom response class

The example below illustrates specifying a custom response class for the returned response.

```
1 require_once 'Services/Request.php';
2 require_once 'Services/Response.php';
3 require_once 'Services/Exception.php';
4 require_once 'Services/Comb.php';
5 require_once 'Services/Brush.php';
6 require_once 'Services/Pick.php';
7
8 // Allow Services_Exceptions to report as fault responses
9 Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');
10
11 $server = new Zend\XmlRpc\Server();
12 $server->setClass('Services\Comb', 'comb'); // methods called as comb.*
```

```

13 $server->setClass('Services\Brush', 'brush'); // methods called as brush.*
14 $server->setClass('Services\Pick', 'pick'); // methods called as pick.*
15
16 // Create a request object
17 $request = new Services\Request();
18
19 // Utilize a custom response
20 $server->setResponseClass('Services\Response');
21
22 echo $server->handle($request);

```

Performance optimization

Cache server definitions between requests

The example below illustrates caching server definitions between requests.

```

1 use Zend\XmlRpc\Server as XmlRpcServer;
2
3 // Register the "Services" namespace
4 $loader = new Zend\Loader\StandardAutoloader();
5 $loader->registerNamespace('Services', 'path to Services');
6 $loader->register();
7
8 // Specify a cache file
9 $cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
10
11 // Allow Services\Exceptions to report as fault responses
12 XmlRpcServer\Fault::attachFaultException('Services\Exception');
13
14 $server = new XmlRpcServer();
15
16 // Attempt to retrieve server definition from cache
17 if (!XmlRpcServer\Cache::get($cacheFile, $server)) {
18     $server->setClass('Services\Comb', 'comb'); // methods called as comb.*
19     $server->setClass('Services\Brush', 'brush'); // methods called as brush.*
20     $server->setClass('Services\Pick', 'pick'); // methods called as pick.*
21
22     // Save cache
23     XmlRpcServer\Cache::save($cacheFile, $server);
24 }
25
26 // Create a request object
27 $request = new Services\Request();
28
29 // Utilize a custom response
30 $server->setResponseClass('Services\Response');
31
32 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 its *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.

```
1 use Zend\XmlRpc;
2
3 XmlRpc\AbstractValue::setGenerator(new XmlRpc\Generator\XmlWriter());
4
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.

Introduction

ZendService\Akismet\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.

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.

```
1 // Instantiate with the API key and a URL to the application or
2 // resource being used
3 $akismet = new ZendService\Akismet\Akismet($apiKey,
4                                             'http://framework.zend.com/wiki/');
5 if ($akismet->verifyKey($apiKey)) {
6     echo "Key is valid.\n";
7 } else {
8     echo "Key is not valid\n";
9 }
```

If called with no arguments, `verifyKey()` uses the *API* key provided to the constructor.

`verifyKey()` implements Akismet's *verify-key* REST method.

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

```
1 $data = array(  
2     'user_ip'           => '111.222.111.222',  
3     'user_agent'       => 'Mozilla/5.0 ' . '(Windows; U; Windows NT ' .  
4                          '5.2; en-GB; rv:1.8.1) Gecko/20061010 ' .  
5                          'Firefox/2.0',  
6     'comment_type'     => 'contact',  
7     'comment_author'   => 'John Doe',  
8     'comment_author_email' => 'nospam@myhaus.net',  
9     'comment_content'  => "I'm not a spammer, honest!"  
10 );  
11 if ($akismet->isSpam($data)) {  
12     echo "Sorry, but we think you're a spammer.";  
13 } else {  
14     echo "Welcome to our site!";  
15 }
```

`isSpam()` implements the *comment-check* Akismet *API* method.

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

```

1  $data = array(
2      'user_ip'           => '111.222.111.222',
3      'user_agent'       => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
4                          'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
5      'comment_type'     => 'contact',
6      'comment_author'   => 'John Doe',
7      'comment_author_email' => 'nospam@myhaus.net',
8      'comment_content'  => "I'm not a spammer, honest!"
9  );
10 $akismet->submitSpam($data);

```

`submitSpam()` implements the *submit-spam* Akismet *API* method.

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

```

1  $data = array(
2      'user_ip'           => '111.222.111.222',
3      'user_agent'       => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
4                          'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
5      'comment_type'     => 'contact',
6      'comment_author'   => 'John Doe',
7      'comment_author_email' => 'nospam@myhaus.net',
8      'comment_content'  => "I'm not a spammer, honest!"
9  );
10 $akismet->submitHam($data);

```

`submitHam()` implements the *submit-ham* Akismet *API* method.

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.

Introduction

`ZendService\Amazon\Amazon` is a simple *API* for using Amazon web services. `ZendService\Amazon\Amazon` has two *APIs*: a more traditional one that follows Amazon's own *API*, and a simpler “Query *API*” for constructing even complex search queries easily.

`ZendService\Amazon\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\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\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.

```
1 $amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
2 $results = $amazon->itemSearch(array('SearchIndex' => 'Books',
3                                     'Keywords' => 'php'));
4 foreach ($results as $result) {
5     echo $result->Title . '<br />';
6 }
```

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.

```
1 $query = new ZendService\Amazon\Query('AMAZON_API_KEY',
2                                     'US',
3                                     'AMAZON_SECRET_KEY');
4 $query->category('Books')->Keywords('PHP');
5 $results = $query->search();
6 foreach ($results as $result) {
7     echo $result->Title . '<br />';
8 }
```

Country Codes

By default, `ZendService\Amazon\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

```
1 // Connect to Amazon in Japan
2 $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*.

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

```
1 $amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
2 $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*.

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

```
1 $amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
2 $results = $amazon->itemSearch(array('SearchIndex' => 'Books',
3                                     'Keywords' => 'php'));
4 foreach ($results as $result) {
5     echo $result->Title . '<br />';
6 }
```

Using the ResponseGroup Option

The *ResponseGroup* option is used to control the specific information that will be returned in the response.

```
1 $amazon = new ZendService\Amazon\Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
2 $results = $amazon->itemSearch(array(
3     'SearchIndex'    => 'Books',
4     'Keywords'       => 'php',
5     'ResponseGroup'  => 'Small,ItemAttributes,Images,SalesRank,Reviews,' .
6                       'EditorialReview,Similarities,ListmaniaLists'
7 ));
8 foreach ($results as $result) {
9     echo $result->Title . '<br />';
10 }
```

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.

Using the Alternative Query API

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:

```
1 $query = new ZendService\Amazon\Query('MY_API_KEY', 'US', 'AMAZON_SECRET_KEY');
2 $query->Category('Books')->Keywords('PHP');
3 $results = $query->search();
4 foreach ($results as $result) {
5     echo $result->Title . '<br />';
6 }
```

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](#).

ZendService\Amazon Classes

The following classes are all returned by `ZendServiceAmazonAmazon::itemLookup()` and `ZendServiceAmazonAmazon::itemSearch()`:

- *ZendServiceAmazonItem*
- *ZendServiceAmazonImage*
- *ZendServiceAmazonResultSet*
- *ZendServiceAmazonOfferSet*
- *ZendServiceAmazonOffer*
- *ZendServiceAmazonSimilarProduct*
- *ZendServiceAmazonAccessories*
- *ZendServiceAmazonCustomerReview*
- *ZendServiceAmazonEditorialReview*
- *ZendServiceAmazonListMania*

ZendService\Amazon\Item

`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.

ZendService\Amazon\Item::asXML()

string:asXML()

Return the original *XML* for the item

Properties

`ZendService\Amazon\Item` has a number of properties directly related to their standard Amazon *API* counterparts.

Table 334.1: `ZendServiceAmazonItem` Properties

Name	Type	Description
ASIN	string	Amazon Item ID
Detail-PageURL	string	URL to the Items Details Page
SalesRank	int	Sales Rank for the Item
SmallImage	<code>ZendServiceAmazon-Image</code>	Small Image of the Item
MediumImage	<code>ZendServiceAmazon-Image</code>	Medium Image of the Item
LargeImage	<code>ZendServiceAmazon-Image</code>	Large Image of the Item
Subjects	array	Item Subjects
Offers	<code>ZendServiceAmazonOfferSet</code>	Offer Summary and Offers for the Item
CustomerReviews	array	Customer reviews represented as an array of <code>ZendServiceAmazonCustomerReview</code> objects
EditorialReviews	array	Editorial reviews represented as an array of <code>ZendServiceAmazonEditorialReview</code> objects
SimilarProducts	array	Similar Products represented as an array of <code>ZendServiceAmazonSimilarProduct</code> objects
Accessories	array	Accessories for the item represented as an array of <code>ZendServiceAmazonAccessories</code> objects
Tracks	array	An array of track numbers and names for Music CDs and DVDs
ListmaniaLists	array	Item related Listmania Lists as an array of <code>ZendServiceAmazonListmaniaList</code> objects
Promotional-Tag	string	Item Promotional Tag

[Back to Class List](#)

ZendService\Amazon\Image

`ZendService\Amazon\Image` represents a remote Image for a product.

Properties

Table 334.2: `ZendServiceAmazonImage` Properties

Name	Type	Description
Url	<code>ZendUriUri</code>	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](#)

ZendService\Amazon\ResultSet

`ZendService\Amazon\ResultSet` objects are returned by `ZendServiceAmazonAmazon::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()`.

ZendService\Amazon\ResultSet::totalResults()

`int:totalResults()` Returns the total number of results returned by the search

[Back to Class List](#)

ZendService\Amazon\OfferSet

Each result returned by `ZendServiceAmazonAmazon::itemSearch()` and `ZendServiceAmazonAmazon::itemLookup()` contains a `ZendService\Amazon\OfferSet` object through which pricing information for the item can be retrieved.

Properties

Table 334.3: ZendServiceAmazonOfferSet Properties

Name	Type	Description
LowestNewPrice	int	Lowest Price for the item in “New” condition
LowestNewPriceCurrency	string	The currency for the LowestNewPrice
LowestOldPrice	int	Lowest Price for the item in “Used” condition
LowestOldPriceCurrency	string	The currency for the LowestOldPrice
TotalNew	int	Total number of “new” condition available for the item
TotalUsed	int	Total number of “used” condition available for the item
TotalCollectible	int	Total number of “collectible” condition available for the item
TotalRefurbished	int	Total number of “refurbished” condition available for the item
Offers	array	An array of ZendServiceAmazonOffer objects.

[Back to Class List](#)

ZendService\Amazon\Offer

Each offer for an item is returned as an `ZendService\Amazon\Offer` object.

Properties

Table 334.4: 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](#)

ZendService\Amazon\SimilarProduct

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.

Properties

Table 334.5: ZendServiceAmazonSimilarProduct Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

[Back to Class List](#)

ZendService\Amazon\Accessories

Accessories for the returned item are represented as `ZendService\Amazon\Accessories` objects

Properties

Table 334.6: ZendServiceAmazonAccessories Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

[Back to Class List](#)

ZendService\Amazon\CustomerReview

Each Customer Review is returned as a `ZendService\Amazon\CustomerReview` object.

Properties

Table 334.7: 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](#)

ZendService\Amazon\EditorialReview

Each items Editorial Reviews are returned as a `ZendService\Amazon\EditorialReview` object

Properties

Table 334.8: ZendServiceAmazonEditorialReview Properties

Name	Type	Description
Source	string	Source of the Editorial Review
Content	string	Review Content

[Back to Class List](#)

ZendService\Amazon>Listmania

Each results List Mania List items are returned as `ZendService\Amazon>Listmania` objects.

Properties

Table 334.9:
ZendServiceAmazonListmania
Properties

Name	Type	Description
ListId	string	List ID
ListName	string	List Name

[Back to Class List](#)

Introduction

Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

Registering with Amazon S3

Before you can get started with `ZendService\Amazon\S3`, you must first register for an account. Please see the [S3 FAQ](#) page on the Amazon website for more information.

After registering, you will receive an application key and a secret key. You will need both to access the S3 service.

API Documentation

The `ZendService\Amazon\S3` class provides the *PHP* wrapper to the Amazon S3 REST interface. Please consult the [Amazon S3 documentation](#) for detailed description of the service. You will need to be familiar with basic concepts in order to use this service.

Features

`ZendService\Amazon\S3` provides the following functionality:

- A single point for configuring your `amazon.s3` authentication credentials that can be used across the `amazon.s3` 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.

Getting Started

Once you have registered with Amazon S3, you're ready to store your first data object on the S3. The objects on S3 are stored in containers, called "buckets". Bucket names are unique on S3, and each user can have no more than 100 buckets simultaneously. Each bucket can contain unlimited amount of objects, identified by name.

The following example demonstrates creating a bucket, storing and retrieving the data.

ZendService\Amazon\S3 Usage Example

```
1 $s3 = new ZendService\Amazon\S3($my_aws_key, $my_aws_secret_key);
2
3 $s3->createBucket("my-own-bucket");
4
5 $s3->putObject("my-own-bucket/myobject", "somedata");
6
7 echo $s3->getObject("my-own-bucket/myobject");
```

Since `ZendService\Amazon\S3` service requires authentication, you should pass your credentials (AWS key and secret key) to the constructor. If you only use one account, you can set default credentials for the service:

```
1 ZendService\Amazon\S3::setKeys($my_aws_key, $my_aws_secret_key);
2 $s3 = new ZendService\Amazon\S3();
```

Bucket operations

All objects in S3 system are stored in buckets. Bucket has to be created before any storage operation. Bucket name is unique in the system, so you can not have bucket named the same as someone else's bucket.

Bucket name can contain lowercase letters, digits, periods (.), underscores (_), and dashes (-). No other symbols allowed. Bucket name should start with letter or digit, and be 3 to 255 characters long. Names looking like an IP address (e.g. "192.168.16.255") are not allowed.

- `createBucket()` creates a new bucket.
- `cleanBucket()` removes all objects that are contained in a bucket.
- `removeBucket()` removes the bucket from the system. The bucket should be empty to be removed.

ZendServiceAmazonS3 Bucket Removal Example

```
1 $s3 = new ZendService\Amazon\S3($my_aws_key, $my_aws_secret_key);
2
```



```

3 $s3->cleanBucket("my-own-bucket");
4 $s3->removeBucket("my-own-bucket");

```

- `getBuckets()` returns the list of the names of all buckets belonging to the user.

ZendServiceAmazonS3 Bucket Listing Example

```

1 $s3 = new ZendService\Amazon\S3($my_aws_key, $my_aws_secret_key);
2
3 $list = $s3->getBuckets();
4 foreach ($list as $bucket) {
5     echo "I have bucket $bucket\n";
6 }

```

- `isBucketAvailable()` check if the bucket exists and returns TRUE if it does.

Object operations

The object is the basic storage unit in S3. Object stores unstructured data, which can be any size up to 4 gigabytes. There's no limit on how many objects can be stored on the system.

The object are contained in buckets. Object is identified by name, which can be any utf-8 string. It is common to use hierarchical names (such as *Pictures/Myself/CodingInPHP.jpg*) to organise object names. Object name is prefixed with bucket name when using object functions, so for object "mydata" in bucket "my-own-bucket" the name would be *my-own-bucket/mydata*.

Objects can be replaced (by rewriting new data with the same key) or deleted, but not modified, appended, etc. Object is always stored whole.

By default, all objects are private and can be accessed only by their owner. However, it is possible to specify object with public access, in which case it will be available through the URL: *http://s3.amazonaws.com/[bucket-name]/[object-name]*.

- `putObject($object, $data, $meta)` created an object with name `$object` (should contain the bucket name as prefix!) having `$data` as its content.

Optional `$meta` parameter is the array of metadata, which currently supports the following parameters as keys:

S3_CONTENT_TYPE_HEADER *MIME* content type of the data. If not specified, the type will be guessed according to the file extension of the object name.

S3_ACL_HEADER The access to the item. Following access constants can be used:

S3_ACL_PRIVATE Only the owner has access to the item.

S3_ACL_PUBLIC_READ Anybody can read the object, but only owner can write. This is setting may be used to store publicly accessible content.

S3_ACL_PUBLIC_WRITE Anybody can read or write the object. This policy is rarely useful.

S3_ACL_AUTH_READ Only the owner has write access to the item, and other authenticated S3 users have read access. This is useful for sharing data between S3 accounts without exposing them to the public.

By default, all the items are private.

ZendServiceAmazonS3 Public Object Example

```
1 $s3 = new ZendService\Amazon\S3($my_aws_key, $my_aws_secret_key);
2
3 $s3->putObject("my-own-bucket/Pictures/Me.png", file_get_contents("me.png"),
4     array(ZendService\Amazon\S3::S3_ACL_HEADER =>
5         ZendService\Amazon\S3::S3_ACL_PUBLIC_READ));
6 // or:
7 $s3->putFile("me.png", "my-own-bucket/Pictures/Me.png",
8     array(ZendService\Amazon\S3::S3_ACL_HEADER =>
9         ZendService\Amazon\S3::S3_ACL_PUBLIC_READ));
10 echo "Go to http://s3.amazonaws.com/my-own-bucket/Pictures/Me.png to see me!\n";
```

- `getObject($object)` retrieves object data from the storage by name.
- `removeObject($object)` removes the object from the storage.
- `getInfo($object)` retrieves the metadata information about the object. The function will return array with metadata information. Some of the useful keys are:

type The *MIME* type of the item.

size The size of the object data.

mtime UNIX-type timestamp of the last modification for the object.

etag The ETag of the data, which is the MD5 hash of the data, surrounded by quotes ("").

The function will return `FALSE` if the key does not correspond to any existing object.

- `getObjectsByBucket($bucket)` returns the list of the object keys, contained in the bucket.

ZendServiceAmazonS3 Object Listing Example

```
1 $s3 = new ZendService\Amazon\S3($my_aws_key, $my_aws_secret_key);
2
3 $list = $s3->getObjectsByBucket("my-own-bucket");
4 foreach ($list as $name) {
5     echo "I have $name key:\n";
6     $data = $s3->getObject("my-own-bucket/$name");
7     echo "with data: $data\n";
8 }
```

- `isObjectAvailable($object)` checks if the object with given name exists.
- `putFile($path, $object, $meta)` puts the content of the file in `$path` into the object named `$object`.

The optional `$meta` argument is the same as for *putObject*. If the content type is omitted, it will be guessed basing on the source file name.

Data Streaming

It is possible to get and put objects using not stream data held in memory but files or *PHP* streams. This is especially useful when file sizes are large in order not to overcome memory limits.

To receive object using streaming, use method `getObjectStream($object, $filename)`. This method will return `Zend\Http\Response\Stream`, which can be used as described in [HTTP Client Data Streaming](#) section.

ZendServiceAmazonS3 Data Streaming Example

```

1 $response = $amazon->getObjectStream("mybucket/zftest");
2 // copy file
3 copy($response->getStreamName(), "my/downloads/file");
4 // use stream
5 $fp = fopen("my/downloads/file2", "w");
6 stream_copy_to_stream($response->getStream(), $fp);

```

Second parameter for `getObjectStream()` is optional and specifies target file to write the data. If not specified, temporary file is used, which will be deleted after the response object is destroyed.

To send object using streaming, use `putFileStream()` which has the same signature as `putFile()` but will use streaming and not read the file into memory.

Also, you can pass stream resource to `putObject()` method data parameter, in which case the data will be read from the stream when sending the request to the server.

Stream wrapper

In addition to the interfaces described above, `ZendService\Amazon\S3` also supports operating as a stream wrapper. For this, you need to register the client object as the stream wrapper:

ZendService\Amazon\S3 Streams Example

```

1 $s3 = new ZendService\Amazon\S3($my_aws_key, $my_aws_secret_key);
2
3 $s3->registerStreamWrapper("s3");
4
5 mkdir("s3://my-own-bucket");
6 file_put_contents("s3://my-own-bucket/testdata", "mydata");
7
8 echo file_get_contents("s3://my-own-bucket/testdata");

```

Directory operations (`mkdir`, `rmdir`, `opendir`, etc.) will operate on buckets and thus their arguments should be of the form of `s3://bucketname`. File operations operate on objects. Object creation, reading, writing, deletion, stat and directory listing is supported.

Introduction

Amazon Simple Queue Service (Amazon SQS) offers a reliable, highly scalable, hosted queue for storing messages as they travel between computers. By using Amazon SQS, developers can simply move data between distributed components of their applications that perform different tasks, without losing messages or requiring each component to be always available. Amazon SQS makes it easy to build an automated workflow, working in close conjunction with the Amazon Elastic Compute Cloud (Amazon EC2) and the other AWS infrastructure web services.

Amazon SQS works by exposing Amazon's web-scale messaging infrastructure as a web service. Any computer on the Internet can add or read messages without any installed software or special firewall configurations. Components of applications using Amazon SQS can run independently, and do not need to be on the same network, developed with the same technologies, or running at the same time.

Registering with Amazon SQS

Before you can get started with `ZendService\Amazon\Sqs`, you must first register for an account. Please see the [SQS FAQ](#) page on the Amazon website for more information.

After registering, you will receive an application key and a secret key. You will need both to access the SQS service.

API Documentation

The `ZendService\Amazon\Sqs` class provides the *PHP* wrapper to the Amazon SQS REST interface. Please consult the [Amazon SQS documentation](#) for detailed description of the service. You will need to be familiar with basic concepts in order to use this service.

Features

`ZendService\Amazon\Sqs` provides the following functionality:

- A single point for configuring your amazon.sqs authentication credentials that can be used across the amazon.sqs 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.

Getting Started

Once you have registered with Amazon SQS, you're ready to create your queue and store some messages on SQS. Each queue can contain unlimited amount of messages, identified by name.

The following example demonstrates creating a queue, storing and retrieving messages.

ZendService\Amazon\Sqs Usage Example

```
1 $sqs = new ZendService\Amazon\Sqs($my_aws_key, $my_aws_secret_key);
2
3 $queue_url = $sqs->create('test');
4
5 $message = 'this is a test';
6 $message_id = $sqs->send($queue_url, $message);
7
8 foreach ($sqs->receive($queue_url) as $message) {
9     echo $message['body'].'<br/>';
10 }
```

Since the `ZendService\Amazon\Sqs` service requires authentication, you should pass your credentials (AWS key and secret key) to the constructor. If you only use one account, you can set default credentials for the service:

```
1 ZendService\Amazon\Sqs::setKeys($my_aws_key, $my_aws_secret_key);
2 $sqs = new ZendService\Amazon\Sqs();
```

Queue operations

All messages SQS are stored in queues. A queue has to be created before any message operations. Queue names must be unique under your access key and secret key.

Queue names can contain lowercase letters, digits, periods (.), underscores (_), and dashes (-). No other symbols allowed. Queue names can be a maximum of 80 characters.

- `create()` creates a new queue.
- `delete()` removes all messages in the queue.

ZendServiceAmazonSqs Queue Removal Example

```
1 $sqs = new ZendService\Amazon\Sqs($my_aws_key, $my_aws_secret_key);
2 $queue_url = $sqs->create('test_1');
3 $sqs->delete($queue_url);
```

- `count()` gets the approximate number of messages in the queue.

ZendServiceAmazonSqs Queue Count Example

```
1 $sqs = new ZendService\Amazon\Sqs($my_aws_key, $my_aws_secret_key);
2 $queue_url = $sqs->create('test_1');
3 $sqs->send($queue_url, 'this is a test');
4 $count = $sqs->count($queue_url); // Returns '1'
```

- `getQueues()` returns the list of the names of all queues belonging to the user.

ZendServiceAmazonSqs Queue Listing Example

```
1 $sqs = new ZendService\Amazon\Sqs($my_aws_key, $my_aws_secret_key);
2 $list = $sqs->getQueues();
3 foreach ($list as $queue) {
4     echo "I have queue $queue\n";
5 }
```

Message operations

After a queue is created, simple messages can be sent into the queue then received at a later point in time. Messages can be up to 8KB in length. If longer messages are needed please see [S3](#). There is no limit to the number of messages a queue can contain.

- `send($queue_url, $message)` send the `$message` to the `$queue_url` SQS queue *URL*.

ZendServiceAmazonSqs Message Send Example

```
1 $sqs = new ZendService\Amazon\Sqs($my_aws_key, $my_aws_secret_key);
2 $queue_url = $sqs->create('test_queue');
3 $sqs->send($queue_url, 'this is a test message');
```

- `receive($queue_url)` retrieves messages from the queue.

ZendServiceAmazonSqs Message Receive Example

```
1 $sqs = new ZendService\Amazon\Sqs($my_aws_key, $my_aws_secret_key);
2 $queue_url = $sqs->create('test_queue');
3 $sqs->send($queue_url, 'this is a test message');
4 foreach ($sqs->receive($queue_url) as $message) {
```

```
5     echo "got message ".$message['body'].'<br/>';
6 }
```

- `deleteMessage($queue_url, $handle)` deletes a message from a queue. A message must first be received using the `receive()` method before it can be deleted.

ZendServiceAmazonSqs Message Delete Example

```
1 $sqs = new ZendService\Amazon\Sqs($my_aws_key, $my_aws_secret_key);
2 $queue_url = $sqs->create('test_queue');
3 $sqs->send($queue_url, 'this is a test message');
4 foreach ($sqs->receive($queue_url) as $message) {
5     echo "got message ".$message['body'].'<br/>';
6
7     if ($sqs->deleteMessage($queue_url, $message['handle'])) {
8         echo "Message deleted";
9     }
10    else {
11        echo "Message not deleted";
12    }
13 }
```


Introduction

ZendService\Amazon\Ec2 provides an interface to Amazon Elastic Cloud Computing (EC2).

What is Amazon Ec2?

Amazon EC2 is a web service that enables you to launch and manage server instances in Amazon's data centers using *APIs* or available tools and utilities. You can use Amazon EC2 server instances at any time, for as long as you need, and for any legal purpose.

Static Methods

To make using the Ec2 class easier to use there are two static methods that can be invoked from any of the Ec2 Elements. The first static method is *setKeys* which will define your AWS Access Keys as default keys. When you then create any new object you don't need to pass in any keys to the constructor.

setKeys() Example

```
1 ZendService\Amazon\Ec2\Ebs::setKeys('aws_key', 'aws_secret_key');
```

To set the region that you are working in you can call the *setRegion* to set which Amazon Ec2 Region you are working in. Currently there is only two region available us-east-1 and eu-west-1. If an invalid value is passed it will throw an exception stating that.

setRegion() Example

```
1 ZendService\Amazon\Ec2\Ebs::setRegion('us-east-1');
```

Note: Set Amazon Ec2 Region

Alternatively you can set the region when you create each class as the third parameter in the constructor method.

ZendService\Amazon\Ec2: CloudWatch Monitoring

Amazon CloudWatch is an easy-to-use web service that provides comprehensive monitoring for Amazon Elastic Compute Cloud (Amazon *EC2*) and Elastic Load Balancing. For more details information check out the [Amazon CloudWatch Developers Guide](#)

CloudWatch Usage

Listing Available Metrics

`listMetrics()` returns a list of up to 500 valid metrics for which there is recorded data available to a you and a `NextToken` string that can be used to query for the next set of results.

```
1 $ec2_ebs = new ZendService\Amazon\Ec2\CloudWatch('aws_key','aws_secret_key');  
2 $return = $ec2_ebs->listMetrics();
```

Return Statistics for a given metric

`getMetricStatistics()` Returns data for one or more statistics of given a metric.

Note: The maximum number of datapoints that the Amazon CloudWatch service will return in a single `GetMetricStatistics` request is 1,440. If a request is made that would generate more datapoints than this amount, Amazon CloudWatch will return an error. You can alter your request by narrowing the time range (`StartTime`, `EndTime`) or increasing the `Period` in your single request. You may also get all of the data at the granularity you originally asked for by making multiple requests with adjacent time ranges.

`getMetricStatistics()` only requires two parameters but it also has four additional parameters that are optional.

- **Required:**

- **MeasureName** The measure name that corresponds to the measure for the gathered metric. Valid *EC2* Values are *CPUUtilization*, *NetworkIn*, *NetworkOut*, *DiskWriteOps*, *DiskReadBytes*, *DiskReadOps*, *DiskWriteBytes*. Valid Elastic Load Balancing Metrics are *Latency*, *RequestCount*, *HealthyHostCount*, *UnHealthyHostCount*. [For more information click here](#)
- **Statistics** The statistics to be returned for the given metric. Valid values are *Average*, *Maximum*, *Minimum*, *Samples*, *Sum*. You can specify this as a string or as an array of values. If you don't specify one it will default to *Average* instead of failing out. If you specify an incorrect option it will just skip it. [For more information click here](#)
- **Optional:**
- **Dimensions** Amazon CloudWatch allows you to specify one Dimension to further filter metric data on. If you don't specify a dimension, the service returns the aggregate of all the measures with the given measure name and time range.
- **Unit** The standard unit of Measurement for a given Measure. Valid Values: *Seconds*, *Percent*, *Bytes*, *Bits*, *Count*, *Bytes/Second*, *Bits/Second*, *Count/Second*, and *None*. Constraints: When using *count/second* as the unit, you should use *Sum* as the statistic instead of *Average*. Otherwise, the sample returns as equal to the number of requests instead of the number of 60-second intervals. This will cause the *Average* to always equals one when the unit is *count/second*.
- **StartTime** The timestamp of the first datapoint to return, inclusive. For example, *2008-02-26T19:00:00+00:00*. We round your value down to the nearest minute. You can set your start time for more than two weeks in the past. However, you will only get data for the past two weeks. (in *ISO 8601* format). Constraints: Must be before *EndTime*.
- **EndTime** The timestamp to use for determining the last datapoint to return. This is the last datapoint to fetch, exclusive. For example, *2008-02-26T20:00:00+00:00* (in *ISO 8601* format).

```
1 $sec2_ebs = new ZendService\Amazon\Ec2\CloudWatch('aws_key','aws_secret_key');
2 $return = $sec2_ebs->getMetricStatistics(
3     array('MeasureName' => 'NetworkIn',
4           'Statistics' => array('Average')));
```

ZendService\Amazon\Ec2: Elastic Block Storage (EBS)

Amazon Elastic Block Store (Amazon EBS) is a new type of storage designed specifically for Amazon EC2 instances. Amazon EBS allows you to create volumes that can be mounted as devices by Amazon EC2 instances. Amazon EBS volumes behave like raw unformatted external block devices. They have user supplied device names and provide a block device interface. You can load a file system on top of Amazon EBS volumes, or use them just as you would use a block device.

You can create up to twenty Amazon EBS volumes of any size (from one GiB up to one TiB). Each Amazon EBS volume can be attached to any Amazon EC2 instance in the same Availability Zone or can be left unattached.

Amazon EBS provides the ability to create snapshots of your Amazon EBS volumes to Amazon S3. You can use these snapshots as the starting point for new Amazon EBS volumes and can protect your data for long term durability.

Create EBS Volumes and Snapshots

Create a new EBS Volume

Creating a brand new EBS Volume requires the size and which zone you want the EBS Volume to be in.

createNewVolume will return an array containing information about the new Volume which includes the volumeId, size, zone, status and createTime.

```
1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key','aws_secret_key');  
2 $return = $sec2_ebs->createNewVolume(40, 'us-east-1a');
```

Create an EBS Volume from a Snapshot

Creating an EBS Volume from a snapshot requires the snapshot_id and which zone you want the EBS Volume to be in.

createVolumeFromSnapshot will return an array containing information about the new Volume which includes the volumeId, size, zone, status, createTime and snapshotId.

```
1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key','aws_secret_key');
2 $return = $sec2_ebs->createVolumeFromSnapshot('snap-78a54011','us-east-1a');
```

Create a Snapshot of an EBS Volume

Creating a Snapshot of an EBS Volume requires the volumeId of the EBS Volume.

createSnapshot will return an array containing information about the new Volume Snapshot which includes the snapshotId, volumeId, status, startTime and progress.

```
1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key','aws_secret_key');
2 $return = $sec2_ebs->createSnapshot('volumeId');
```

Describing EBS Volumes and Snapshots

Describing an EBS Volume

describeVolume allows you to get information on an EBS Volume or a set of EBS Volumes. If nothing is passed in then it will return all EBS Volumes. If only one EBS Volume needs to be described a string can be passed in while an array of EBS Volume Id's can be passed in to describe them.

describeVolume will return an array with information about each Volume which includes the volumeId, size, status and createTime. If the volume is attached to an instance, an addition value of attachmentSet will be returned. The attachment set contains information about the instance that the EBS Volume is attached to, which includes volumeId, instanceId, device, status and attachTime.

```
1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key','aws_secret_key');
2 $return = $sec2_ebs->describeVolume('volumeId');
```

Describe Attached Volumes

To return a list of EBS Volumes currently attached to a running instance you can call this method. It will only return EBS Volumes attached to the instance with the passed in instanceId.

describeAttachedVolumes returns the same information as the *describeVolume* but only for the EBS Volumes that are currently attached to the specified instanceId.

```
1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key','aws_secret_key');
2 $return = $sec2_ebs->describeAttachedVolumes('instanceId');
```

Describe an EBS Volume Snapshot

describeSnapshot allows you to get information on an EBS Volume Snapshot or a set of EBS Volume Snapshots. If nothing is passed in then it will return information about all EBS Volume Snapshots. If only one EBS Volume Snapshot needs to be described its snapshotId can be passed in while an array of EBS Volume Snapshot Id's can be passed in to describe them.

describeSnapshot will return an array containing information about each EBS Volume Snapshot which includes the snapshotId, volumeId, status, startTime and progress.

```

1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key', 'aws_secret_key');
2 $return = $sec2_ebs->describeSnapshot('volumeId');

```

Attach and Detaching Volumes from Instances

Attaching an EBS Volume

attachVolume will attach an EBS Volume to a running Instance. To attach a volume you need to specify the volumeId, the instanceId and the device (ex: **/dev/sdh**).

attachVolume will return an array with information about the attach status which contains volumeId, instanceId, device, status and attachTime

```

1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key', 'aws_secret_key');
2 $return = $sec2_ebs->attachVolume('volumeId', 'instanceid', '/dev/sdh');

```

Detaching an EBS Volume

detachVolume will detach an EBS Volume from a running Instance. *detachVolume* requires that you specify the volumeId with the optional instanceId and device name that was passed when attaching the volume. If you need to force the detachment you can set the fourth parameter to be TRUE and it will force the volume to detach.

detachVolume returns an array containing status information about the EBS Volume which includes volumeId, instanceId, device, status and attachTime.

```

1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key', 'aws_secret_key');
2 $return = $sec2_ebs->detachVolume('volumeId');

```

Note: Forced Detach

You should only force a detach if the previous detachment attempt did not occur cleanly (logging into an instance, unmounting the volume, and detaching normally). This option can lead to data loss or a corrupted file system. Use this option only as a last resort to detach a volume from a failed instance. The instance will not have an opportunity to flush file system caches or file system meta data. If you use this option, you must perform file system check and repair procedures.

Deleting EBS Volumes and Snapshots

Deleting an EBS Volume

deleteVolume will delete an unattached EBS Volume.

deleteVolume will return boolean TRUE or FALSE.

```

1 $sec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key', 'aws_secret_key');
2 $return = $sec2_ebs->deleteVolume('volumeId');

```

Deleting an EBS Volume Snapshot

deleteSnapshot will delete an EBS Volume Snapshot.

deleteSnapshot returns boolean TRUE or FALSE.

```
1 $ec2_ebs = new ZendService\Amazon\Ec2\Ebs('aws_key','aws_secret_key');  
2 $return = $ec2_ebs->deleteSnapshot('snapshotId');
```

ZendService\Amazon\Ec2: Elastic IP Addresses

By default, all Amazon EC2 instances are assigned two IP addresses at launch: a private (RFC 1918) address and a public address that is mapped to the private IP address through Network Address Translation (NAT).

If you use dynamic DNS to map an existing DNS name to a new instance's public IP address, it might take up to 24 hours for the IP address to propagate through the Internet. As a result, new instances might not receive traffic while terminated instances continue to receive requests.

To solve this problem, Amazon EC2 provides elastic IP addresses. Elastic IP addresses are static IP addresses designed for dynamic cloud computing. Elastic IP addresses are associated with your account, not specific instances. Any elastic IP addresses that you associate with your account remain associated with your account until you explicitly release them. Unlike traditional static IP addresses, however, elastic IP addresses allow you to mask instance or Availability Zone failures by rapidly remapping your public IP addresses to any instance in your account.

Allocating a new Elastic IP

allocate will assign your account a new Elastic IP Address.

allocate returns the newly allocated ip.

```
1 $ec2_eip = new ZendService\Amazon\Ec2\Elasticip('aws_key', 'aws_secret_key');
2 $ip = $ec2_eip->allocate();
3
4 // print out your newly allocated elastic ip address;
5 print $ip;
```

Describing Allocated Elastic IP Addresses

describe has an optional parameter to describe all of your allocated Elastic IP addresses or just some of your allocated addresses.

describe returns an array that contains information on each Elastic IP Address which contains the publicIp and the instanceId if it is associated.

```
1 $ec2_eip = new ZendService\Amazon\Ec2\Elasticip('aws_key', 'aws_secret_key');
2 // describe all
3 $ips = $ec2_eip->describe();
4
5 // describe a subset
6 $ips = $ec2_eip->describe(array('ip1', 'ip2', 'ip3'));
7
8 // describe a single ip address
9 $ip = $ec2_eip->describe('ip1');
```

Releasing Elastic IP

release will release an Elastic IP to Amazon.

Returns a boolean TRUE or FALSE.

```
1 $ec2_eip = new ZendService\Amazon\Ec2\Elasticip('aws_key', 'aws_secret_key');
2 $ec2_eip->release('ipaddress');
```

Associates an Elastic IP to an Instance

associate will assign an Elastic IP to an already running instance.

Returns a boolean TRUE or FALSE.

```
1 $ec2_eip = new ZendService\Amazon\Ec2\Elasticip('aws_key', 'aws_secret_key');
2 $ec2_eip->associate('instance_id', 'ipaddress');
```

Disassociate an Elastic IP from an instance

disassociate will disassociate an Elastic IP from an instance. If you terminate an Instance it will automatically disassociate the Elastic IP address for you.

Returns a boolean TRUE or FALSE.

```
1 $ec2_eip = new ZendService\Amazon\Ec2\Elasticip('aws_key', 'aws_secret_key');
2 $ec2_eip->disassociate('ipaddress');
```

ZendService\Amazon\Ec2: Instances

Instance Types

Amazon EC2 instances are grouped into two families: standard and High-CPU. Standard instances have memory to *CPU* ratios suitable for most general purpose applications; High-CPU instances have proportionally more *CPU* resources than memory (RAM) and are well suited for compute-intensive applications. When selecting instance types, you might want to use less powerful instance types for your web server instances and more powerful instance types for your database instances. Additionally, you might want to run *CPU* instance types for *CPU*-intensive data processing tasks.

One of the advantages of EC2 is that you pay by the instance hour, which makes it convenient and inexpensive to test the performance of your application on different instance families and types. One good way to determine the most appropriate instance family and instance type is to launch test instances and benchmark your application.

Note: Instance Types

The instance types are defined as constants in the code. Column eight in the table is the defined constant name

Table 341.1: Available Instance Types

Type	CPU	Memory	Storage	Platform	I/O	Name	Constant Name
Small	1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)	1.7 GB	160 GB instance storage (150 GB plus 10 GB root partition)	32-bit	Mod-erate	m1.small	ZendServiceAmazonEc2Instance::SMALL
Large	4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)	7.5 GB	850 GB instance storage (2 x 420 GB plus 10 GB root partition)	64-bit	High	m1.large	ZendServiceAmazonEc2Instance::LARGE
Extra Large	8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each)	15 GB	1,690 GB instance storage (4 x 420 GB plus 10 GB root partition)	64-bit	High	m1.xlarge	ZendServiceAmazonEc2Instance::XLARGE
High-CPU Medium	5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each)	1.7 GB	350 GB instance storage (340 GB plus 10 GB root partition)	32-bit	Mod-erate	c1.medium	ZendServiceAmazonEc2Instance::HCPU_MEDIUM
High-CPU Extra Large	20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each)	7 GB	1,690 GB instance storage (4 x 420 GB plus 10 GB root partition)	64-bit	High	c1.xlarge	ZendServiceAmazonEc2Instance::HCPU_XLARGE

Running Amazon EC2 Instances

This section describes the operation methods for maintaining Amazon EC2 Instances.

Starting New Ec2 Instances

run will launch a specified number of EC2 Instances. *run* takes an array of parameters to start, below is a table containing the valid values.

Table 341.2: Valid Run Options

Name	Description	Required
imageId	ID of the AMI with which to launch instances.	Yes
minCount	Minimum number of instances to launch. Default: 1	No
maxCount	Maximum number of instances to launch. Default: 1	No
keyName	Name of the key pair with which to launch instances. If you do not provide a key, all instances will be inaccessible.	No
securityGroup	Names of the security groups with which to associate the instances.	No
userData	The user data available to the launched instances. This should not be Base64 encoded.	No
instanceType	Specifies the instance type. Default: m1.small	No
placement	Specifies the availability zone in which to launch the instance(s). By default, Amazon EC2 selects an availability zone for you.	No
kernelId	The ID of the kernel with which to launch the instance.	No
ramdiskId	The ID of the RAM disk with which to launch the instance.	No
blockDevice-VirtualName	Specifies the virtual name to map to the corresponding device name. For example: instancestore0	No
blockDevice-Name	Specifies the device to which you are mapping a virtual name. For example: sdb	No
monitor	Turn on AWS CloudWatch Instance Monitoring	No

run will return information about each instance that is starting up.

```

1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->run(array('imageId' => 'ami-509320',
4                                     'keyName' => 'myKey',
5                                     'securityGroup' => array('web',
6                                                             'default')));

```

Rebooting an Ec2 Instances

reboot will reboot one or more instances.

This operation is asynchronous; it only queues a request to reboot the specified instance(s). The operation will succeed if the instances are valid and belong to the user. Requests to reboot terminated instances are ignored.

reboot returns boolean TRUE or FALSE

```

1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->reboot('instanceId');

```

Terminating an Ec2 Instances

terminate shuts down one or more instances. This operation is idempotent; if you terminate an instance more than once, each call will succeed.

terminate returns boolean TRUE or FALSE

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                                         'aws_secret_key');
3 $return = $sec2_instance->terminate('instanceId');
```

Note: Terminated Instances

Terminated instances will remain visible after termination (approximately one hour).

Amazon Instance Utilities

In this section you will find out how to retrieve information, the console output and see if an instance contains a product code.

Describing Instances

describe returns information about instances that you own.

If you specify one or more instance IDs, Amazon EC2 returns information for those instances. If you do not specify instance IDs, Amazon EC2 returns information for all relevant instances. If you specify an invalid instance ID, a fault is returned. If you specify an instance that you do not own, it will not be included in the returned results.

describe will return an array containing information on the instance.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                                         'aws_secret_key');
3 $return = $sec2_instance->describe('instanceId');
```

Note: Terminated Instances

Recently terminated instances might appear in the returned results. This interval is usually less than one hour. If you do not want terminated instances to be returned, pass in a second variable of boolean `TRUE` to *describe* and the terminated instances will be ignored.

Describing Instances By Image Id

describeByImageId is functionally the same as *describe* but it will only return the instances that are using the provided *imageId*.

describeByImageId will return an array containing information on the instances there were started by the passed in *imageId*

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                                         'aws_secret_key');
3 $return = $sec2_instance->describeByImageId('imageId');
```

Note: Terminated Instances

Recently terminated instances might appear in the returned results. This interval is usually less than one hour. If you do not want terminated instances to be returned, pass in a second variable of boolean `TRUE` to *describe* and the terminated instances will be ignored.

Retrieving Console Output

consoleOutput retrieves console output for the specified instance.

Instance console output is buffered and posted shortly after instance boot, reboot, and termination. Amazon EC2 preserves the most recent 64 KB output which will be available for at least one hour after the most recent post.

consoleOutput returns an array containing the *instanceId*, *timestamp* from the last output and the *output* from the console.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                                         'aws_secret_key');
3 $return = $sec2_instance->consoleOutput('instanceId');
```

Confirm Product Code on an Instance

confirmProduct returns `TRUE` if the specified product code is attached to the specified instance. The operation returns `FALSE` if the product code is not attached to the instance.

The *confirmProduct* operation can only be executed by the owner of the *AMI*. This feature is useful when an *AMI* owner is providing support and wants to verify whether a user's instance is eligible.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                                         'aws_secret_key');
3 $return = $sec2_instance->confirmProduct('productCode', 'instanceId');
```

Turn on CloudWatch Monitoring on an Instance(s)

monitor returns the list of instances and their current state of the CloudWatch Monitoring. If the instance does not currently have Monitoring enabled it will be turned on.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                                         'aws_secret_key');
3 $return = $sec2_instance->monitor('instanceId');
```

Turn off CloudWatch Monitoring on an Instance(s)

monitor returns the list of instances and their current state of the CloudWatch Monitoring. If the instance currently has Monitoring enabled it will be turned off.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance('aws_key',
2                                                         'aws_secret_key');
3 $return = $sec2_instance->unmonitor('instanceId');
```

ZendService\Amazon\Ec2: Regions and Availability Zones

Amazon EC2 provides the ability to place instances in different regions and Availability Zones. Regions are dispersed in separate geographic areas or countries. Availability Zones are located within regions and are engineered to be insulated from failures in other Availability Zones and provide inexpensive low latency network connectivity to other Availability Zones in the same region. By launching instances in separate Availability Zones, you can protect your applications from the failure of a single Availability Zone.

Amazon EC2 Regions

Amazon EC2 provides multiple regions so you can launch Amazon EC2 instances in locations that meet your requirements. For example, you might want to launch instances in Europe to be closer to your European customers or to meet legal requirements.

Each Amazon EC2 region is designed to be completely isolated from the other Amazon EC2 regions. This achieves the greatest possible failure independence and stability, and it makes the locality of each EC2 resource unambiguous.

Viewing the available regions

describe is used to find out which regions your account has access to.

describe will return an array containing information about which regions are available. Each array will contain region-Name and regionUrl.

```
1 $ec2_region = new ZendService\Amazon\Ec2\Region('aws_key','aws_secret_key');
2 $regions = $ec2_region->describe();
3
4 foreach ($regions as $region) {
5     print $region['regionName'] . ' -- ' . $region['regionUrl'] . '<br />';
6 }
```

Amazon EC2 Availability Zones

When you launch an instance, you can optionally specify an Availability Zone. If you do not specify an Availability Zone, Amazon EC2 selects one for you in the region that you are using. When launching your initial instances, we recommend accepting the default Availability Zone, which allows Amazon EC2 to select the best Availability Zone for you based on system health and available capacity. Even if you have other instances running, you might consider not specifying an Availability Zone if your new instances do not need to be close to, or separated from, your existing instances.

Viewing the available zones

describe is used to find out which what the status is of each availability zone.

describe will return an array containing information about which zones are available. Each array will contain zone-Name and zoneState.

```
1 $ec2_zones = new ZendService\Amazon\Ec2\Availabilityzones('aws_key',
2                                                         'aws_secret_key');
3 $zones = $ec2_zones->describe();
4
5 foreach ($zones as $zone) {
6     print $zone['zoneName'] . ' -- ' . $zone['zoneState'] . '<br />';
7 }
```

ZendService\Amazon\Ec2: Reserved Instances

With Amazon *EC2* Reserved Instances, you can make a low one-time payment for each instance to reserve and receive a significant discount on the hourly usage charge for that instance.

Amazon *EC2* Reserved Instances are based on instance type and location (region and Availability Zone) for a specified period of time (e.g., 1 year or 3 years) and are only available for Linux or UNIX instances.

How Reserved Instances are Applied

Reserved Instances are applied to instances that meet the type/location criteria during the specified period. In this example, a user is running the following instances:

- m1.small instances in Availability Zone us-east-1a (4)
- c1.medium instances in Availability Zone us-east-1b (4)
- c1.xlarge instances in Availability Zone us-east-1b (2)

The user then purchases the following Reserved Instances.

- m1.small instances in Availability Zone us-east-1a (2)
- c1.medium instances in Availability Zone us-east-1a (2)
- m1.xlarge instances in Availability Zone us-east-1a (2)

Amazon *EC2* applies the two m1.small Reserved Instances to two of the instances in Availability Zone us-east-1a. Amazon *EC2* doesn't apply the two c1.medium Reserved Instances because the c1.medium instances are in a different Availability Zone and does not apply the m1.xlarge Reserved Instances because there are no running m1.xlarge instances.

Reserved Instances Usage

Describes Reserved Instances that you purchased

`describeInstances()` will return information about a reserved instance or instances that you purchased.

`describeInstances()` returns a multi-dimensional array that contains `reservedInstancesId`, `instanceType`, `availabilityZone`, `duration`, `fixedPrice`, `usagePrice`, `productDescription`, `instanceCount` and `state`.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance\Reserved('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->describeInstances('instanceId');
```

Describe current Reserved Instance Offerings available

`describeOfferings()` Describes Reserved Instance offerings that are available for purchase. With Amazon *EC2* Reserved Instances, you purchase the right to launch Amazon *EC2* instances for a period of time (without getting insufficient capacity errors) and pay a lower usage rate for the actual time used.

`describeOfferings()` returns a multi-dimensional array that contains `reservedInstancesId`, `instanceType`, `availabilityZone`, `duration`, `fixedPrice`, `usagePrice` and `productDescription`.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance\Reserved('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->describeOfferings();
```

Turn off CloudWatch Monitoring on an Instance(s)

`purchaseOffering()` Purchases a Reserved Instance for use with your account. With Amazon *EC2* Reserved Instances, you purchase the right to launch Amazon *EC2* instances for a period of time (without getting insufficient capacity errors) and pay a lower usage rate for the actual time used.

`purchaseOffering()` returns the `reservedInstanceId`.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance\Reserved('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->purchaseOffering('offeringId', 'instanceCount');
```

ZendService\Amazon\Ec2: Security Groups

A security group is a named collection of access rules. These access rules specify which ingress (i.e., incoming) network traffic should be delivered to your instance. All other ingress traffic will be discarded.

You can modify rules for a group at any time. The new rules are automatically enforced for all running instances and instances launched in the future.

Note: Maximum Security Groups

You can create up to 100 security groups.

Security Group Maintenance

Create a new Security Group

create a new security group. Every instance is launched in a security group. If no security group is specified during launch, the instances are launched in the default security group. Instances within the same security group have unrestricted network access to each other. Instances will reject network access attempts from other instances in a different security group.

create returns boolean TRUE or FALSE

```
1 $sec2_sg = new ZendService\Amazon\Ec2\Securitygroups('aws_key',  
2                                     'aws_secret_key');  
3 $return = $sec2_sg->create('mygroup', 'my group description');
```

Describe a Security Group

describe returns information about security groups that you own.

If you specify security group names, information about those security groups is returned. Otherwise, information for all security groups is returned. If you specify a group that does not exist, a fault is returned.

describe will return an array containing information about security groups which includes the ownerId, groupName, groupDescription and an array containing all the rules for that security group.

```
1 $sec2_sg = new ZendService\Amazon\Ec2\Securitygroups('aws_key',
2                                                     'aws_secret_key');
3 $return = $sec2_sg->describe('mygroup');
```

Delete a Security Group

delete will remove the security group. If you attempt to delete a security group that contains instances, a fault is returned. If you attempt to delete a security group that is referenced by another security group, a fault is returned. For example, if security group B has a rule that allows access from security group A, security group A cannot be deleted until the allow rule is removed.

delete returns boolean TRUE or FALSE.

```
1 $sec2_sg = new ZendService\Amazon\Ec2\Securitygroups('aws_key',
2                                                     'aws_secret_key');
3 $return = $sec2_sg->delete('mygroup');
```

Authorizing Access

Authorizing by IP

authorizeIp Adds permissions to a security group based on an IP address, protocol type and port range.

Permissions are specified by the IP protocol (TCP, UDP or ICMP), the source of the request (by IP range or an Amazon EC2 user-group pair), the source and destination port ranges (for *TCP* and *UDP*), and the ICMP codes and types (for *ICMP*). When authorizing *ICMP*, -1 can be used as a wildcard in the type and code fields.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

authorizeIp returns boolean TRUE or FALSE

```
1 $sec2_sg = new ZendService\Amazon\Ec2\Securitygroups('aws_key',
2                                                     'aws_secret_key');
3 $return = $sec2_sg->authorizeIp('mygroup',
4                                'protocol',
5                                'fromPort',
6                                'toPort',
7                                'ipRange');
```

Authorize By Group

authorizeGroup Adds permissions to a security group.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

authorizeGroup returns boolean TRUE or FALSE.

```

1 $sec2_sg = new ZendService\Amazon\Ec2\Securitygroups('aws_key',
2                                                     'aws_secret_key');
3 $return = $sec2_sg->authorizeGroup('mygroup', 'securityGroupName', 'ownerId');
```

Revoking Access

Revoke by IP

revokeIp Revokes permissions to a security group based on an IP address, protocol type and port range. The permissions used to revoke must be specified using the same values used to grant the permissions.

Permissions are specified by the IP protocol (TCP, UDP or ICMP), the source of the request (by IP range or an Amazon EC2 user-group pair), the source and destination port ranges (for *TCP* and *UDP*), and the ICMP codes and types (for *ICMP*). When authorizing *ICMP*, -1 can be used as a wildcard in the type and code fields.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

revokeIp returns boolean TRUE or FALSE

```

1 $sec2_sg = new ZendService\Amazon\Ec2\Securitygroups('aws_key',
2                                                     'aws_secret_key');
3 $return = $sec2_sg->revokeIp('mygroup',
4                             'protocol',
5                             'fromPort',
6                             'toPort',
7                             'ipRange');
```

Revoke By Group

revokeGroup Adds permissions to a security group. The permissions to revoke must be specified using the same values used to grant the permissions.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

revokeGroup returns boolean TRUE or FALSE.

```

1 $sec2_sg = new ZendService\Amazon\Ec2\Securitygroups('aws_key',
2                                                     'aws_secret_key');
3 $return = $sec2_sg->revokeGroup('mygroup', 'securityGroupName', 'ownerId');
```

ZendService\Amazon\Ec2: Windows Instances

Using Amazon EC2 instances running Windows is similar to using instances running Linux and UNIX. The following are the major differences between instances that use Linux or UNIX and Windows:

- Remote Desktop—To access Windows instances, you use Remote Desktop instead of SSH.
- Administrative Password—To access Windows instances the first time, you must obtain the administrative password using the `ec2-get-password` command.
- Simplified Bundling—To bundle a Windows instance, you use a single command that shuts down the instance, saves it as an *AMI*, and restarts it.

As part of this service, Amazon EC2 instances can now run Microsoft Windows Server 2003. Our base Windows image provides you with most of the common functionality associated with Windows. However, if you require more than two concurrent Windows users or need to leverage applications that require *LDAP*, Kerberos, *RADIUS*, or other credential services, you must use Windows with Authentication Services. For example, Microsoft Exchange Server and Microsoft SharePoint Server require Windows with Authentication Services.

Note: To get started using Windows instances, we recommend using the *AWS* Management Console. There are differences in pricing between Windows and Windows with Authentication Services instances. For information on pricing, go to the Amazon EC2 Product Page.

Amazon EC2 currently provides the following Windows *AMIs*:

- Windows Authenticated (32-bit)
- Windows Authenticated (64-bit)
- Windows Anonymous (32-bit)
- Windows Anonymous (64-bit)

The Windows public *AMIs* that Amazon provides are unmodified versions of Windows with the following two exceptions: we added drivers to improve the networking and disk I/O performance and we created the Amazon EC2 configuration service. The Amazon EC2 configuration service performs the following functions:

- Randomly sets the Administrator password on initial launch, encrypts the password with the user's SSH key, and reports it to the console. This operation happens upon initial *AMI* launch. If you change the password, *AMIs* that are created from this instance use the new password.
- Configures the computer name to the internal DNS name. To determine the internal DNS name, see Using Instance Addressing.
- Sends the last three system and application errors from the event log to the console. This helps developers to identify problems that caused an instance to crash or network connectivity to be lost.

Windows Instances Usage

Bundles an Amazon EC2 instance running Windows

`bundle()` has three required parameters and one optional

- **instanceId** The instance you want to bundle
- **s3Bucket** Where you want the ami to live on S3
- **s3Prefix** The prefix you want to assign to the *AMI* on S3
- **uploadExpiration** The expiration of the upload policy. Amazon recommends 12 hours or longer. This is based in number of minutes. Default is 1440 minutes (24 hours)

`bundle()` returns a multi-dimensional array that contains `instanceId`, `bundleId`, `state`, `startTime`, `updateTime`, `progress`, `s3Bucket` and `s3Prefix`.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance\Windows('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->bundle('instanceId', 's3Bucket', 's3Prefix');
```

Describes current bundling tasks

`describeBundle()` Describes current bundling tasks

`describeBundle()` returns a multi-dimensional array that contains `instanceId`, `bundleId`, `state`, `startTime`, `updateTime`, `progress`, `s3Bucket` and `s3Prefix`.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance\Windows('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->describeBundle('bundleId');
```

Cancels an Amazon EC2 bundling operation

`cancelBundle()` Cancels an Amazon EC2 bundling operation

`cancelBundle()` returns a multi-dimensional array that contains `instanceId`, `bundleId`, `state`, `startTime`, `updateTime`, `progress`, `s3Bucket` and `s3Prefix`.

```
1 $sec2_instance = new ZendService\Amazon\Ec2\Instance\Windows('aws_key',
2                                     'aws_secret_key');
3 $return = $sec2_instance->cancelBundle('bundleId');
```

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 Development 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.

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.

```
1 use ZendService\Apple\Apns\Client\Message as Client;
2 use ZendService\Apple\Apns\Message;
3 use ZendService\Apple\Apns\Message\Alert;
4 use ZendService\Apple\Apns\Response\Message as Response;
5 use ZendService\Apple\Exception\RuntimeException;
6
7 $client = new Client();
8 $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.

```
1 $message = new Message();
2 $message->setId('my_unique_id');
3 $message->setToken('DEVICE_TOKEN');
4 $message->setBadge(5);
5 $message->setSound('bingbong.aiff');
6
7 // simple alert:
8 $message->setAlert('Bob wants to play poker');
9 // complex alert:
10 $alert = new Alert();
11 $alert->setBody('Bob wants to play poker');
12 $alert->setActionLocKey('PLAY');
13 $alert->setLocKey('GAME_PLAY_REQUEST_FORMAT');
14 $alert->setLocArgs(array('Jenna', 'Frank'));
15 $alert->setLaunchImage('Play.png');
16 $message->setAlert($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.

```
1 try {
2     $response = $client->send($message);
3 } catch (RuntimeException $e) {
4     echo $e->getMessage() . PHP_EOL;
5     exit(1);
6 }
7 $client->close();
8
9 if ($response->getCode() != Response::RESULT_OK) {
10     switch ($response->getCode()) {
11         case Response::RESULT_PROCESSING_ERROR:
12             // you may want to retry
13             break;
14         case Response::RESULT_MISSING_TOKEN:
15             // you were missing a token
16             break;
17         case Response::RESULT_MISSING_TOPIC:
18             // you are missing a message id
19             break;
20         case Response::RESULT_MISSING_PAYLOAD:
21             // you need to send a payload
22             break;
23         case Response::RESULT_INVALID_TOKEN_SIZE:
```

```

24         // the token provided was not of the proper size
25         break;
26     case Response::RESULT_INVALID_TOPIC_SIZE:
27         // the topic was too long
28         break;
29     case Response::RESULT_INVALID_PAYLOAD_SIZE:
30         // the payload was too large
31         break;
32     case Response::RESULT_INVALID_TOKEN:
33         // the token was invalid; remove it from your system
34         break;
35     case Response::RESULT_UNKNOWN_ERROR:
36         // apple didn't tell us what happened
37         break;
38     }
39 }

```

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.

```

1  use ZendService\Apple\Apns\Client\Feedback as Client;
2  use ZendService\Apple\Apns\Response\Feedback as Response;
3  use ZendService\Apple\Exception\RuntimeException;
4
5  $client = new Client();
6  $client->open(Client::SANDBOX_URI, '/path/to/push-certificate.pem',
7  ↪ 'optionalPassPhrase');
8  $responses = $client->feedback();
9  $client->close();
10
11 foreach ($responses as $response) {
12     echo $response->getTime() . ': ' . $response->getToken();
13 }

```


Introduction

ZendService\Audioscrobbler\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\Audioscrobbler class begin with one of these terms. The syntax and namespaces of the Audioscrobbler Web Service are mirrored in ZendService\Audioscrobbler\Audioscrobbler. For more information about the Audioscrobbler REST Web Service, please visit the [Audioscrobbler Web Service site](#).

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\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:

```
1 $as = new ZendService\Audioscrobbler\Audioscrobbler();
2 // Set the user whose profile information we want to retrieve
3 $as->setUser('BigDaddy71');
4 // Retrieve BigDaddy71's profile information
5 $profileInfo = $as->userGetProfileInformation();
6 // Display some of it
7 print "Information for $profileInfo->realname "
8     . "can be found at $profileInfo->url";
```


Retrieving a User's Weekly Artist Chart

```

1  $as = new ZendService\Audioscrobbler\Audioscrobbler();
2  // Set the user whose profile weekly artist chart we want to retrieve
3  $as->setUser('lo_fye');
4  // Retrieves a list of previous weeks for which there are chart data
5  $weeks = $as->userGetWeeklyChartList();
6  if (count($weeks) < 1) {
7      echo 'No data available';
8  }
9  sort($weeks); // Order the list of weeks
10
11 $as->setFromDate($weeks[0]); // Set the starting date
12 $as->setToDate($weeks[0]); // Set the ending date
13
14 $previousWeeklyArtists = $as->userGetPreviousWeeklyArtistChart();
15
16 echo 'Artist Chart For Week Of '
17     . date('Y-m-d h:i:s', $as->from_date)
18     . '<br />';
19
20 foreach ($previousWeeklyArtists as $artist) {
21     // Display the artists' names with links to their profiles
22     print '<a href="' . $artist->url . '"' . $artist->name . '</a><br />';
23 }

```

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

```

1  $as = new ZendService\Audioscrobbler\Audioscrobbler();
2  // Set the artist for whom you would like to retrieve related artists
3  $as->setArtist('LCD Soundsystem');
4  // Retrieve the related artists
5  $relatedArtists = $as->artistGetRelatedArtists();
6  foreach ($relatedArtists as $artist) {
7      // Display the related artists

```

```
8     print '<a href="' . $artist->url . '"' . $artist->name . '</a><br />';  
9 }
```

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.

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.

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 from-Date to the current toDate for the current Group.

- `groupGetPreviousWeeklyAlbumChart($fromDate, $toDate)`: Requires `setFromDate()` and `setToDate()`. Returns a SimpleXML object containing the Weekly Album Chart from the current from-Date to the current toDate for the current Group.
- `groupGetPreviousWeeklyTrackChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Track Chart from the current fromDate to the current toDate for the current Group.

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.

Introduction

ZendService\Delicious\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

```
1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2 $posts = $delicious->getAllPosts();
3
4 foreach ($posts as $post) {
5     echo "--\n";
6     echo "Title: {$post->getTitle() }\n";
7     echo "Url: {$post->getUrl() }\n";
8 }
```

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.

```
1 /**
2  * Get posts matching the arguments. If no date or url is given,
3  * most recent date will be used.
4  *
5  * @param string $tag Optional filtering by tag
6  * @param DateTime $dt Optional filtering by date
```

```
7  * @param string $url Optional filtering by url
8  * @return ZendService\Delicious\PostList
9  */
10 public function getPosts($tag = null, $dt = null, $url = null);
11
12 /**
13  * Get recent posts
14  *
15  * @param string $tag Optional filtering by tag
16  * @param string $count Maximal number of posts to be returned
17  *                      (default 15)
18  * @return ZendService\Delicious\PostList
19  */
20 public function getRecentPosts($tag = null, $count = 15);
21
22 /**
23  * Get all posts
24  *
25  * @param string $tag Optional filtering by tag
26  * @return ZendService\Delicious\PostList
27  */
28 public function getAllPosts($tag = null);
```

ZendService\Delicious\PostList

Instances of this class are returned by the `getPosts()`, `getAllPosts()`, `getRecentPosts()`, and `getUserPosts()` methods of `ZendService\Delicious\Delicious`.

For easier data access this class implements the *Countable*, *Iterator*, and *ArrayAccess* interfaces.

Accessing post lists

```
1  $delicious = new ZendService\Delicious\Delicious('username', 'password');
2  $posts = $delicious->getAllPosts();
3
4  // count posts
5  echo count($posts);
6
7  // iterate over posts
8  foreach ($posts as $post) {
9      echo "--\n";
10     echo "Title: {$post->getTitle()}\n";
11     echo "Url: {$post->getUrl()}\n";
12 }
13
14 // get post using array access
15 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.

```

1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2 $posts = $delicious->getAllPosts();
3
4 // Print posts having "php" and "zend" tags
5 foreach ($posts->withTags(array('php', 'zend')) as $post) {
6     echo "Title: {$post->getTitle() }\n";
7     echo "Url: {$post->getUrl() }\n";
8 }

```

Filtering a Post List by URL

Posts may be filtered by *URL* matching a specified regular expression using the `withUrl()` method:

```

1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2 $posts = $delicious->getAllPosts();
3
4 // Print posts having "help" in the URL
5 foreach ($posts->withUrl('/help/') as $post) {
6     echo "Title: {$post->getTitle() }\n";
7     echo "Url: {$post->getUrl() }\n";
8 }

```

Editing posts

Post editing

```

1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2 $posts = $delicious->getPosts();
3
4 // set title
5 $posts[0]->setTitle('New title');
6 // save changes
7 $posts[0]->save();

```

Method call chaining

Every setter method returns the post object so that you can chain method calls using a fluent interface.

```

1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2 $posts = $delicious->getPosts();
3
4 $posts[0]->setTitle('New title')

```

```
5     ->setNotes('New notes')
6     ->save();
```

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

```
1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2
3 // by specifying URL
4 $delicious->deletePost('http://framework.zend.com');
5
6 // or by calling the method upon a post object
7 $posts = $delicious->getPosts();
8 $posts[0]->delete();
9
10 // another way of using deletePost()
11 $delicious->deletePost($posts[0]->getUrl());
```

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 `delicio.us` database by calling the `save()` method.

Adding a post

```
1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2
3 // create a new post and save it (with method call chaining)
4 $delicious->createNewPost('Zend Framework', 'http://framework.zend.com')
5     ->setNotes('Zend Framework Homepage')
6     ->save();
7
8 // create a new post and save it (without method call chaining)
9 $newPost = $delicious->createNewPost('Zend Framework',
10     'http://framework.zend.com');
11 $newPost->setNotes('Zend Framework Homepage');
12 $newPost->save();
```

Tags

Tags

```

1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2
3 // get all tags
4 print_r($delicious->getTags());
5
6 // rename tag ZF to zendFramework
7 $delicious->renameTag('ZF', 'zendFramework');
```

Bundles

Bundles

```

1 $delicious = new ZendService\Delicious\Delicious('username', 'password');
2
3 // get all bundles
4 print_r($delicious->getBundles());
5
6 // delete bundle someBundle
7 $delicious->deleteBundle('someBundle');
8
9 // add bundle
10 $delicious->addBundle('newBundle', array('tag1', 'tag2'));
```

Public data

The del.icio.us web *API* allows access to the public data of all users.

Table 348.1: Methods for retrieving public data

Name	Description	Return type
getUserFans()	Retrieves fans of a user	Array
getUserNetwork()	Retrieves network of a user	Array
getUserPosts()	Retrieves posts of a user	ZendServiceDeliciousPostList
getUserTags()	Retrieves tags of a user	Array

Note: When using only these methods, a username and password combination is not required when constructing a new `ZendService\Delicious\Delicious` object.

Retrieving public data

```

1 // username and password are not required
2 $delicious = new ZendService\Delicious\Delicious();
3
4 // get fans of user someUser
5 print_r($delicious->getUserFans('someUser'));
```

```
6 // get network of user someUser
7 print_r($delicious->getUserNetwork('someUser'));
8
9
10 // get tags of user someUser
11 print_r($delicious->getUserTags('someUser'));
```

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 348.2: Methods of the `ZendServiceDeliciousSimplePost` class

Name	Description	Return type
<code>getNotes()</code>	Returns notes of a post	String
<code>getTags()</code>	Returns tags of a post	Array
<code>getTitle()</code>	Returns title of a post	String
<code>getUrl()</code>	Returns URL of a post	String

HTTP client

`ZendService\Delicious\Delicious` uses `Zend\Rest\Client` for making *HTTP* requests to the *del.icio.us* web service. To change which *HTTP* client `ZendService\Delicious\Delicious` uses, you need to change the *HTTP* client of `Zend\Rest\Client`.

Changing the HTTP client of `ZendRestClient`

```
1 $myHttpClient = new My_Http_Client();
2 Zend\Rest\Client::setHttpClient($myHttpClient);
```

When you are making more than one request with `ZendService\Delicious\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

```
1 Zend\Rest\Client::getHttpClient()->setConfig(array(
2     'keepalive' => true
3 ));
```

Note: When a `ZendService\Delicious\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).

Introduction

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 *APIs*). You can access the Developer Garden services directly via *SOAP* or *REST*.

The family of `ZendService\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*: Locate 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](#).

Sign Up for an Account

Before you can start using the DeveloperGarden *API*, you first have to [sign up](#) for an account.

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.

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

```
1 require_once 'ZendService/DeveloperGarden/SendSms.php';
2 $config = array(
3     'username'    => 'yourUsername',
4     'password'    => 'yourPassword',
5     'environment' => ZendService\DeveloperGarden\SendSms::ENV_PRODUCTION,
6 );
7 $service = new ZendService\DeveloperGarden\SendSms($config);
```

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

```
1 $service = new ZendService\DeveloperGarden\BaseUserService($config);
2 print_r($service->getAccountBalance());
```

Get quota information

You can fetch quota informations for a specific service module with the provided methods.

Get quota information example

```

1 $service = new ZendService\DeveloperGarden\BaseUserService($config);
2 $result = $service->getSmsQuotaInformation(
3     ZendService\DeveloperGarden\BaseUserService::ENV_PRODUCTION
4 );
5 echo 'Sms Quota:<br />';
6 echo 'Max Quota: ', $result->getMaxQuota(), '<br />';
7 echo 'Max User Quota: ', $result->getMaxUserQuota(), '<br />';
8 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()

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

```

1 $service = new ZendService\DeveloperGarden\BaseUserService($config);
2 $result = $service->changeSmsQuotaPool(
3     1000,
4     ZendService\DeveloperGarden\BaseUserService::ENV_PRODUCTION
5 );
6 if (!$result->hasError()) {
7     echo 'updated Quota Pool';
8 }

```

Here a list of all changeQuotaPool methods:

- changeConferenceCallQuotaPool()
- changeIPLocationQuotaPool()
- changeLocalSearchQuotaPool()
- changeSmsQuotaPool()
- changeVoiceCallQuotaPool()

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

```
1 $service = new ZendService\DeveloperGarden\IpLocation($config);
2 $service->setEnvironment(
3     ZendService\DeveloperGarden\IpLocation::ENV MOCK
4 );
5 $ip = new ZendService\DeveloperGarden\IpLocation\IpAddress('127.0.0.1');
6 print_r($service->locateIp($ip));
```

Local Search

The Local Search service provides the location based search machine [suchen.de](http://www.suchen.de) via web service interface. For more details, refer to [the documentation](#).

Locate a Restaurant

```
1 $service = new ZendService\DeveloperGarden\LocalSearch($config);
2 $search = new ZendService\DeveloperGarden\LocalSearch\SearchParameters();
3 /**
4  * @see http://www.developergarden.com/static/docu/en/ch04s02s06s04.html
5  */
6 $search->setWhat('pizza')
7     ->setWhere('jena');
8 print_r($service->localSearch($search));
```

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

```

1 $service = new ZendService\DeveloperGarden\SendSms($config);
2 $sms = $service->createSms(
3     '+49-172-123456; +49-177-789012',
4     'your test message',
5     'yourname'
6 );
7 print_r($service->send($sms));

```

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

```

1 $service = new ZendService\DeveloperGarden\SmsValidation($config);
2 print_r($service->sendValidationKeyword('+49-172-123456'));

```

Validate a number with a keyword

```

1 $service = new ZendService\DeveloperGarden\SmsValidation($config);
2 print_r($service->validate('TheKeyWord', '+49-172-123456'));

```

To invalidate a validated number, call the method `invalidate()`.

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

```
1 $service = new ZendService\DeveloperGarden\VoiceCall($config);
2 $aNumber = '+49-30-000001';
3 $bNumber = '+49-30-000002';
4 $expiration = 30; // seconds
5 $maxDuration = 300; // 5 mins
6 $newCall = $service->newCall($aNumber, $bNumber, $expiration, $maxDuration);
7 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

```
1 $service = new ZendService\DeveloperGarden\VoiceCall($config);
2 $aNumber = '+49-30-000001';
3 $bNumber = '+49-30-000002';
4 $expiration = 30; // seconds
5 $maxDuration = 300; // 5 mins
6
7 $newCall = $service->newCall($aNumber, $bNumber, $expiration, $maxDuration);
8
9 $sessionId = $newCall->getSessionId();
10
11 $service->callStatus($sessionId, true); // extend the call
12
13 sleep(10); // sleep 10s and then tearDown
14
15 $service->tearDownCall($sessionId);
```

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

```

1  $client = new ZendService\DeveloperGarden\ConferenceCall($config);
2
3  $conferenceDetails =
4      new ZendService\DeveloperGarden\ConferenceCall\ConferenceDetail(
5          'Zend-Conference',           // name for the conference
6          'this is my private zend conference', // description
7          60                          // duration in seconds
8      );
9
10 $conference = $client->createConference('MyName', $conferenceDetails);
11
12 $part1 = new ZendService\DeveloperGarden\ConferenceCall\ParticipantDetail(
13     'Jon',
14     'Doe',
15     '+49-123-4321',
16     'your.name@example.com',
17     true
18 );

```

```
19
20 $client->newParticipant($conference->getConferenceId(), $part1);
21 // add a second, third ... participant
22
23 $client->commitConference($conference->getConferenceId());
```

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

```
1 ZendService\DeveloperGarden\SecurityTokenServer\Cache::setWsdCache (
2     [PHP CONSTANT]
3 );
```

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

```
1 $cache = Zend_Cache::factory('Core', ...);
2 ZendService\DeveloperGarden\SecurityTokenServer\Cache::setCache($cache);
```

Introduction

ZendService\Flickr\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

```
1 $flickr = new ZendService\Flickr\Flickr('MY_API_KEY');
2
3 $results = $flickr->tagSearch("php");
4
5 foreach ($results as $result) {
6     echo $result->title . '<br />';
7 }
```

Note: Optional parameter

`tagSearch()` accepts an optional second parameter as an array of options.

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:

```
1 $flickr = new ZendService\Flickr\Flickr('MY_API_KEY');
2
3 $results = $flickr->userSearch($userEmail);
4
5 foreach ($results as $result) {
6     echo $result->title . '<br />';
7 }
```

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

```
1 $flickr = new ZendService\Flickr\Flickr('MY_API_KEY');
2
3 $results = $flickr->groupPoolGetPhotos($groupId);
4
5 foreach ($results as $result) {
6     echo $result->title . '<br />';
7 }
```

Note: Optional parameter

`groupPoolGetPhotos()` accepts an optional second parameter as an array of options.

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:

```

1 $flickr = new ZendService\Flickr\Flickr('MY_API_KEY');
2
3 $image = $flickr->getImageDetails($imageId);
4
5 echo "Image ID $imageId is $image->width x $image->height pixels.<br />\n";
6 echo "<a href=\"\$image->clickUri\">Click for Image</a>\n";

```

ZendService\Flickr Result Classes

The following classes are all returned by `tagSearch()` and `userSearch()`:

- *ZendServiceFlickrResultSet*
- *ZendServiceFlickrResult*
- *ZendServiceFlickrImage*

ZendService\Flickr\ResultSet

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()`.

Properties

Table 350.1: ZendServiceFlickrResultSet Properties

Name	Type	Description
<code>totalResultsAvailable</code>	int	Total Number of Results available
<code>totalResultsReturned</code>	int	Total Number of Results returned
<code>firstResultPosition</code>	int	The offset in the total result set of this result set

ZendService\Flickr\ResultSet::totalResults()

`int:totalResults()`

Returns the total number of results in this result set.

[Back to Class List](#)

ZendService\Flickr\Result

A single Image result from a Flickr query

Properties

Table 350.2: ZendServiceFlickrResult Properties

Name	Type	Description
id	string	Image ID
owner	string	The photo owner's NSID.
secret	string	A key used in url construction.
server	string	The servername to use for URL construction.
title	string	The photo's title.
ispublic	string	The photo is public.
isfriend	string	The photo is visible to you because you are a friend of the owner.
isfamily	string	The photo is visible to you because you are family of the owner.
license	string	The license the photo is available under.
dateupload	string	The date the photo was uploaded.
datetaken	string	The date the photo was taken.
ownername	string	The screenname of the owner.
iconserver	string	The server used in assembling icon URLs.
Square	ZendServiceFlickrImage	A 75x75 thumbnail of the image.
Thumbnail	ZendServiceFlickrImage	A 100 pixel thumbnail of the image.
Small	ZendServiceFlickrImage	A 240 pixel version of the image.
Medium	ZendServiceFlickrImage	A 500 pixel version of the image.
Large	ZendServiceFlickrImage	A 640 pixel version of the image.
Original	ZendServiceFlickrImage	The original image.

[Back to Class List](#)

ZendService\Flickr\Image

Represents an Image returned by a Flickr search.

Properties

Table 350.3: 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](#)

ZendService\Google\Gcm

Introduction

ZendService\Google\Gcm provides a client for the Google Cloud Messaging (GCM) API. ZendService\Google\Gcm\Client allows you to send data from servers to your Android Applications on Android devices (Google API driven).

In order to leverage GCM you **must** create your project in the Google API Console and enable the GCM service on your device. To get started with GCM prior to building out the 3rd-party server please see [GCM: Getting Started](#)

The service is composed of 3 distinct parts:

- The Client: ZendService\Google\Gcm\Client
- The Message: ZendService\Google\Gcm\Message
- The Response: ZendService\Google\Gcm\Response

The Client is the broker that sends the message to the GCM server and returns the response. The Message is where you define all of the message specific data that you would like to send. The response is the feedback given back from the GCM server on success, failures and any new canonical id's that must be updated.

Quick Start

In order to send messages; you must have your API key ready and available. Here we will setup the client and prepare ourselves to send out messages.

```
1 use ZendService\Google\Gcm\Client;
2 use ZendService\Google\Gcm\Message;
3 use ZendService\Google\Exception\RuntimeException;
4
5 $client = new Client();
6 $client->setApiKey('the-api-key-for-gcm');
```

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 registration id's 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.

```
1 $message = new Message();
2
3 // up to 100 registration ids can be sent to at once
4 $message->setRegistrationIds(array(
5     '1-an-id-from-gcm',
6     '2-an-id-from-gcm',
7 ));
8
9 // optional fields
10 $message->setData(array(
11     'pull-request' => '1',
12 ));
13 $message->setCollapseKey('pull-request');
14 $message->setRestrictedPackageName('com.zf.manual');
15 $message->setDelayWhileIdle(false);
16 $message->setTimeToLive(600);
17 $message->setDryRun(false);
```

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.

```
1 try {
2     $response = $client->send($message);
3 } catch (RuntimeException $e) {
4     echo $e->getMessage() . PHP_EOL;
5     exit(1);
6 }
7 echo 'Successful: ' . $response->getSuccessCount() . PHP_EOL;
8 echo 'Failures: ' . $response->getFailureCount() . PHP_EOL;
9 echo 'Canonicals: ' . $response->getCanonicalCount() . PHP_EOL;
```


Introduction

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\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\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\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`.
- [LiveDocx in PHP](#).
- [LiveDocx SOAP API documentation](#).
- [LiveDocx WSDL](#).
- [LiveDocx blog and web site](#).

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](#).

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.

Supported File Formats

LiveDocx supports the following file formats:

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

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

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

ZendService\LiveDocx\MailMerge

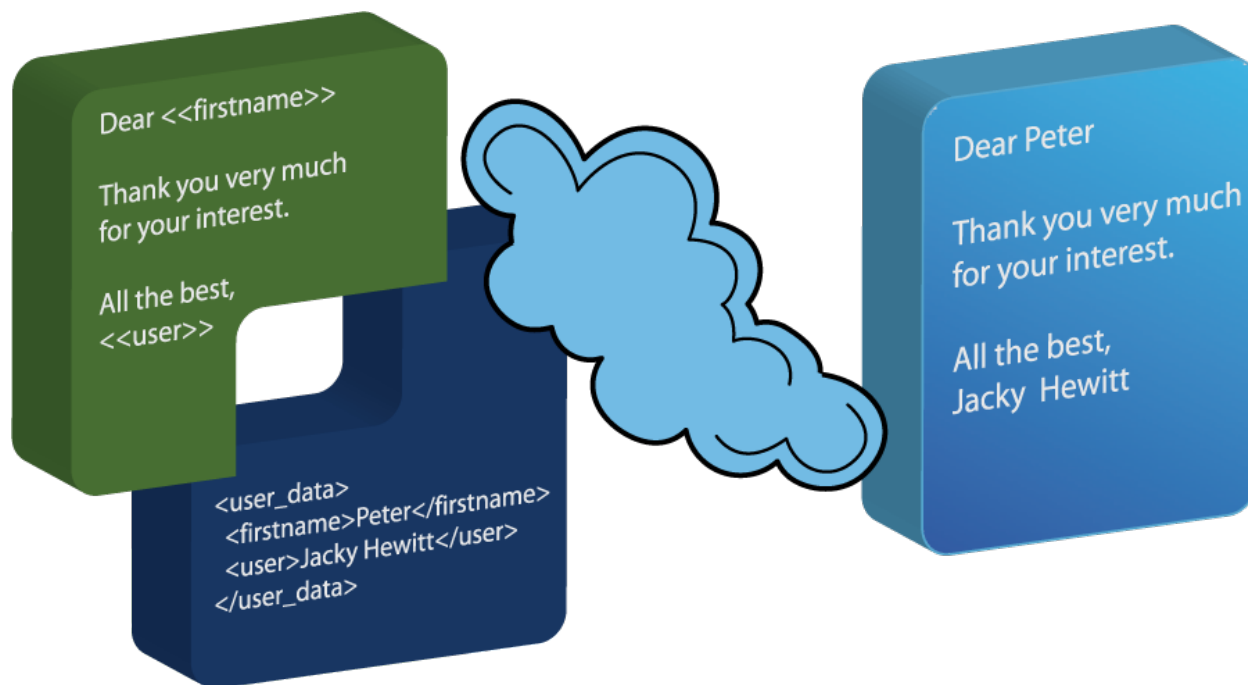
MailMerge is the mail-merge object in the ZendService\LiveDocx\LiveDocx family.

Document Generation Process

The document generation process can be simplified with the following equation:

Template + Data = Document

Or expressed by the following diagram:

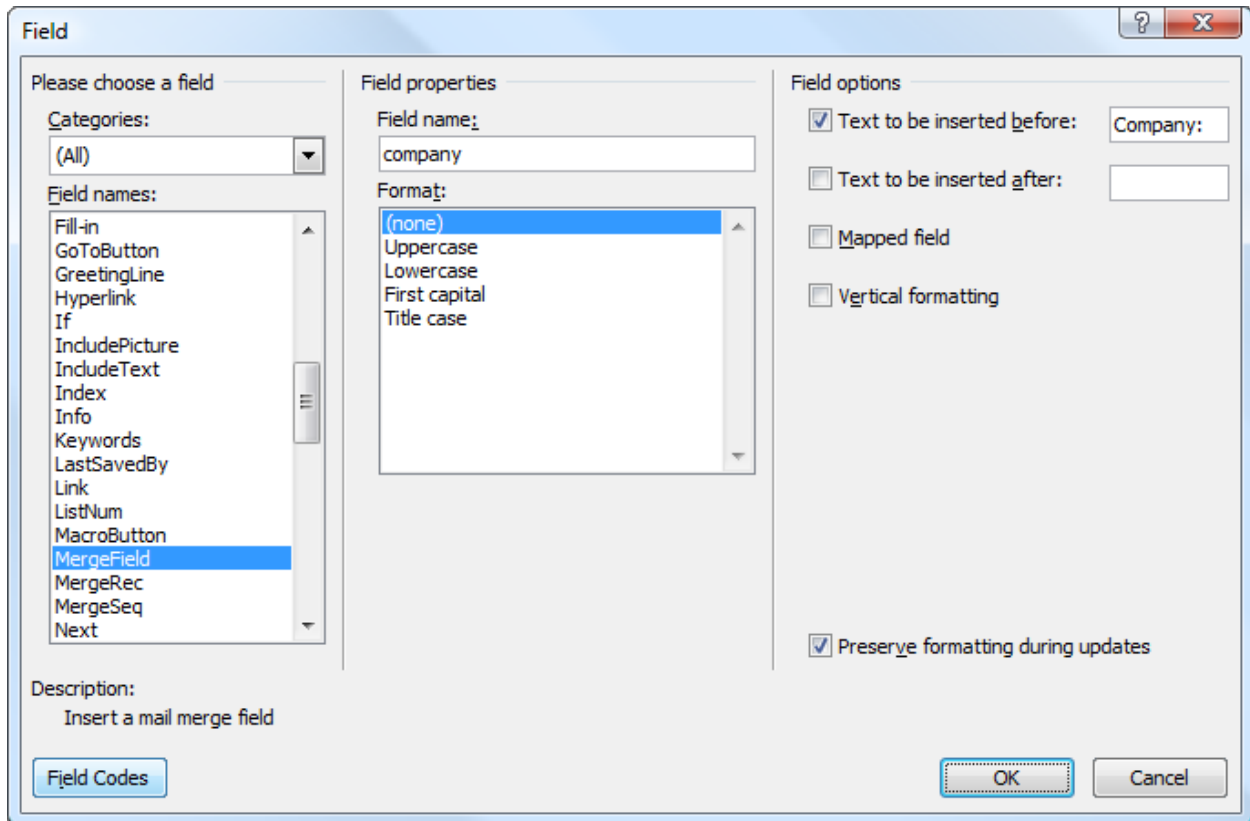


Data is inserted into template to create a document.

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.

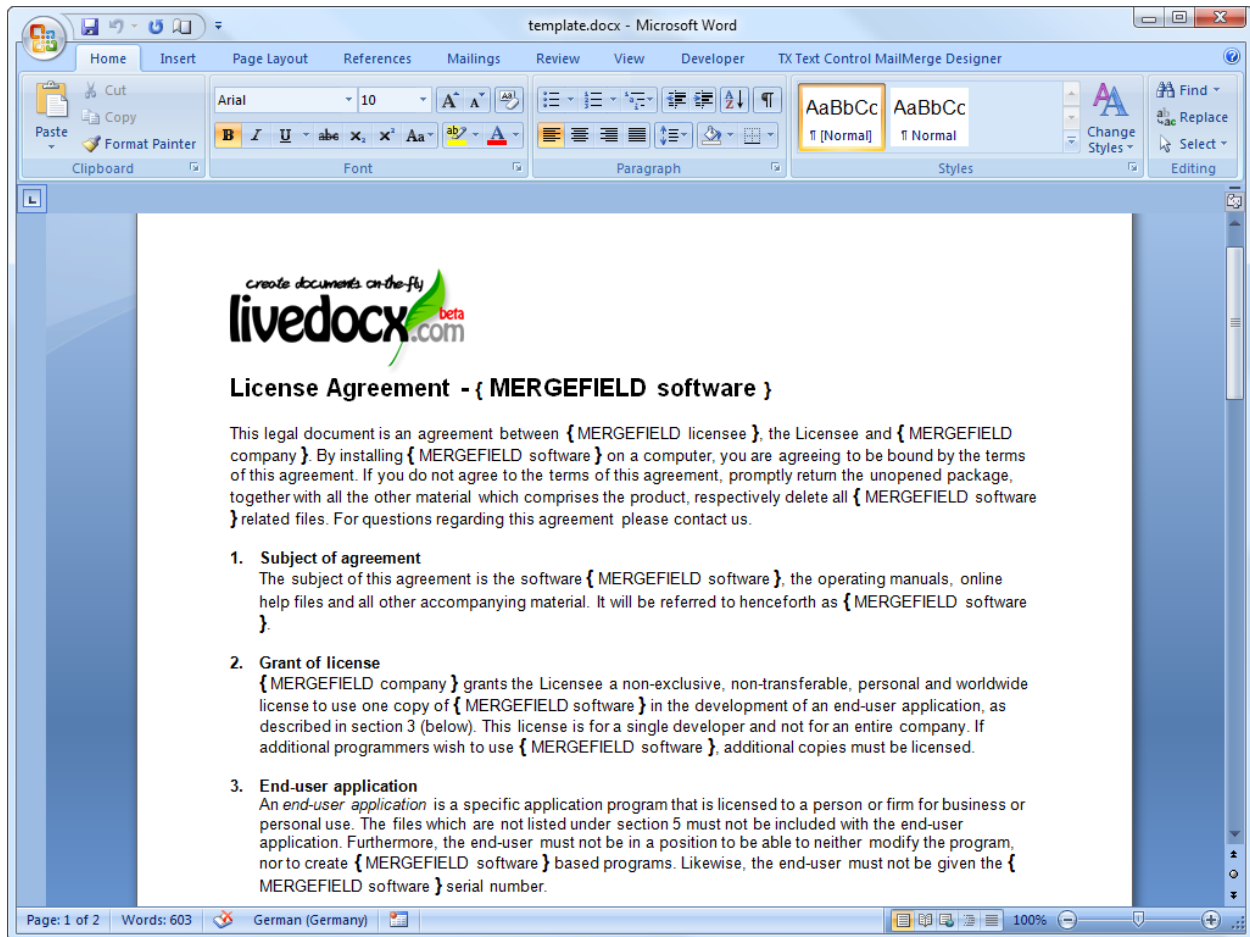
Creating Templates in Microsoft Word 2007

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*.

License Agreement - { MERGEFIELD software }

This legal document is an agreement between { MERGEFIELD licensee }, the Licensee and { MERGEFIELD company }. By installing { MERGEFIELD software } on a computer, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened package, together with all the other material which comprises the product, respectively delete all { MERGEFIELD software } related files. For questions regarding this agreement please contact us.

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:

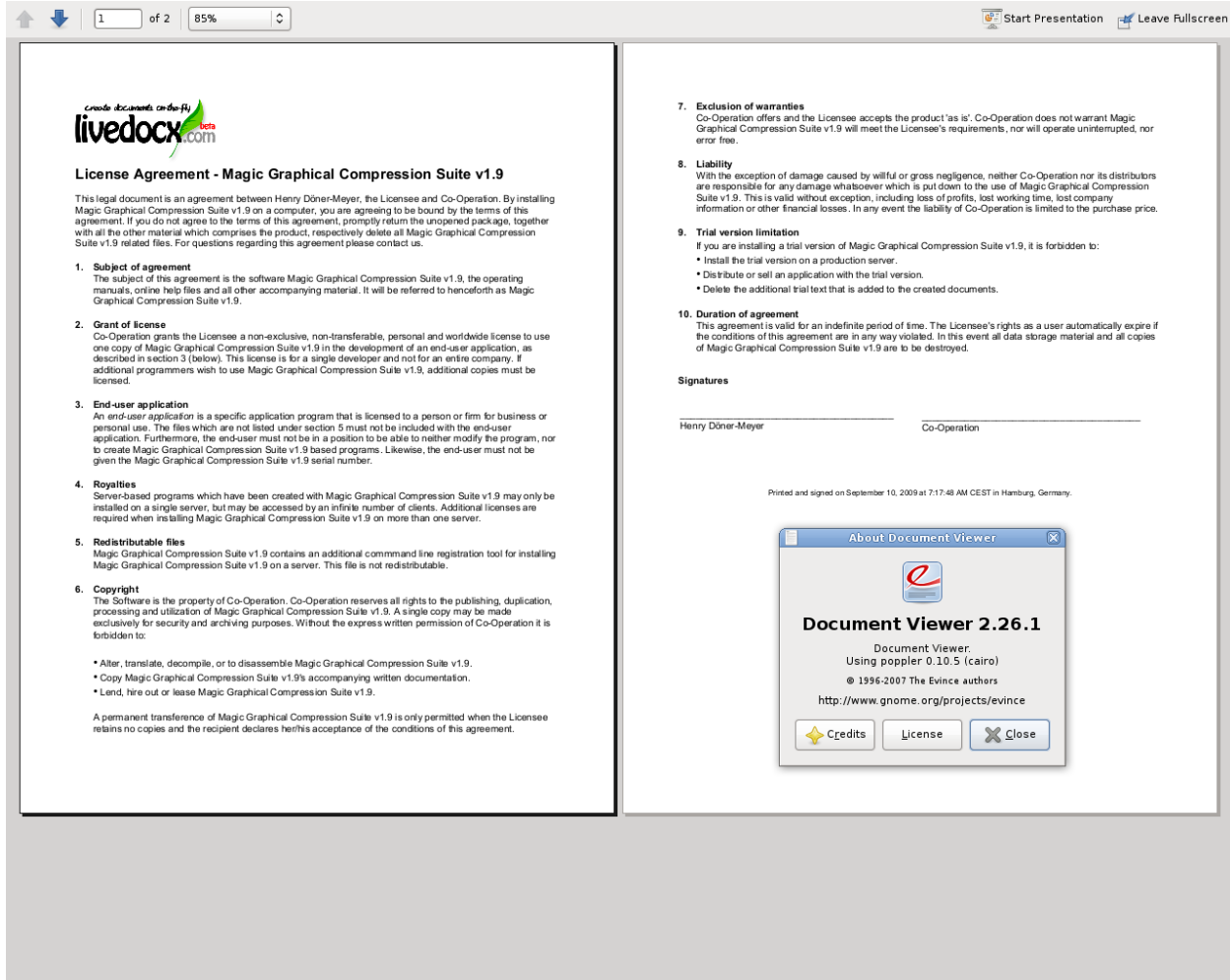
```

1  use ZendService\LiveDocx\MailMerge;
2
3  $locale      = Locale::getDefault();
4  $timestamp   = time();
5
6  $intlTimeFormatter = new IntlDateFormatter($locale,
7      IntlDateFormatter::NONE, IntlDateFormatter::SHORT);

```

```
8
9  $intlDateFormatter = new IntlDateFormatter($locale,
10      IntlDateFormatter::LONG, IntlDateFormatter::NONE);
11
12  $mailMerge = new MailMerge();
13
14  $mailMerge->setUsername('myUsername')
15      ->setPassword('myPassword')
16      ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↪MailMerge::SERVICE_PREMIUM
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`.

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:

<div>blockStart_block1</div> <div>Connection: { MERGEFIELD connection number } { MERGEFIELD connection duration }</div>	{ MERGEFIELD fee }
<div>Total net</div> <div>blockEnd_block1</div>	{ MERGEFIELD total_net }

The format of a block is as follows:

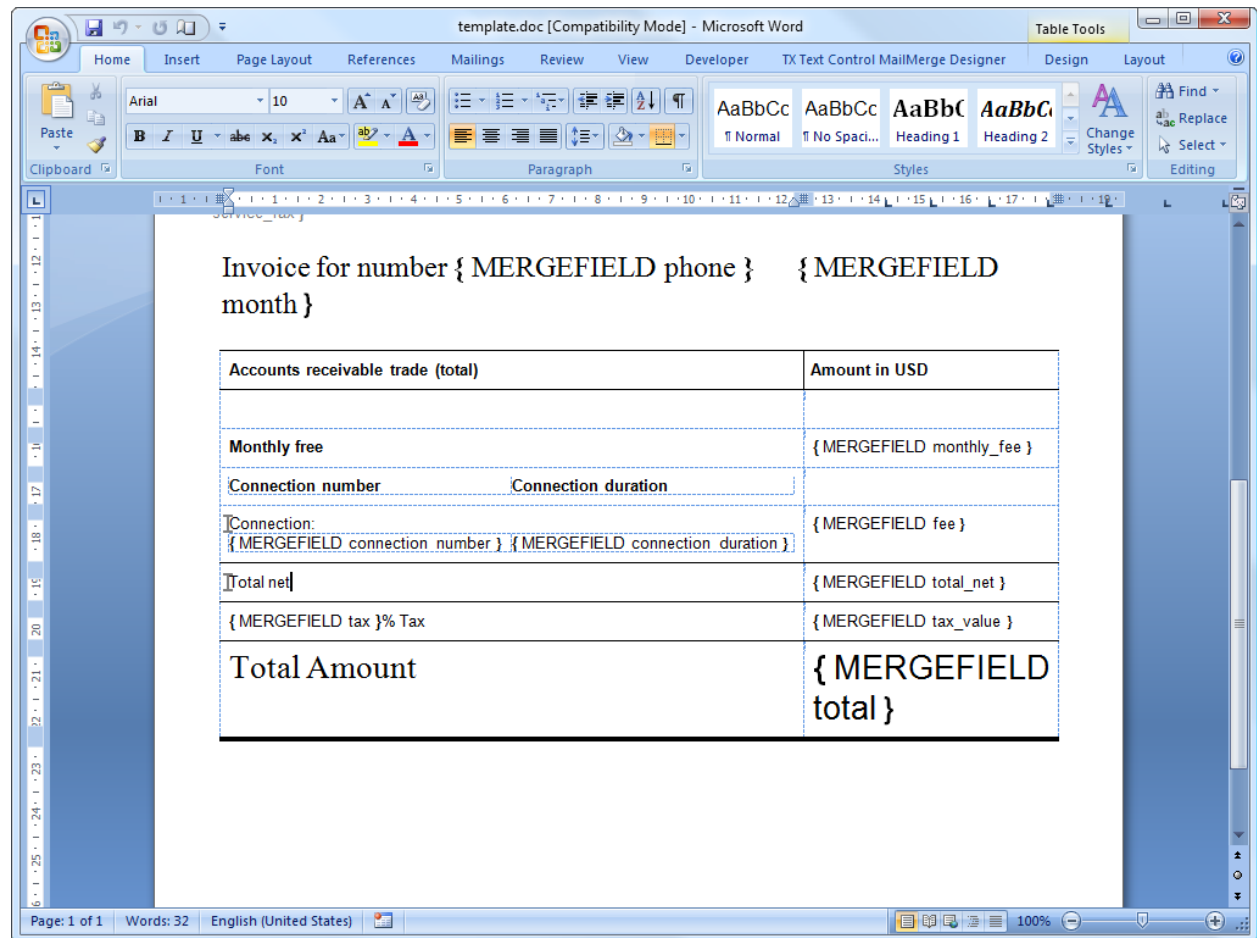
```
blockStart_ + unique name
blockEnd_ + unique name
```

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.


```

1  use ZendService\LiveDocx\MailMerge;
2
3  $locale      = Locale::getDefault();
4  $timestamp   = time();
5
6  $intlDateFormatter1 = new IntlDateFormatter($locale,
7      IntlDateFormatter::LONG, IntlDateFormatter::NONE);
8
9  $intlDateFormatter2 = new IntlDateFormatter($locale,
10     null, null, null, null, 'LLLL yyyy');
11
12  $mailMerge = new MailMerge();
13
14  $mailMerge->setUsername('myUsername')
15      ->setPassword('myPassword')
16      ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↪MailMerge::SERVICE_PREMIUM
17
18  $mailMerge->setLocalTemplate('telephone-bill-template.doc');
19
20  $mailMerge->assign('customer_number', sprintf("#%10s", rand(0,1000000000)))
21      ->assign('invoice_number',    sprintf("#%10s", rand(0,1000000000)))
22      ->assign('account_number',    sprintf("#%10s", rand(0,1000000000)));
23
24  $billData = array (
25      'phone'      => '+22 (0)333 444 555',
26      'date'       => $intlDateFormatter1->format($timestamp),
27      'name'       => 'James Henry Brown',
28      'service_phone' => '+22 (0)333 444 559',
29      'service_fax' => '+22 (0)333 444 558',
30      'month'      => $intlDateFormatter2->format($timestamp),
31      'monthly_fee' => '15.00',
32      'total_net'  => '19.60',
33      'tax'        => '19.00',
34      'tax_value'  => '3.72',
35      'total'      => '23.32'
36  );
37
38  $mailMerge->assign($billData);
39
40  $billConnections = array(
41      array(
42          'connection_number' => '+11 (0)222 333 441',
43          'connection_duration' => '00:01:01',
44          'fee' => '1.15'
45      ),
46      array(
47          'connection_number' => '+11 (0)222 333 442',
48          'connection_duration' => '00:01:02',
49          'fee' => '1.15'
50      ),
51      array(
52          'connection_number' => '+11 (0)222 333 443',
53          'connection_duration' => '00:01:03',
54          'fee' => '1.15'
55      ),
56      array(
57          'connection_number' => '+11 (0)222 333 444',

```

```

58     'connection_duration' => '00:01:04',
59     'fee'                  => '1.15'
60 )
61 );
62
63 $mailMerge->assign('connection', $billConnections);
64
65 $mailMerge->createDocument();
66
67 $document = $mailMerge->retrieveDocument('pdf');
68
69 file_put_contents('telephone-bill-document.pdf', $document);
70
71 unset($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**:

The screenshot shows a web-based document viewer displaying a PDF invoice. The invoice is from 'Telecom Company' and is dated September 10, 2009. It includes contact information for James Henry Brown and details about the account and services. A table of charges is visible, showing a total amount of € 130.00.

Accounts receivable trade (total)	Amount in USD
Monthly free	€ 15.00
Connection number	Connection duration
Connection: +49 421 335 912	00:00:07
Connection: +49 421 335 913	00:00:07
Connection: +49 421 335 914	00:00:07
Connection: +49 421 335 916	00:00:07
Total net	€ 100.00
19% Tax	€ 15.00
Total Amount	€ 130.00

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`.

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:` - it must start with `image:`:

```

1  use ZendService\LiveDocx\MailMerge;
2
3  $locale      = Locale::getDefault();
4  $timestamp   = time();
5
6  $intlDateFormatter = new IntlDateFormatter($locale,
7      IntlDateFormatter::LONG, IntlDateFormatter::NONE);
8
9  $mailMerge = new MailMerge();
10
11  $mailMerge->setUsername('myUsername')
12      ->setPassword('myPassword')
13      ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use
↳ MailMerge::SERVICE_PREMIUM
14
15  $photoFilename = __DIR__ . '/dailemaitre.jpg';
16  $photoFile     = basename($photoFilename);
17
18  if (!$mailMerge->imageExists($photoFile)) {           // pass image file *without* path
19      $mailMerge->uploadImage($photoFilename);          // pass image file *with* path
20  }
21
22  $mailMerge->setLocalTemplate('conference-pass-template.docx');
23
24  $mailMerge->assign('name',      'Daï Lemaitre')
25      ->assign('company',      'Megasoft Co-operation')
26      ->assign('date',         $intlDateFormatter->format($timestamp))
27      ->assign('image:photo', $photoFile);             // pass image file *without* path
28
29  $mailMerge->createDocument();
30
31  $document = $mailMerge->retrieveDocument('pdf');
32
33  file_put_contents('conference-pass-document.pdf', $document);
34
35  $mailMerge->deleteImage($photoFilename);
36
37  unset($mailMerge);

```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/conference-pass`.

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()`.

```

1  use ZendService\LiveDocx\MailMerge;
2
3  $locale      = Locale::getDefault();
4  $timestamp   = time();
5
6  $intlTimeFormatter = new IntlDateFormatter($locale,
7      IntlDateFormatter::NONE, IntlDateFormatter::SHORT);
8
9  $intlDateFormatter = new IntlDateFormatter($locale,
10     IntlDateFormatter::LONG, IntlDateFormatter::NONE);
11
12  $mailMerge = new MailMerge();
13
14  $mailMerge->setUsername('myUsername')
15      ->setPassword('myPassword')
16      ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↪MailMerge::SERVICE_PREMIUM
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  // Get all bitmaps
31  // (zoomFactor, format)
32  $bitmaps = $mailMerge->getAllBitmaps(100, 'png');
33
34  // Get just bitmaps in specified range
35  // (fromPage, toPage, zoomFactor, format)
36  //$bitmaps = $mailMerge->getBitmaps(2, 2, 100, 'png');
37
38  foreach ($bitmaps as $pageNumber => $bitmapData) {
39      $filename = sprintf('license-agreement-page-%d.png', $pageNumber);
40      file_put_contents($filename, $bitmapData);
41  }

```

```
42  
43  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 - Magic Graphical Compression Suite v1.9

This legal document is an agreement between Dai Lemaitre, the Licensee and Megasoft Co-operation. By installing Magic Graphical Compression Suite v1.9 on a computer, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened package, together with all the other material which comprises the product, respectively delete all Magic Graphical Compression Suite v1.9 related files. For questions regarding this agreement please contact us.

1. Subject of agreement

The subject of this agreement is the software Magic Graphical Compression Suite v1.9, the operating manuals, online help files and all other accompanying material. It will be referred to henceforth as Magic Graphical Compression Suite v1.9.

2. Grant of license

Megasoft Co-operation grants the Licensee a non-exclusive, non-transferable, personal and worldwide license to use one copy of Magic Graphical Compression Suite v1.9 in the development of an end-user application, as described in section 3 (below). This license is for a single developer and not for an entire company. If additional programmers wish to use Magic Graphical Compression Suite v1.9, additional copies must be licensed.

3. End-user application

An *end-user application* is a specific application program that is licensed to a person or firm for business or personal use. The files which are not listed under section 5 must not be included with the end-user application. Furthermore, the end-user must not be in a position to be able to neither modify the program, nor to create Magic Graphical Compression Suite v1.9 based programs. Likewise, the end-user must not be given the Magic Graphical Compression Suite v1.9 serial number.

4. Royalties

Server-based programs which have been created with Magic Graphical Compression Suite v1.9 may only be installed on a single server, but may be accessed by an infinite number of clients. Additional licenses are required when installing Magic Graphical Compression Suite v1.9 on more than one server.

5. Redistributable files

Magic Graphical Compression Suite v1.9 contains an additional command line registration tool for installing Magic Graphical Compression Suite v1.9 on a server. This file is not redistributable.

6. Copyright

The Software is the property of Megasoft Co-operation. Megasoft Co-operation reserves all rights to the publishing, duplication, processing and utilization of Magic Graphical Compression Suite v1.9. A single copy may be made exclusively for security and archiving purposes. Without the express written permission of Megasoft Co-operation it is forbidden to:

- Alter, translate, decompile, or to disassemble Magic Graphical Compression Suite v1.9.
- Copy Magic Graphical Compression Suite v1.9's accompanying written documentation.
- Lend, hire out or lease Magic Graphical Compression Suite v1.9.

A permanent transference of Magic Graphical Compression Suite v1.9 is only permitted when the Licensee retains no copies and the recipient declares her/his acceptance of the conditions of this agreement.

license-agreement-page-1.png.

7. Exclusion of warranties

Megasoft Co-operation offers and the Licensee accepts the product 'as is'. Megasoft Co-operation does not warrant Magic Graphical Compression Suite v1.9 will meet the Licensee's requirements, nor will operate uninterrupted, nor error free.

8. Liability

With the exception of damage caused by willful or gross negligence, neither Megasoft Co-operation nor its distributors are responsible for any damage whatsoever which is put down to the use of Magic Graphical Compression Suite v1.9. This is valid without exception, including loss of profits, lost working time, lost company information or other financial losses. In any event the liability of Megasoft Co-operation is limited to the purchase price.

9. Trial version limitation

If you are installing a trial version of Magic Graphical Compression Suite v1.9, it is forbidden to:

- Install the trial version on a production server.
- Distribute or sell an application with the trial version.
- Delete the additional trial text that is added to the created documents.

10. Duration of agreement

This agreement is valid for an indefinite period of time. The Licensee's rights as a user automatically expire if the conditions of this agreement are in any way violated. In this event all data storage material and all copies of Magic Graphical Compression Suite v1.9 are to be destroyed.

Signatures

Dai Lemaitre

Megasoft Co-operation

Printed and signed on December 2, 2009 at 6:34:57 AM CET in Lyon, France.

license-agreement-page-2.png. For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/bitmaps`.

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.

```
1  use ZendService\LiveDocx\MailMerge;
2
3  $mailMerge = new MailMerge();
4
5  $mailMerge->setUsername('myUsername')
6             ->setPassword('myPassword')
7             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↳ MailMerge::SERVICE_PREMIUM
8
9  $mailMerge->setLocalTemplate('template.docx');
10
11 // assign data and create document
12
13 unset($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:

```
1  use ZendService\LiveDocx\MailMerge;
2
3  $mailMerge = new MailMerge();
4
5  $mailMerge->setUsername('myUsername')
6             ->setPassword('myPassword')
7             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↳ MailMerge::SERVICE_PREMIUM
8
9  $mailMerge->uploadTemplate('template.docx');
10
11 unset($mailMerge);
```

The following code illustrates how to reference the remotely stored template on all subsequent requests:

```
1  use ZendService\LiveDocx\MailMerge;
2
3  $mailMerge = new MailMerge();
4
5  $mailMerge->setUsername('myUsername')
6             ->setPassword('myPassword')
7             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↳ MailMerge::SERVICE_PREMIUM
8
9  $mailMerge->setRemoteTemplate('template.docx');
```



```

11 // assign data and create document
12
13 unset($mailMerge);

```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/templates`.

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.

```

1 use ZendService\LiveDocx\MailMerge;
2
3 $mailMerge = new MailMerge();
4
5 $mailMerge->setUsername('myUsername')
6             ->setPassword('myPassword')
7             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use
↳MailMerge::SERVICE_PREMIUM
8
9 $templateName = 'template-1-text-field.docx';
10 $mailMerge->setLocalTemplate($templateName);
11
12 $fieldNames = $mailMerge->getFieldNames();
13 foreach ($fieldNames as $fieldName) {
14     printf('- %s%s', $fieldName, PHP_EOL);
15 }
16
17 unset($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.

```

1 use ZendService\LiveDocx\MailMerge;
2
3 $mailMerge = new MailMerge();
4
5 $mailMerge->setUsername('myUsername')
6             ->setPassword('myPassword')
7             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use
↳MailMerge::SERVICE_PREMIUM

```

```
8
9 $templateName = 'template-block-fields.doc';
10 $mailMerge->setLocalTemplate($templateName);
11
12 $blockNames = $mailMerge->getBlockNames();
13 foreach ($blockNames as $blockName) {
14     $blockFieldNames = $mailMerge->getBlockFieldNames($blockName);
15     foreach ($blockFieldNames as $blockFieldName) {
16         printf('- %s::%s%s', $blockName, $blockFieldName, PHP_EOL);
17     }
18 }
19
20 unset($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.

```
1 use ZendService\LiveDocx\MailMerge;
2 use Zend\Debug\Debug;
3
4 $mailMerge = new MailMerge();
5
6 $mailMerge->setUsername('myUsername')
7             ->setPassword('myPassword')
8             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use
↳ MailMerge::SERVICE_PREMIUM
9
10 Debug::dump($mailMerge->getFontNames());
11
12 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.

```
1 use ZendService\LiveDocx\MailMerge;
2 use Zend\Debug\Debug;
```

```

3
4 $mailMerge = new MailMerge()
5
6 $mailMerge->setUsername('myUsername')
7             ->setPassword('myPassword')
8             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↳ MailMerge::SERVICE_PREMIUM
9
10 Debug::dump($mailMerge->getTemplateFormats());
11
12 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.

```

1 use ZendService\LiveDocx\MailMerge;
2 use Zend\Debug\Debug;
3
4 $mailMerge = new MailMerge();
5
6 $mailMerge->setUsername('myUsername')
7             ->setPassword('myPassword')
8             ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use_
↳ MailMerge::SERVICE_PREMIUM
9
10 Debug::dump($mailMerge->getDocumentFormats());
11
12 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.

```

1 use ZendService\LiveDocx\MailMerge;
2 use Zend\Debug\Debug;
3
4 $mailMerge = new MailMerge();
5
6 $mailMerge->setUsername('myUsername')

```

```
7         ->setPassword('myPassword')
8         ->setService (MailMerge::SERVICE_FREE); // for LiveDocx Premium, use
↪MailMerge::SERVICE_PREMIUM
9
10     Debug::dump($mailMerge->getImageExportFormats());
11
12     unset($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`.

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:

```
1     use ZendService\LiveDocx\MailMerge;
2
3     $mailMerge = new MailMerge()
4
5     $mailMerge->setUsername('myUsername')
6                 ->setPassword('myPassword')
7                 ->setService (MailMerge::SERVICE_FREE);
8
9     // rest of your application here
10
11     unset($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:

```
1     use ZendService\LiveDocx\MailMerge;
2
3     $mailMerge = new MailMerge()
4
5     $mailMerge->setUsername('myPremiumUsername')
6                 ->setPassword('myPremiumPassword')
7                 ->setService (MailMerge::SERVICE_PREMIUM);
8
9     // rest of your application here
10
11     unset($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`.

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:

```

1  use ZendService\LiveDocx\MailMerge;
2
3  $mailMerge = new MailMerge()
4
5  $mailMerge->setUsername('myUsername')
6              ->setPassword('myPassword')
7              ->setService (MailMerge::SERVICE_FREE);
8              // or
9              // ->setService (MailMerge::SERVICE_PREMIUM);
10
11 // rest of your application here
12
13 unset($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:

```

1  use ZendService\LiveDocx\MailMerge;
2
3  $mailMerge = new MailMerge()
4
5  $mailMerge->setUsername('myFullyLicensedUsername')
6              ->setPassword('myFullyLicensedPassword')
7              ->setWsd1    ('http://api.example.com/2.1/mailmerge.asmx?wsdl');
8
9  // rest of your application here
10
11 unset($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`.

Introduction

The `ZendService\Rackspace\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.

Registering with Rackspace

Before you can get started with `ZendService\Rackspace\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\Rackspace` classes.

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

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

Available Methods

Each service class (Files, Servers) of Rackspace extends the `ZendService\Rackspace\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\Rackspace`. Return **true** in case of success and **false** in case of error.

setServiceNet `setServiceNet(boolean $useServiceNet = true)`

Use the Rackspace 'ServiceNet' internal network.

getServiceNet `getServiceNet()`

Are we using the Rackspace 'ServiceNet' internal network? Returns a boolean.

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.

getErrorMessage `getErrorMessage()`

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

ZendService\Rackspace\Servers

Overview

The `ZendService\Rackspace\Servers` is a class that provides a simple *API* to manage the [Rackspace Cloud Servers](#). Using this class you can:

- Create new servers
- List and get information on each server
- Delete a 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

Terminology

A **server** is a virtual machine instance in the Cloud Servers system. Flavor and image are requisite elements when creating a server.

A **server** is managed using the the class `ZendService\Rackspace\Servers\Server`.

A **flavor** is an available hardware configuration for a server. Each flavor has a unique combination of disk space, memory capacity and priority for CPU time.

An **image** is a collection of files used to create or rebuild a server. Rackspace provides a number of pre-built OS images by default. You may also create custom images from cloud servers you have launched. These custom images are useful

for backup purposes or for producing “gold” server images if you plan to deploy a particular server configuration frequently.

An **image** is managed using the the class `ZendService\Rackspace\Servers\Image`.

A **backup schedule** can be defined to create server images at regular intervals (daily and weekly). Backup schedules are configurable per server.

Public IP addresses can be shared across multiple servers for use in various high availability scenarios. When an IP address is shared to another server, the cloud network restrictions are modified to allow each server to listen to and respond on that IP address (you may optionally specify that the target server network configuration be modified). Shared IP addresses can be used with many standard heartbeat facilities (e.g. keepalived) that monitor for failure and manage IP failover.

A **shared IP group** is a collection of servers that can share IPs with other members of the group. Any server in a group can share one or more public IPs with any other server in the group. With the exception of the first server in a shared IP group, servers must be launched into shared IP groups. A server may only be a member of one shared IP group.

A **shared IP group** is managed using the the class `ZendService\Rackspace\Servers\SharedIpGroup`.

Quick Start

To use this class you have to pass the username and the API’s key of Rackspace in the construction of the class.

```
1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Servers($user, $key);
```

To create a new server you can use the **createServer** method.

```
1 $data = array (
2     'name'      => 'test',
3     'imageId'   => '49',
4     'flavorId'  => '1',
5 );
6
7 $server= $rackspace->createServer($data);
8
9 if (!$rackspace->isSuccessful()) {
10     die('ERROR: '.$rackspace->getErrMsg());
11 }
12
13 printf("Server name      : %s\n", $server->getName());
14 printf("Server Id       : %s\n", $server->getId());
15 printf("Admin password : %s\n", $server->getAdminPass());
```

This example create a server with name **test**, imageId 49, and flavorId 1. The attributes **name**, **imageId** and **flavorId** are required to create a new server. The result of **createServer** is an instance of `ZendService\Rackspace\Servers\Server`.

To get the public and private IP addresses of a server you can use the **getServerIp** method.

```
1 $id = '20054631';
2 $ips = $rackspace->getServerIp($id);
3
4 if (!$rackspace->isSuccessful()) {
```

```

5     die('ERROR: ' . $rackspace->getErrMsg());
6 }
7
8 echo "Private IPs:\n";
9 print_r($ips['private']);
10 echo "Public IPs:\n";
11 print_r($ips['public']);

```

This example get the IP addresses of the server with Id **20054631**. The result of **getServerIp** is an associative arrays with keys 'private' and 'public' contains all the private IP addresses and the public IP addresses of the server.

To get the list of all the available servers you can use the **listServers** method.

```

1 $servers= $rackspace->listServer(true);
2
3 if (!$rackspace->isSuccessful()) {
4     die('ERROR: ' . $rackspace->getErrMsg());
5 }
6
7 foreach ($servers as $srv) {
8     printf("Name      : %s\n", $srv->getName());
9     printf("Server Id : %s\n", $srv->getId());
10    printf("Image  Id : %s\n", $srv->getImageId());
11    printf("Flavor Id : %s\n", $srv->getFlavorId());
12    printf("Status   : %s (%d\%)\n", $srv->getStatus(), $srv->getProgress());
13 }

```

Available Methods

changeBackupSchedule `changeBackupSchedule(string $id, string $weekly, string $daily)`

This operation creates a new backup schedule or updates an existing backup schedule for the specified server. Return **true** in case of success, **false** in case of error.

\$id is the ID of the server

\$weekly, the day of the week for the backup (for instance "THURSDAY")

\$daily, specify the hours for the backup (for instance "H_0400_0600")

changeServerName `changeServerName(string $id, string $name)`

Change the name of a server. Return **true** in case of success, **false** in case of error.

\$id is the ID of the server

\$name is an optional parameter that specify the new name of the server

changeServerPassword `changeServerPassword(string $id, string $password)`

Change the admin password of a server. Return **true** in case of success, **false** in case of error.

\$id is the ID of the server

\$password is an optional parameter that specify the new admin password of the server

confirmResizeServer `confirmResizeServer(string $id)`

Confirm the resize of a server. During a resize operation, the original server is saved for a period of time to allow roll back if there is a problem. Once the newly resized server is tested and has been confirmed to be functioning

properly, use this operation to confirm the resize. After confirmation, the original server is removed and cannot be rolled back to. All resizes are automatically confirmed after 24 hours if they are not explicitly confirmed or reverted. Return **true** in case of success, **false** in case of error.

\$id is Id of the server.

createImage `createImage(string $serverId, string $name)`

Create an image from a server. Return a new instance of `ZendService\Rackspace\Servers\Image`. In case of error the return is **false**.

\$serverId is the Id of the server to use to create the image.

\$name, is the name of image to create

createSharedIpGroup `createSharedIpGroup(string $name, string $serverId)`

This operation creates a new shared IP group. Please note, on a create request, the shared IP group can be created empty or can be initially populated with a single server. Return the shared IP group as instance of `ZendService\Rackspace\Servers\SharedIpGroup` In case of error the return is **false**.

\$name is the name of the shared IP group to create.

\$serverId is the Id of the server.

createServer `createServer(array $data, $metadata=array(), $files=array())`

Create a server with the attributes specified in **\$data**. You can specify also optional parameters: **metadata** and **files**. Metadata is an array contains key/value of metadata related to the server and files is an array contains the paths of some files to upload into the server. The syntax used for the uploading of the files is 'serverPath' => 'localPath'. Return a new instance of `ZendService\Rackspace\Servers\Server`. In case of error the return is **false**.

\$data contains the parameters for the server. The required attributes to create a new server are:

- **name**, contains the name of the server
- **flavorId**, contains the flavor's Id to use
- **imageId**, contains the image's Id to use

\$metadata, contains the array of metadata information

\$files, contains the path of the files to upload in the server using the syntax 'serverPath' => 'localPath'.

disableBackupSchedule `disableBackupSchedule(string $id)`

Disable the backup of a server. Return **true** in case of success, **false** in case of error.

\$id is the Id of the server.

deleteImage `deleteImage(string $id)`

Delete a image. Return **true** in case of success, **false** in case of error.

\$id is the Id of the image.

deleteSharedIpGroup `deleteSharedIpGroup(string $id)`

Delete a shared IP group. Return **true** in case of success, **false** in case of error.

\$id is the Id of the shared IP group.

deleteServer `deleteServer(string $id)`

Delete a server. Return **true** in case of success, **false** in case of error.

\$id is the Id of the server.

getBackupSchedule getBackupSchedule(string \$id)

Return the backup schedule of a server. The return is an associative array with the following values: enabled, weekly, daily. In case of error the return is **false**.

\$id is the Id of the server.

getFlavor getFlavor(string \$flavorId)

Return the information about a flavor. The return is an associative array with the following values: id, ram, disk, name. In case of error the return is **false**.

\$flavorId is the Id of the flavor.

getImage getImage(string \$id)

Return an image as instance of ZendService\Rackspace\Servers\Image. In case of error the return is **false**.

\$id is the Id of the image.

getSharedIpGroup getSharedIpGroup(string \$id)

Return the shared IP group as instance of ZendService\Rackspace\Servers\SharedIpGroup In case of error the return is **false**.

\$id is the Id of the shared IP group.

getServer getServer(string \$id)

Return the server specified by the Id as instance of ZendService\Rackspace\Servers\Server. In case of error the return is **false**.

\$id is Id of the server.

getServerIp getServerIp(string \$id)

Return the public and private IP addresses of a server. Return an associative array contains the key '**public**' and '**private**' for the IP addresses. In case of error the return is **false**.

\$id is Id of the server.

getServerPrivateIp getServerPrivateIp(string \$id)

Return the private IP addresses of the server. Return an associative array contains the IP addresses. In case of error the return is **false**.

\$id is Id of the server.

getServerPublicIp getServerPublicIp(string \$id)

Return the public IP addresses of the server. Return an associative array contains the IP addresses. In case of error the return is **false**.

\$id is Id of the server.

listFlavors listFlavors(boolean \$details=false)

Return all the available flavors as associative array. In case of error the return is **false**.

If **\$details** is **true** return a detailed list, if is **false** return only the **name** and the **Id** of the flavor.

listImages listImages(boolean \$details=false)

Return all the available images as instance of ZendService\Rackspace\Servers\ImageList In case of error the return is **false**.

If **\$details** is **true** return a detailed list, if is **false** return only the **name** and the **Id** of the Image.

listServer `listServer(boolean $details=false)`

Return all the available servers with a new instance of `ZendService\Rackspace\Servers\ServerList`. In case of error the return is **false**.

If **\$details** is **true** return a detailed list, if is **false** return only the **name** and the **Id** of the server.

listSharedIpGroups `listSharedIpGroups(boolean $details=false)`

Return all the shared IP groups as instance of `ZendService\Rackspace\Servers\SharedIpGroupList`. In case of error the return is **false**.

If **\$details** is **true** return a detailed list, if is **false** return only the **name** and the **Id** of the shared IP group.

rebootServer `rebootServer(string $id, boolean $hard=false)`

Reboot a server. Return **true** in case of success, **false** in case of error.

\$id is Id of the server.

If **\$hard** is **false** (default) the server is rebooted in soft mode. That means the operating system is signaled to restart, which allows for a graceful shutdown of all processes. If **\$hard** is **true** the server is rebooted in hard mode. A hard reboot is the equivalent of power cycling the server.

rebuildServer `rebuildServer(string $id, string $imageId)`

Rebuild a server. The rebuild function removes all data on the server and replaces it with the specified image, server's Id and IP addresses will remain the same. Return **true** in case of success, **false** in case of error.

\$id is Id of the server.

\$imageId is the new Image Id of the server.

resizeServer `resizeServer(string $id, string $flavorId)`

Resize a server. The resize function converts an existing server to a different flavor, in essence, scaling the server up or down. The original server is saved for a period of time to allow rollback if there is a problem. All resizes should be tested and explicitly confirmed, at which time the original server is removed. All resizes are automatically confirmed after 24 hours if they are not explicitly confirmed or reverted. Return **true** in case of success, **false** in case of error.

\$id is Id of the server.

\$flavorId is the new flavor Id of the server.

revertResizeServer `revertResizeServer(string $id)`

Revert the resize of a server. During a resize operation, the original server is saved for a period of time to allow for roll back if there is a problem. If you determine there is a problem with a newly resized server, use this operation to revert the resize and roll back to the original server. All resizes are automatically confirmed after 24 hours if they have not already been confirmed explicitly or reverted. Return **true** in case of success, **false** in case of error.

\$id is Id of the server.

shareIpAddress `shareIpAddress(string $id, string $ip, string $groupId, boolean $configure=true)`

Share an IP address for a server. Return **true** in case of success, **false** in case of error.

\$id is Id of the server.

\$ip is the IP address to share.

\$groupId is the group Id to use.

If **\$configure** attribute is set to true, the server is configured with the new address, though the address is not enabled. Note that configuring the server does require a reboot.

unshareIpAddress unshareIpAddress(string \$id, string \$ip)

Unshare an IP address for a server. Return **true** in case of success, **false** in case of error.

\$id is Id of the server.

\$ip is the IP address to share.

updateServer updateServer(string \$id, string \$name=null, string \$password=null)

Change the name or/and the admin password of a server. In case of error the return is **false**.

\$id is the ID of the server

\$name is an optional parameter that specify the new name of the server

\$password is an optional parameter that specify the new admin password of the server

Examples

Authenticate

Check if the username and the key are valid for the Rackspace authentication.

```

1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Servers($user,$key);
5
6 if ($rackspace->authenticate()) {
7     printf("Authenticated with token: %s", $rackspace->getToken());
8 } else {
9     printf("ERROR: %s", $rackspace->getErrorMsg());
10 }

```

Create a server with metadata information and upload of a file

Create a server with some metadata information and upload the file **build.sh** from the local path **/home/user** to the remote path **/root**.

```

1 $data = array (
2     'name' => 'test',
3     'imageId' => '49',
4     'flavorId' => '1',
5 );
6 $metadata = array (
7     'foo' => 'bar',
8 );
9 $files = array (
10     '/root/build.sh' => '/home/user/build.sh',
11 );
12 $server= $rackspace->createServer($data,$metadata,$files);
13
14 if (!$rackspace->isSuccessful()) {

```

```
15     die('ERROR: ' . $rackspace->getErrorMsg());
16 }
17
18 $publicIp= $server->getPublicIp();
19
20 printf("Server name      : %s\n", $server->getName());
21 printf("Server Id       : %s\n", $server->getId());
22 printf("Public IP        : %s\n", $publicIp[0]);
23 printf("Admin password  : %s\n", $server->getAdminPass());
```

Reboot a server

Reboot a server in hard mode (is the equivalent of power cycling the server).

```
1 $flavors= $rackspace->rebootServer('server id', true)
2
3 if (!$rackspace->isSuccessful()) {
4     die('ERROR: ' . $rackspace->getErrorMsg());
5 }
6
7 echo "The server has been rebooted successfully";
```

List all the available flavors

List all the available flavors with all the detailed information.

```
1 $flavors= $rackspace->listFlavors(true);
2
3 if (!$rackspace->isSuccessful()) {
4     die('ERROR: ' . $rackspace->getErrorMsg());
5 }
6
7 print_r($flavors);
```

ZendService\Rackspace\Files

Overview

The `ZendService\Rackspace\Files` is a class that provides a simple *API* to manage the [Rackspace Cloud Files](#).

Quick Start

To use this class you have to pass the username and the API's key of Rackspace in the construction of the class.

```
1 $user = 'username';  
2 $key  = 'secret key';  
3  
4 $rackspace = new ZendService\Rackspace\Files($user, $key);
```

A container is a storage compartment for your data and provides a way for you to organize your data. You can think of a container as a folder in Windows® or a directory in UNIX®. The primary difference between a container and these other file system concepts is that containers cannot be nested. You can, however, create an unlimited number of containers within your account. Data must be stored in a container so you must have at least one container defined in your account prior to uploading data.

The only restrictions on container names is that they cannot contain a forward slash (/) and must be less than 256 bytes in length (please note that the length restriction applies to the name using the URL encoded format).

The containers are managed using the class `ZendService\Rackspace\Files\Container`.

An object (file) is the basic storage entity and any optional metadata that represents the files you store in the Cloud Files system. When you upload data to Cloud Files, the data is stored as-is (no compression or encryption) and consists of a location (container), the object's name, and any metadata consisting of key/value pairs. For instance, you may chose to store a backup of your digital photos and organize them into albums. In this case, each object could be tagged with metadata such as `Album : Caribbean Cruise` or `Album : Aspen Ski Trip`.

The only restriction on object names is that they must be less than 1024 bytes in length after URL encoding. Cloud Files has a limit on the size of a single uploaded object; by default this is 5 GB. For metadata, you should not exceed 90 individual key/value pairs for any one object and the total byte length of all key/value pairs should not exceed 4KB (4096 bytes).

The objects are managed using the class `ZendService\Rackspace\Files\Object`.

To create a new container you can use the **createContainer** method.

```
1 $container= $rackspace->createContainer('test');
2
3 if (!$rackspace->isSuccessful()) {
4     die('ERROR: '.$rackspace->getErrMsg());
5 }
6
7 printf("Name: %s", $container->getName());
```

This example create a container with name **test**. The result of **createContainer** is a new instance of `ZendService\Rackspace\Files\Container`.

To store an object (file) in a container you can use the **storeObject** method.

```
1 $name= 'example.jpg';
2 $file= file_get_contents($name);
3
4 $metadata= array (
5     'foo' => 'bar'
6 );
7
8 $rackspace->storeObject('test', $name, $file, $metadata);
9
10 if ($rackspace->isSuccessful()) {
11     echo 'Object stored successfully';
12 } else {
13     printf("ERROR: %s", $rackspace->getErrMsg());
14 }
```

This example store a file image **example.jpg** in the container **test** with the metadata specified in the array **\$metadata**.

To delete an object (file) you can use the **deleteObject** method.

```
1 $rackspace->deleteObject('test', 'example.jpg');
2
3 if ($rackspace->isSuccessful()) {
4     echo 'Object deleted successfully';
5 } else {
6     printf("ERROR: %s", $rackspace->getErrMsg());
7 }
```

This example delete the object **example.jpg** in the container **test**.

To publish a container as *CDN* (Content Delivery Network) you can use the **enableCdnContainer** method.

```
1 $cdnInfo= $rackspace->enableCdnContainer('test');
2
3 if ($rackspace->isSuccessful()) {
4     print_r($cdnInfo);
5 } else {
6     printf("ERROR: %s", $rackspace->getErrMsg());
7 }
```

This example publish the container **test** as *CDN*. If the operation is successful returns an associative arrays with the following values:

- **cdn_uri**, the url of the CDN container;
- **cdn_uri_ssl**, the ssl url of the CDN container;

Available Methods

copyObject `copyObject(string $container_source, string $obj_source, string $container_dest, string $obj_dest, $metadata=array(), string $content_type=null)`

Copy an object from a container to another. The return is **true** in case of success and **false** in case of error.

The **\$container_source** is the name of the source container.

The **\$obj_source** is the name of the source object.

The **\$container_dest** is the name of the destination container.

The **\$obj_dest** is the name of the destination object.

The **\$metadata** array contains the metadata information related to the destination object.

The **\$content_type** is the optional content type of the destination object (file).

createContainer `createContainer(string $container, $metadata=array())`

Create a container. The return is an instance of `ZendService\Rackspace\Files\Container`. In case of error the return is **false**.

The **\$container** is the name of the container to create.

The **\$metadata** array contains the metadata information related to the container.

deleteContainer `deleteContainer(string $container)`

Delete a container. The return is **true** in case of success and **false** in case of error.

The **\$container** is the name of the container to delete.

deleteObject `deleteObject(string $container, string $object)`

Delete an object in a specific container. Return **true** in case of success, **false** in case of error.

The **\$container** is the name of the container.

The **\$object** is the name of the object to delete.

enableCdnContainer `enableCdnContainer(string $container, integer $ttl=900)`

Publish a container as *CDN* (Content Delivery Network). Return an associative array contains the CDN url and SSL url. In case of error the return is **false**.

The **\$container** is the name of the container.

The **\$ttl** is the time to live for the CDN cache content. The default value is 15 minutes (900 seconds). The minimum TTL that can be set is 15 minutes (900 seconds); the maximum TTL is 50 years (range of 900 to 1577836800 seconds).

getCdnContainers `getCdnContainers($options=array())`

Returns all the CDN containers available. The return is an instance of `ZendService\Rackspace\Files\ContainerList`. In case of error the return is **false**.

The **\$options** contains the following optional parameters:

- **limit**, for an integer value n, limits the number of results to at most n values.
- **marker**, given a string value x, return object names greater in value than the specified marker.

getContainers `getContainers($options=array())`

Returns all the containers available. The return is an instance of `ZendService\Rackspace\Files\ContainerList` In case of error the return is **false**.

The **\$options** contains the following optional parameters:

- **limit**, for an integer value n, limits the number of results to at most n values.
- **marker**, given a string value x, return object names greater in value than the specified marker.

getContainer `getContainer(string $container)`

Returns the container specified as instance of `ZendService\Rackspace\Files\Container` In case of error the return is **false**.

The **\$container** is the name of the container.

getCountContainers `getCountContainers()`

Return the total count of containers.

getCountObjects `getCountObjects()`

Return the count of objects contained in all the containers.

getInfoCdnContainer `getInfoCdnContainer(string $container)`

Get the information of a CDN container. The result is an associative array with all the CDN information. In case of error the return is **false**.

The **\$container** is the name of the container.

getInfoContainers `getInfoContainers()`

Get the information about all the containers available. Return an associative array with the following values:

- **tot_containers**, the total number of containers stored
- **size_containers**, the total size, in byte, of all the containers.
- **tot_objects**, the total number of objects (file) stored in all the containers.

In case of error the return is **false**.

getMetadataContainer `getMetadataContainer(string $container)`

Get the metadata information of a container. The result is an associative array with all the metadata keys/values. In case of error the return is **false**.

The **\$container** is the name of the container.

getMetadataObject `getMetadataObject(string $container, string $object)`

Get the metadata information of an object. The result is an associative array with all the metadata keys/values. In case of error the return is **false**.

The **\$container** is the name of the container.

The **\$object** is the name of the object.

getObjects `getObjects(string $container, $options=array())`

Returns all the objects of a container. The return is an instance of `ZendService\Rackspace\Files\ObjectList`. In case of error the return is **false**.

The **\$container** is the name of the container.

The **\$options** contains the following optional parameters:

- **limit**, for an integer value *n*, limits the number of results to at most *n* values.
- **marker**, given a string value *x*, return object names greater in value than the specified marker.
- **prefix**, for a string value *x*, causes the results to be limited to object names beginning with the substring *x*.
- **path**, for a string value *x*, return the object names nested in the pseudo path.
- **delimiter**, for a character *c*, return all the object names nested in the container (without the need for the directory marker objects).

getObject `getObject(string $container, string $object, $headers=array())`

Returns an object of a container. The return is an instance of `ZendService\Rackspace\Files\Object`. In case of error the return is **false**.

The **\$container** is the name of the container.

The **\$object** is the name of the object.

The **\$headers** contains the following optional parameters (See the [RFC-2616](#) for more info):

- **If-Match**, a client that has one or more entities previously obtained from the resource can verify that one of those entities is current by including a list of their associated entity tags in the If-Match header field.
- **If-None-Match**, a client that has one or more entities previously obtained from the resource can verify that none of those entities is current by including a list of their associated entity tags in the If-None-Match header field.
- **If-Modified-Since**, if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server.
- **If-Unmodified-Since**, if the requested resource has not been modified since the time specified in this field, the server SHOULD perform the requested operation as if the If-Unmodified-Since header were not present.
- **Range**, Rackspace supports a sub-set of Range and do not adhere to the full RFC-2616 specification. We support specifying OFFSET-LENGTH where either OFFSET or LENGTH can be optional (not both at the same time). The following are supported forms of the header:
 - **Range: bytes=-5**, last five bytes of the object
 - **Range: bytes=10-15**, the five bytes after a 10-byte offset
 - **Range: bytes=32-**, all data after the first 32 bytes of the object

getSizeContainers `getSizeContainers()`

Return the size, in bytes, of all the containers.

setMetadataObject `setMetadataObject(string $container, string $object, array $metadata)`

Update metadata information to the object (all the previous metadata will be deleted). Return **true** in case of success, **false** in case of error.

The **\$container** is the name of the container.

The **\$object** is the name of the object to store.

The **\$metadata** array contains the metadata information related to the object.

storeObject `storeObject(string $container, string $object, string $file, $metadata=array())`

Store an object in a specific container. Return **true** in case of success, **false** in case of error.

The **\$container** is the name of the container.

The **\$object** is the name of the object to store.

The **\$file** is the content of the object to store.

The **\$metadata** array contains the metadata information related to the object to store.

updateCdnContainer `updateCdnContainer(string $container, integer $ttl=null, $cdn_enabled=null, $log=null)`

Update the attribute of a *CDN* container. Return an associative array contains the CDN url and SSL url. In case of error the return is **false**.

The **\$container** is the name of the container.

The **\$ttl** is the time to live for the CDN cache content. The default value is 15 minutes (900 seconds). The minimum TTL that can be set is 15 minutes (900 seconds); the maximum TTL is 50 years (range of 900 to 1577836800 seconds).

The **\$cdn_enabled** is the flag to switch on/off the CDN. **True** switch on, **false** switch off.

The **\$log** enable or disable the log retention. **True** switch on, **false** switch off.

Examples

Authenticate

Check if the username and the key are valid for the Rackspace authentication.

```
1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Files($user,$key);
5
6 if ($rackspace->authenticate()) {
7     printf("Authenticated with token: %s", $rackspace->getToken());
8 } else {
9     printf("ERROR: %s", $rackspace->getErrMsg());
10 }
```

Get an object

Get an image file (**example.gif**) from the cloud and render it in the browser

```
1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Files($user,$key);
```



```
5
6 $object= $rackspace->getObject('test','example.gif');
7
8 if (!$rackspace->isSuccessful()) {
9     die('ERROR: ' . $rackspace->getErrMsg());
10 }
11
12 header('Content-type: image/gif');
13 echo $object->getFile();
```

Create a container with metadata

Create a container (**test**) with some metadata information (**\$metadata**)

```
1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Files($user, $key);
5
6 $metadata= array (
7     'foo' => 'bar',
8     'foo2' => 'bar2',
9 );
10
11 $container= $rackspace->createContainer('test', $metadata);
12
13 if (!$rackspace->isSuccessful()) {
14     echo 'Container created successfully';
15 }
```

Get the metadata of a container

Get the metadata of the container **test**

```
1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Files($user, $key);
5
6 $container= $rackspace->getContainer('test');
7
8 if (!$rackspace->isSuccessful()) {
9     die('ERROR: ' . $rackspace->getErrMsg());
10 }
11
12 $metadata= $container->getMetadata();
13 print_r($metadata);
```

Store an object in a container

Store an object using a `ZendService\Rackspace\Files\Container` instance

```
1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Files($user, $key);
5
6 $container= $rackspace->getContainer('test');
7
8 if (!$rackspace->isSuccessful()) {
9     die('ERROR: ' . $rackspace->getErrorMsg());
10 }
11
12 $file      = file_get_contents('test.jpg');
13 $metadata = array (
14     'foo' => 'bar',
15 );
16
17 if ($container->addObject('test.jpg', $file, $metadata)) {
18     echo 'Object stored successfully';
19 }
```

Check if a container is CDN enabled

Check if the **test** container is CDN enabled. If it is not we enable it.

```
1 $user = 'username';
2 $key  = 'secret key';
3
4 $rackspace = new ZendService\Rackspace\Files($user, $key);
5
6 $container= $rackspace->getContainer('test');
7
8 if (!$rackspace->isSuccessful()) {
9     die('ERROR: ' . $rackspace->getErrorMsg());
10 }
11
12 if (!$container->isCdnEnabled()) {
13     if (!$container->enableCdn()) {
14         die('ERROR: ' . $rackspace->getErrorMsg());
15     }
16 }
17 printf(
18     "The container is CDN enabled with the following URLs:\n %s\n %s\n",
19     $container->getCdnUri(),
20     $container->getCdnUriSsl()
21 );
```

Introduction

ZendService\ReCaptcha\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.

Simplest use

Instantiate a ZendService\ReCaptcha\ReCaptcha object, passing it your public and private keys:

Creating an instance of the reCAPTCHA service

```
1 $recaptcha = new ZendService\ReCaptcha\ReCaptcha($pubKey, $privKey);
```

To render the reCAPTCHA, simply call the `getHTML()` method:

Displaying the reCAPTCHA

```
1 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

```
1 $result = $recaptcha->verify(  
2     $_POST['recaptcha_challenge_field'],  
3     $_POST['recaptcha_response_field']  
4 );
```

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

```
1 if (!$result->isValid()) {  
2     // Failed validation  
3 }
```

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.

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

```
1 // The mail address we want to hide  
2 $mail = 'mail@example.com';  
3  
4 // Create an instance of the mailhide component, passing it your public  
5 // and private keys, as well as the mail address you want to hide  
6 $mailHide = new ZendService\ReCaptcha\Mailhide();  
7 $mailHide->setPublicKey($pubKey);  
8 $mailHide->setPrivateKey($privKey);  
9 $mailHide->setEmail($mail);  
10  
11 // Display it  
12 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 356.1: ZendServiceReCaptchaMailHide options

Option	Description	Expected Values	Default Value
linkTitle	The title attribute of the link	string	'Reveal this e=mail address'
linkHidden-Text	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

```

1  // Create an instance of the mailhide component, passing it your public
2  // and private keys, as well as some configuration options
3  $mailHide = new ZendService\ReCaptcha\Mailhide();
4  $mailHide->setPublicKey($pubKey);
5  $mailHide->setPrivateKey($privKey);
6  $mailHide->setOptions(array(
7      'linkTitle' => 'Click me',
8      'linkHiddenText' => '+++++',
9  ));
10
11 // The mail addresses we want to hide
12 $mailAddresses = array(
13     'mail@example.com',
14     'johndoe@example.com',
15     'janedoe@example.com',
16 );
17
18 foreach ($mailAddresses as $mail) {
19     $mailHide->setEmail($mail);
20     print($mailHide);
21 }

```

ZendService\SlideShare

The `ZendService\SlideShare\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.

Getting Started with ZendService\SlideShare

In order to use the `ZendService\SlideShare\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\SlideShare` component.

Once you have setup an account, you can begin using the `ZendService\SlideShare\SlideShare` component by creating a new instance of the `ZendService\SlideShare\SlideShare` object and providing these values as shown below:

```
1 // Create a new instance of the component
2 $ss = new ZendService\SlideShare\SlideShare('APIKEY',
3                                             'SHAREDSECRET',
4                                             'USERNAME',
5                                             'PASSWORD');
```

The SlideShow object

All slide shows in the `ZendService\SlideShare\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.

```
1 class ZendService\SlideShare\SlideShow {
2
3     /**
```

```
4      * Retrieves the location of the slide show
5      */
6      public function getLocation() {
7          return $this->_location;
8      }
9
10     /**
11      * Gets the transcript for this slide show
12      */
13     public function getTranscript() {
14         return $this->_transcript;
15     }
16
17     /**
18      * Adds a tag to the slide show
19      */
20     public function addTag($tag) {
21         $this->_tags[] = (string) $tag;
22         return $this;
23     }
24
25     /**
26      * Sets the tags for the slide show
27      */
28     public function setTags(Array $tags) {
29         $this->_tags = $tags;
30         return $this;
31     }
32
33     /**
34      * Gets all of the tags associated with the slide show
35      */
36     public function getTags() {
37         return $this->_tags;
38     }
39
40     /**
41      * Sets the filename on the local filesystem of the slide show
42      * (for uploading a new slide show)
43      */
44     public function setFilename($file) {
45         $this->_slideShowFilename = (string) $file;
46         return $this;
47     }
48
49     /**
50      * Retrieves the filename on the local filesystem of the slide show
51      * which will be uploaded
52      */
53     public function getFilename() {
54         return $this->_slideShowFilename;
55     }
56
57     /**
58      * Gets the ID for the slide show
59      */
60     public function getId() {
61         return $this->_slideShowId;
```



```

62     }
63
64     /**
65      * Retrieves the HTML embed code for the slide show
66      */
67     public function getEmbedCode() {
68         return $this->_embedCode;
69     }
70
71     /**
72      * Retrieves the Thumbnail URi for the slide show
73      */
74     public function getThumbnailUrl() {
75         return $this->_thumbnailUrl;
76     }
77
78     /**
79      * Sets the title for the Slide show
80      */
81     public function setTitle($title) {
82         $this->_title = (string) $title;
83         return $this;
84     }
85
86     /**
87      * Retrieves the Slide show title
88      */
89     public function getTitle() {
90         return $this->_title;
91     }
92
93     /**
94      * Sets the description for the Slide show
95      */
96     public function setDescription($desc) {
97         $this->_description = (string) $desc;
98         return $this;
99     }
100
101     /**
102      * Gets the description of the slide show
103      */
104     public function getDescription() {
105         return $this->_description;
106     }
107
108     /**
109      * Gets the numeric status of the slide show on the server
110      */
111     public function getStatus() {
112         return $this->_status;
113     }
114
115     /**
116      * Gets the textual description of the status of the slide show on
117      * the server
118      */
119     public function getStatusDescription() {

```

```
120     return $this->_statusDescription;
121 }
122
123 /**
124  * Gets the permanent link of the slide show
125  */
126 public function getPermaLink() {
127     return $this->_permalink;
128 }
129
130 /**
131  * Gets the number of views the slide show has received
132  */
133 public function getNumViews() {
134     return $this->_numViews;
135 }
136 }
```

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\SlideShare` component, this data class will be used frequently to browse or add new slide shows to or from the web service.

Retrieving a single slide show

The simplest usage of the `ZendService\SlideShare\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\SlideShare` object and using the resulting `ZendService\SlideShare\SlideShow` object as shown.

```
1 // Create a new instance of the component
2 $ss = new ZendService\SlideShare\SlideShare('APIKEY',
3                                             'SHAREDSECRET',
4                                             'USERNAME',
5                                             'PASSWORD');
6
7 $slideshow = $ss->getSlideShow(123456);
8
9 print "Slide Show Title: {$slideshow->getTitle()}<br/>\n";
10 print "Number of views: {$slideshow->getNumViews()}<br/>\n";
```

Retrieving Groups of Slide Shows

If you do not know the specific ID of a slide show you are interested in retrieving, you can retrieve 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:

```

1 // Create a new instance of the component
2 $ss = new ZendService\SlideShare\SlideShare('APIKEY',
3                                             'SHAREDSECRET',
4                                             'USERNAME',
5                                             'PASSWORD');
6
7 $starting_offset = 0;
8 $limit = 10;
9
10 // Retrieve the first 10 of each type
11 $ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
12 $ss_tags = $ss->getSlideShowsByTag('zend', $starting_offset, $limit);
13 $ss_group = $ss->getSlideShowsByGroup('mygroup', $starting_offset, $limit);
14
15 // Iterate over the slide shows
16 foreach ($ss_user as $slideshow) {
17     print "Slide Show Title: {$slideshow->getTitle}<br/>\n";
18 }

```

ZendService\SlideShare 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:

```

1 $frontendOptions = array(
2     'lifetime' => 7200,
3     'automatic_serialization' => true);
4 $backendOptions = array(
5     'cache_dir' => '/webtmp/');
6
7 $cache = Zend\Cache\Cache::factory('Core',
8                                     'File',
9                                     $frontendOptions,
10                                    $backendOptions);
11
12 $ss = new ZendService\SlideShare\SlideShare('APIKEY',
13                                             'SHAREDSECRET',
14                                             'USERNAME',
15                                             'PASSWORD');
16 $ss->setCacheObject($cache);
17
18 $ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);

```

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 than default as shown:

```
1 $client = new Zend\Http\Client();
2 $client->setConfig(array('timeout' => 5));
3
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);
```

ZendService\StrikeIron

ZendService\StrikeIron\StrikeIron provides a *PHP 5* client to StrikeIron web services. See the following sections:

- *ZendServiceStrikeIronStrikeIron*
- *Bundled Services*
- *Advanced Use*

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\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\StrikeIron.

The ZendService\StrikeIron\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.

Registering with StrikeIron

Before you can get started with `ZendService\StrikeIron\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\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.

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`.

StrikeIron consists of hundreds of different web services. `ZendService\StrikeIron\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\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:

```
1 $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
2 ↪',
3                                     'password' => 'your-password'));
4 $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*.

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.

```

1 $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
  ↳ ',
2                                     'password' => 'your-password'));
3
4 // Get a client for the Sales & Use Tax Basic service
5 $taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
6
7 // Query tax rate for Ontario, Canada
8 $rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
9 echo $rateInfo->province;
10 echo $rateInfo->abbreviation;
11 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.

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()`:

```

1 <?php
2 $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
  ↳ ',
3                                     'password' => 'your-password'));
4
5 $taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
6
7 $rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
8 print_r($rateInfo);
9 ?>
10
11 ZendService\StrikeIron\Decorator Object
12 (
13     [_name:protected] => GetTaxRateCanadaResult
14     [_object:protected] => stdClass Object
15         (
16             [abbreviation] => ON
17             [province] => ONTARIO
18             [GST] => 0.06

```

```
19         [PST] => 0.08
20         [total] => 0.14
21         [HST] => Y
22     )
23 )
```

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()`.

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:

```
1  $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
2  ↪',
3                                     'password' => 'your-password'));
4  $taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
5
6  try {
7
8      $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
9
10 } catch (ZendService\StrikeIron\Exception\RuntimeException $e) {
11
12     // error handling for events like connection
13     // problems or subscription errors
14
15 }
```

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 in an 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.

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:

```
1 // Get a client for the Sales & Use Tax Basic service
2 $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
3 ↪',
4                                     'password' => 'your-password'));
5 $taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
6
7 // Check remaining hits for the Sales & Use Tax Basic service
8 $subscription = $taxBasic->getSubscriptionInfo();
9 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)`.

ZendService\StrikeIron: Bundled Services

ZendService\StrikeIron\StrikeIron comes with wrapper classes for three popular StrikeIron services.

ZIP Code Information

ZendService\StrikeIron\ZipCodeInfo provides a client for StrikeIron's Zip Code Information Service. For more information on this service, visit these StrikeIron resources:

- [Zip Code Information Service Page](#)
- [Zip Code Information Service WSDL](#)

The service contains a `getZipCode()` method that will retrieve information about a United States ZIP code or Canadian postal code:

```
1 $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
2                                     ↪',
3                                     'password' => 'your-password'));
4
5 // Get a client for the Zip Code Information service
6 $zipInfo = $strikeIron->getService(array('class' => 'ZipCodeInfo'));
7
8 // Get the Zip information for 95014
9 $response = $zipInfo->getZipCode(array('ZipCode' => 95014));
10 $zips = $response->serviceResult;
11
12 // Display the results
13 if ($zips->count == 0) {
14     echo 'No results found';
15 } else {
16     // a result with one single zip code is returned as an object,
17     // not an array with one element as one might expect.
18     if (! is_array($zips->zipCodes)) {
19         $zips->zipCodes = array($zips->zipCodes);
20     }
21 }
```

```
20
21 // print all of the possible results
22 foreach ($zips->zipCodes as $z) {
23     $info = $z->zipCodeInfo;
24
25     // show all properties
26     print_r($info);
27
28     // or just the city name
29     echo $info->preferredCityName;
30 }
31
32
33 // Detailed status information
34 // http://www.strikeiron.com/exampledata/StrikeIronZipCodeInformation_v3.pdf
35 $status = $response->serviceStatus;
```

U.S. Address Verification

ZendService\StrikeIron\USAddressVerification provides a client for StrikeIron's U.S. Address Verification Service. For more information on this service, visit these StrikeIron resources:

- [U.S. Address Verification Service Page](#)
- [U.S. Address Verification Service WSDL](#)

The service contains a `verifyAddressUSA()` method that will verify an address in the United States:

```
1 $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
2                                     'password' => 'your-password'));
3
4 // Get a client for the Zip Code Information service
5 $verifier = $strikeIron->getService(array('class' => 'USAddressVerification'));
6
7 // Address to verify. Not all fields are required but
8 // supply as many as possible for the best results.
9 $address = array('firm' => 'Zend Technologies',
10                 'addressLine1' => '19200 Stevens Creek Blvd',
11                 'addressLine2' => '',
12                 'city_state_zip' => 'Cupertino CA 95014');
13
14 // Verify the address
15 $result = $verifier->verifyAddressUSA($address);
16
17 // Display the results
18 if ($result->addressErrorNumber != 0) {
19     echo $result->addressErrorNumber;
20     echo $result->addressErrorMessage;
21 } else {
22     // show all properties
23     print_r($result);
24
25     // or just the firm name
26     echo $result->firm;
27
```

```

28     // valid address?
29     $valid = ($result->valid == 'VALID');
30 }

```

Sales & Use Tax Basic

ZendService\StrikeIron\SalesUseTaxBasic provides a client for StrikeIron's Sales & Use Tax Basic service. For more information on this service, visit these StrikeIron resources:

- [Sales & Use Tax Basic Service Page](#)
- [Sales & Use Tax Basic Service WSDL](#)

The service contains two methods, `getTaxRateUSA()` and `getTaxRateCanada()`, that will retrieve sales and use tax data for the United States and Canada, respectively.

```

1  $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
   ↪',
2                                     'password' => 'your-password'));
3
4  // Get a client for the Sales & Use Tax Basic service
5  $taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
6
7  // Query tax rate for Ontario, Canada
8  $rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'foo'));
9  print_r($rateInfo);           // show all properties
10 echo $rateInfo->GST;           // or just the GST (Goods & Services Tax)
11
12 // Query tax rate for Cupertino, CA USA
13 $rateInfo = $taxBasic->getTaxRateUS(array('zip_code' => 95014));
14 print_r($rateInfo);           // show all properties
15 echo $rateInfo->state_sales_tax; // or just the state sales tax

```

ZendService\StrikeIron: Advanced Uses

This section describes the more advanced uses of `ZendService\StrikeIron\StrikeIron`.

Using Services by WSDL

Some StrikeIron services may have a *PHP* wrapper class available, such as those described in *Bundled Services*. However, StrikeIron offers hundreds of services and many of these may be usable even without creating a special wrapper class.

To try a StrikeIron service that does not have a wrapper class available, give the `wsdl` option to `getService()` instead of the `class` option:

```
1 $strikeIron = new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username
2                                     'password' => 'your-password'));
3
4 // Get a generic client to the Reverse Phone Lookup service
5 $phone = $strikeIron->getService(
6     array('wsdl' => 'http://ws.strikeiron.com/ReversePhoneLookup?WSDL')
7 );
8
9 $result = $phone->lookup(array('Number' => '(408) 253-8800'));
10 echo $result->listingName;
11
12 // Zend Technologies USA Inc
```

Using StrikeIron services from the WSDL will require at least some understanding of the WSDL files. StrikeIron has many resources on its site to help with this. Also, [Jan Schneider](#) from the [Horde project](#) has written a [small PHP routine](#) that will format a WSDL file into more readable *HTML*.

Please note that only the services described in the *Bundled Services* section are officially supported.

Viewing SOAP Transactions

All communication with StrikeIron is done using the *SOAP* extension. It is sometimes useful to view the *XML* exchanged with StrikeIron for debug purposes.

Every StrikeIron client (subclass of `ZendService\StrikeIron\Base`) contains a `getSoapClient()` method to return the underlying instance of `SOAPClient` used to communicate with StrikeIron.

PHP'`SOAPClient` has a `trace` option that causes it to remember the *XML* exchanged during the last transaction. `ZendService\StrikeIron\StrikeIron` does not enable the `trace` option by default but this can easily be changed by specifying the options that will be passed to the `SOAPClient` constructor.

To view a SOAP transaction, call the `getSoapClient()` method to get the `SOAPClient` instance and then call the appropriate methods like `__getLastRequest()` and `__getLastResponse()`:

```
1 $strikeIron =
2     new ZendService\StrikeIron\StrikeIron(array('username' => 'your-username',
3                                                  'password' => 'your-password',
4                                                  'options' => array('trace' => true)));
5
6 // Get a client for the Sales & Use Tax Basic service
7 $taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
8
9 // Perform a method call
10 $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
11
12 // Get SOAPClient instance and view XML
13 $soapClient = $taxBasic->getSoapClient();
14 echo $soapClient->__getLastRequest();
15 echo $soapClient->__getLastResponse();
```


Introduction

`ZendService\Technorati\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.

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\Technorati`.

Making Your First Query

In order to run a query, first you need a `ZendService\Technorati\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

```
1 // create a new ZendService\Technorati\Technorati
2 // with a valid API_KEY
3 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
4
5 // search Technorati for PHP keyword
6 $resultSet = $technorati->search('PHP');
```

Each query method accepts an array of optional parameters that can be used to refine your query.

Refining your query

```
1 // create a new ZendService\Technorati\Technorati
2 // with a valid API_KEY
3 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
4
5 // filter your query including only results
6 // with some authority (Results from blogs with a handful of links)
7 $options = array('authority' => 'a4');
8
9 // search Technorati for PHP keyword
10 $resultSet = $technorati->search('PHP', $options);
```

A `ZendService\Technorati\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\Technorati` object as long as you need it.

Sending multiple queries with the same `ZendService\Technorati\Technorati` instance

```
1 // create a new ZendService\Technorati\Technorati
2 // with a valid API_KEY
3 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
4
5 // search Technorati for PHP keyword
6 $search = $technorati->search('PHP');
7
8 // get top tags indexed by Technorati
9 $topTags = $technorati->topTags();
```

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

```

1 // create a new ZendService\Technorati\Technorati
2 // with a valid API_KEY
3 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
4
5 // search Technorati for PHP keyword
6 // $resultSet is an instance of ZendService\Technorati\SearchResultSet
7 $resultSet = $technorati->search('PHP');
8
9 // loop over all result objects
10 foreach ($resultSet as $result) {
11     // $result is an instance of ZendService\Technorati\SearchResult
12 }

```

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

```

1 // create a new ZendService\Technorati\Technorati
2 // with a valid API_KEY
3 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
4
5 // search Technorati for PHP keyword
6 // $resultSet is an instance of ZendService\Technorati\SearchResultSet
7 $resultSet = $technorati->search('PHP');
8
9 // $result is an instance of ZendService\Technorati\SearchResult
10 $resultSet->seek(1);
11 $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

```

1 // create a new ZendService\Technorati\Technorati
2 // with a valid API_KEY
3 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
4
5 // get info about weppos author
6 $result = $technorati->getInfo('weppos');
7
8 $author = $result->getAuthor();
9 echo '<h2>Blogs authored by ' . $author->getFirstName() . " " .
10     $author->getLastName() . '</h2>';

```

```
11 echo '<ol>';
12 foreach ($result->getWeblogs() as $weblog) {
13     echo '<li>' . $weblog->getName() . '</li>';
14 }
15 echo "</ol>";
```

Please read the *ZendServiceTechnoratiTechnorati Classes* section for further details about response classes.

Handling Errors

Each `ZendService\Technorati\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\Technorati` query to fail. `ZendService\Technorati\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

```
1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 try {
3     $resultSet = $technorati->search('PHP');
4 } catch (ZendService\Technorati\Exception $e) {
5     echo "An error occurred: " . $e->getMessage();
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\Technorati` if you try to use it beyond this limit. You can get information about your *API* key usage using the `ZendService\Technorati\Technorati::keyInfo()` method.

`ZendService\Technorati\Technorati::keyInfo()` returns a `ZendService\Technorati\KeyInfoResult` object. For full details please see the [API reference guide](#).

Getting API key daily usage information

```
1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 $key = $technorati->keyInfo();
3
4 echo "API Key: " . $key->getApiKey() . "<br />";
5 echo "Daily Usage: " . $key->getApiQueries() . "/" .
6     $key->getMaxQueries() . "<br />";
```

Available Technorati Queries

`ZendService\Technorati\Technorati` provides support for the following queries:

- *Cosmos*
- *Search*
- *Tag*
- *DailyCounts*
- *TopTags*
- *BlogInfo*
- *BlogPostTags*
- *GetInfo*

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\Technorati::cosmos()` in the [API reference guide](#).

Cosmos Query

```

1  $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2  $resultSet = $technorati->cosmos('http://devzone.zend.com/');
3
4  echo "<p>Reading " . $resultSet->totalResults() .
5      " of " . $resultSet->totalResultsAvailable() .
6      " available results</p>";
7  echo "<ol>";
8  foreach ($resultSet as $result) {
9      echo "<li>" . $result->getWeblog()->getName() . "</li>";
10 }
11 echo "</ol>";

```

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

```

1  $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2  $resultSet = $technorati->search('zend framework');
3
4  echo "<p>Reading " . $resultSet->totalResults() .
5      " of " . $resultSet->totalResultsAvailable() .
6      " available results</p>";

```

```
7 echo "<ol>";
8 foreach ($resultSet as $result) {
9     echo "<li>" . $result->getWeblog()->getName() . "</li>";
10 }
11 echo "</ol>";
```

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

```
1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 $resultSet = $technorati->tag('php');
3
4 echo "<p>Reading " . $resultSet->totalResults() .
5     " of " . $resultSet->totalResultsAvailable() .
6     " available results</p>";
7 echo "<ol>";
8 foreach ($resultSet as $result) {
9     echo "<li>" . $result->getWeblog()->getName() . "</li>";
10 }
11 echo "</ol>";
```

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\Technorati::dailyCounts()` in the [API reference guide](#).

DailyCounts Query

```
1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 $resultSet = $technorati->dailyCounts('php');
3
4 foreach ($resultSet as $result) {
5     echo "<li>" . $result->getDate() .
6         "(" . $result->getCount() . ")</li>";
7 }
8 echo "</ol>";
```

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

```

1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 $resultSet = $technorati->topTags();
3
4 echo "<p>Reading " . $resultSet->totalResults() .
5      " of " . $resultSet->totalResultsAvailable() .
6      " available results</p>";
7 echo "<ol>";
8 foreach ($resultSet as $result) {
9     echo "<li>" . $result->getTag() . "</li>";
10 }
11 echo "</ol>";

```

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

```

1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 $result = $technorati->blogInfo('http://devzone.zend.com/');
3
4 echo '<h2><a href="' . (string) $result->getWeblog()->getUrl() . '"' .
5      $result->getWeblog()->getName() . '</a></h2>';

```

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

```

1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 $resultSet = $technorati->blogPostTags('http://devzone.zend.com/');
3
4 echo "<p>Reading " . $resultSet->totalResults() .
5      " of " . $resultSet->totalResultsAvailable() .
6      " available results</p>";
7 echo "<ol>";
8 foreach ($resultSet as $result) {
9     echo "<li>" . $result->getTag() . "</li>";
10 }
11 echo "</ol>";

```

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

```
1 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
2 $result = $technorati->getInfo('weppos');
3
4 $author = $result->getAuthor();
5 echo "<h2>Blogs authored by " . $author->getFirstName() . " " .
6     $author->getLastName() . "</h2>";
7 echo "<ol>";
8 foreach ($result->getWeblogs() as $weblog) {
9     echo "<li>" . $weblog->getName() . "</li>";
10 }
11 echo "</ol>";
```

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.

ZendService\Technorati 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\Technorati` objects, such as `ZendService\Technorati\Author` and `ZendService\Technorati\Weblog`.

The `ZendService\Technorati\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 *URLs* or blog name. For full details please see `ZendService\Technorati\Technorati` in the [API reference guide](#).

ZendService\Technorati\ResultSet

`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

```

1 // run a simple query
2 $technorati = new ZendService\Technorati\Technorati('VALID_API_KEY');
3 $resultSet = $technorati->search('php');
4
5 // $resultSet is now an instance of
6 // ZendService\Technorati\SearchResultSet
7 // it extends ZendService\Technorati\ResultSet
8 foreach ($resultSet as $result) {
9     // do something with your
10     // ZendService\Technorati\SearchResult object
11 }
```

ZendService\Technorati\CosmosResultSet

`ZendService\Technorati\CosmosResultSet` represents a Technorati Cosmos query result set.

Note: `ZendService\Technorati\CosmosResultSet` extends *ZendServiceTechnoratiResultSet*.

ZendService\Technorati\SearchResultSet

ZendService\Technorati\SearchResultSet represents a Technorati Search query result set.

Note: ZendService\Technorati\SearchResultSet extends *ZendServiceTechnoratiResultSet*.

ZendService\Technorati\TagResultSet

ZendService\Technorati\TagResultSet represents a Technorati Tag query result set.

Note: ZendService\Technorati\TagResultSet extends *ZendServiceTechnoratiResultSet*.

ZendService\Technorati\DailyCountsResultSet

ZendService\Technorati\DailyCountsResultSet represents a Technorati DailyCounts query result set.

Note: ZendService\Technorati\DailyCountsResultSet extends *ZendServiceTechnoratiResultSet*.

ZendService\Technorati\TagsResultSet

ZendService\Technorati\TagsResultSet represents a Technorati TopTags or BlogPostTags queries result set.

Note: ZendService\Technorati\TagsResultSet extends *ZendServiceTechnoratiResultSet*.

ZendService\Technorati\Result

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.

ZendService\Technorati\CosmosResult

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*.

ZendService\Technorati\SearchResult

`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*.

ZendService\Technorati\TagResult

`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*.

ZendService\Technorati\DailyCountsResult

`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*.

ZendService\Technorati\TagsResult

`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*.

ZendService\Technorati\GetInfoResult

`ZendService\Technorati\GetInfoResult` represents a single Technorati GetInfo query result object.

ZendService\Technorati\BlogInfoResult

`ZendService\Technorati\BlogInfoResult` represents a single Technorati BlogInfo query result object.

ZendService\Technorati\KeyInfoResult

`ZendService\Technorati\KeyInfoResult` represents a single Technorati KeyInfo query result object. It provides information about your *Technorati API Key daily usage*.

Introduction

`ZendService\Twitter\Twitter` provides a client for the [Twitter API](#). `ZendService\Twitter\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\Twitter` wraps all web service operations, including OAuth, and all methods return an instance of `ZendService\Twitter\Response`.

`ZendService\Twitter\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.

Quick Start

To get started, first you'll need to either create a new application with Twitter, or get the details of an existing one you control. To do this:

- Go to <https://dev.twitter.com/> and sign in.
- Go to <https://dev.twitter.com/apps>
- Either create a new application, or select an existing one.
- On the application's settings page, grab the following information:
 - From the header “OAuth settings”, grab the “Consumer key” and “Consumer secret” values.
 - From the header “Your access token”, grab the “Access token” and “Access token secret” values.

Armed with this information, you can now configure and create your `ZendService\Twitter\Twitter\Twitter` instance:

```
1 $config = array(  
2     'access_token' => array(  
3         'token' => 'twitter-access-token-here',  
4         'secret' => 'twitter-access-secret-here',  
5     ),  
6     'oauth_options' => array(  
7         'consumerKey' => 'twitter-consumer-key-here',  
8         'consumerSecret' => 'twitter-consumer-secret-here',  
9     ),  
10    'http_client_options' => array(  
11        'adapter' => 'Zend\Http\Client\Adapter\Curl',  
12        'curloptions' => array(  
13            CURLOPT_SSL_VERIFYHOST => false,  
14            CURLOPT_SSL_VERIFYPEER => false,  
15        ),  
16    ),  
17 );  
18  
19 $twitter = new Twitter($config);
```

Make sure you substitute the values you discovered earlier in the configuration before attempting to connect.

Note: Twitter has a known issue with the SSL certificate for their API endpoints, which requires that you use insecure settings for the SSL certificate verification.

Once you have the client configured, you can start consuming it:

```
1 // Verify your credentials:  
2 $response = $twitter->account->verifyCredentials();  
3 if (!$response->isSuccess()) {  
4     die('Something is wrong with my credentials!');  
5 }  
6  
7 // Search for something:  
8 $response = $twitter->search->tweets('#zf2');  
9 foreach ($response->toValue() as $tweet) {  
10     printf("%s\n- (%s)\n", $tweet->text, $tweet->user->name);  
11 }  
12
```

```

13 // Tweet something:
14 $twitter->statuses->update('Hello world!');

```

Every action you take returns a `ZendService\Twitter\Twitter\Response` object. This object contains some general purpose methods for determining the status of the response (`isSuccess()`, `isError()`), and otherwise acts as a value object containing the data returned. Essentially, if the response returns an object, you will be able to access the members listed by the [Twitter API documentation](#). In the case of responses that return arrays, such as the `$twitter->search->tweets()` example shown earlier, you should use the `toValue()` method of the response to retrieve the array.

If you wish to dive in more into how authentication works, and what methods are exposed, keep reading!

Authentication

With the exception of fetching the public timeline, `ZendService\Twitter\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\Twitter` is `ZendOAuth`.

Creating the Twitter Class

`ZendService\Twitter\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\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\Twitter` including constructor options. You may also use `ZendOAuth` directly and only pass the resulting access token into `ZendService\Twitter\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\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\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\Twitter`. The `ZendOAuth` documentation demonstrates the workflow and objects involved.

```

1 /**
2  * We assume $serializedToken is the serialized token retrieved from a database
3  * or even $_SESSION (if following the simple ZendOAuth documented example)
4  */
5 $token = unserialize($serializedToken);
6
7 $twitter = new ZendService\Twitter\Twitter(array(
8     'accessToken' => $token,
9     'oauth_options' => array(

```

```
10         'username' => 'johndoe',
11     ),
12 );
13
14 // verify user's credentials with Twitter
15 $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 by 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\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:

```
1 $twitter = new ZendService\Twitter\Twitter(array(
2     'access_token' => array( // or use "accessToken" as the key; both work
3         'token' => 'your-access-token',
4         'secret' => 'your-access-token-secret',
5     ),
6     'oauth_options' => array( // or use "oauthOptions" as the key; both work
7         'consumerKey' => 'your-consumer-key',
8         'consumerSecret' => 'your-consumer-secret',
9     ),
10 ));
```

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:

```
1 $twitter = new ZendService\Twitter\Twitter(array(
2     'access_token' => array( // or use "accessToken" as the key; both work
3         'token' => 'your-access-token',
4         'secret' => 'your-access-token-secret',
5     ),
6     'oauth_options' => array( // or use "oauthOptions" as the key; both work
7         'consumerKey' => 'your-consumer-key',
8         'consumerSecret' => 'your-consumer-secret',
9     ),
10     'http_client_options' => array(
11         'adapter' => 'Zend_Http\Client\Adapter\Curl',
12     ),
13 ));
```

Account Methods

Verifying credentials

`verifyCredentials()` tests if supplied user credentials are valid with minimal overhead.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->account->verifyCredentials();
```

Application Methods

Rating limit status

`rateLimitStatus()` returns the remaining number of *API* requests available to the authenticating user before the *API* limit is reached for the current hour.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->application->rateLimitStatus();
3 $userTimelineLimit = $response->resources->statuses->{'/statuses/user_timeline'}->
  ↳remaining;
```

Blocking Methods

Blocking a user

`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.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->blocks->create('usertoblock');
```

Removing a block

`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.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->blocks->destroy('blockeduser');
```

- `ids()` returns an array of user identifiers that the authenticating user is blocking.

Who are you blocking (identifiers only)

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->blocks->ids();
```

Who are you blocking

`list()` returns an array of user objects that the authenticating user is blocking.

```
1 $twitter = new ZendService\Twitter\Twitter($options);  
2 $response = $twitter->blocks->list();
```

Direct Message Methods

Retrieving recent direct messages received

`messages()` returns a list of the 20 most recent direct messages sent to the authenticating user.

```
1 $twitter = new ZendService\Twitter\Twitter($options);  
2 $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.

Retrieving recent direct messages sent

`sent()` returns a list of the 20 most recent direct messages sent by the authenticating user.

```
1 $twitter = new ZendService\Twitter\Twitter($options);  
2 $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.

Sending a direct message

`new()` sends a new direct message to the specified user from the authenticating user. Requires both the user and text parameters below.

```

1 $twitter = new ZendService\Twitter\Twitter($options);
2 $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

```

1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->directMessages->destroy(123548);

```

Favorites Methods

Retrieving favorites

`list()` returns the 20 most recent favorite statuses for the authenticating user or user specified by the *id* parameter.

```

1 $twitter = new ZendService\Twitter\Twitter($options);
2 $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.

Creating favorites

`create()` favorites the status specified in the *id* parameter as the authenticating user.

```

1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->favorites->create(12351);

```

Deleting a favorite

`destroy()` un-favorites the status specified in the *id* parameter as the authenticating user.

```

1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->favorites->destroy(12351);

```

Friendship Methods

Creating a friend

`create()` befriends the user specified in the *id* parameter with the authenticating user.

```
1 $twitter = new ZendService\Twitter\Twitter($options);  
2 $response = $twitter->friendships->create('mynewfriend');
```

Deleting a friend

`destroy()` discontinues friendship with the user specified in the *id* parameter and the authenticating user.

```
1 $twitter = new ZendService\Twitter\Twitter($options);  
2 $response = $twitter->friendships->destroy('myoldfriend');
```

Search Methods

Searching for tweets

`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.

```
1 $twitter = new ZendService\Twitter\Twitter($options);  
2 $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.

Status Methods

Retrieving the public timeline

`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.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->statuses->sample();
```

Retrieving the home timeline

`homeTimeline()` returns the 20 most recent statuses posted by the authenticating user and that user's friends.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $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.

Retrieving the user timeline

`userTimeline()` returns the 20 most recent statuses posted from the authenticating user.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $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.

Showing user status

`show()` returns a single status, specified by the *id* parameter below. The status' author will be returned inline.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->statuses->show(1234);
```

Updating user status

`update()` updates the authenticating user's status. This method requires that you pass in the status update that you want to post to Twitter.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $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.

Showing user replies

`mentionsTimeline()` returns the 20 most recent @replies (status updates prefixed with @username) for the authenticating user.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $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.

Deleting user status

`destroy()` destroys the status specified by the required *id* parameter.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->statuses->destroy(12345);
```

User Methods

Showing user information

`show()` returns extended information of a given user, specified by ID or screen name as per the required *id* parameter below.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $response = $twitter->users->show('myfriend');
```

Searching for users

`search()` will search for users matching the query provided.

```
1 $twitter = new ZendService\Twitter\Twitter($options);
2 $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.

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.

Installing the Windows Azure SDK

There are two development scenario's when working with Windows Azure.

- You can develop your application using `ZendService\WindowsAzure\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\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](#).

API Documentation

The `ZendService\WindowsAzure\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.

Features

`ZendService\WindowsAzure\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

Architecture

`ZendService\WindowsAzure\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\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.

orphan

ZendService\WindowsAzure\Storage\Blob

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.

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.

Creating a storage container

Using the following code, a blob storage container can be created on development storage.

Creating a storage container

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();  
2 $result = $storageClient->createContainer('testcontainer');  
3  
4 echo 'Container name is: ' . $result->Name;
```

Deleting a storage container

Using the following code, a blob storage container can be removed from development storage.

Deleting a storage container

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2 $storageClient->deleteContainer('testcontainer');
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2
3 // upload /home/maarten/example.txt to Azure
4 $result = $storageClient->putBlob(
5     'testcontainer', 'example.txt', '/home/maarten/example.txt'
6 );
7
8 echo 'Blob name is: ' . $result->Name;
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2
3 // copy example.txt to example2.txt
4 $result = $storageClient->copyBlob(
5     'testcontainer', 'example.txt', 'testcontainer', 'example2.txt'
6 );
7
8 echo 'Copied blob name is: ' . $result->Name;
```

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

```

1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2
3 // download file to /home/maarten/example.txt
4 $storageClient->getBlob(
5     'testcontainer', 'example.txt', '/home/maarten/example.txt'
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

```

1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2
3 // make container publicly available
4 $storageClient->setContainerAcl(
5     'testcontainer',
6     ZendService\WindowsAzure\Storage\Blob::ACL_PUBLIC
7 );

```

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`.

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

```

1 $fileHandle = fopen('azure://mycontainer/myfile.txt', 'r');
2
3 // ...
4
5 fclose($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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2 // registers azure:// on this storage client
3 $storageClient->registerStreamWrapper();
4
5
6 // or:
7
8 // registers blob:// on this storage client
9 $storageClient->registerStreamWrapper('blob://');
```

To unregister the stream wrapper, the `unregisterStreamWrapper()` method can be used.

Shared Access Signature

Windows Azure Blob 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=2009-08-17T09%3A56%3A17Z&sr=c&sp=w&sig=hscQ7SuInqd91OfMTwTkxabhJSasp%2BD
↪%2Fz8UqZAgn9s%3D
```

The above signature gives write access to the “phpazuretestshared1” container of the “phpstorage” account.

Generating a Shared Access Signature

When you are the owner of a Windows Azure Blob 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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2 $sharedAccessUrl = $storageClient->generateSharedAccessUrl(
3     'container1',
4     '',
5     'c',
6     'w',
7     $storageClient->isoDate(time() - 500),
8     $storageClient->isoDate(time() + 3000)
9 );
```

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

```

1 $storageClient = new ZendService\WindowsAzure\Storage\Blob();
2 $sharedAccessUrl = $storageClient->generateSharedAccessUrl(
3     'container1',
4     'test.txt',
5     'b',
6     'r',
7     $storageClient->isoDate(time() - 500),
8     $storageClient->isoDate(time() + 3000)
9 );

```

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=2009-08-17T09%3A56%3A17Z&sr=c&sp=w&sig=hscQ7SuInqd91OfMTwTkxabhJSaspx%2BD
↪%2Fz8UqZAgn9s%3D

```

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

```

1 $storageClient = new ZendService\WindowsAzure\Storage\Blob(
2     'blob.core.windows.net', 'phpstorage', ''
3 );
4 $storageClient->setCredentials(
5     new ZendService\WindowsAzure\Credentials\SharedAccessSignature()
6 );
7 $storageClient->getCredentials()->setPermissionSet(array(
8     'http://phpstorage.blob.core.windows.net/phpazuretestshared1?st=2009-08-17T09%3A06
↪%3A17Z&se=2009-08-17T09%3A56%3A17Z&sr=c&sp=w&sig=hscQ7SuInqd91OfMTwTkxabhJSaspx%2BD
↪%2Fz8UqZAgn9s%3D'
9 ));
10 $storageClient->putBlob(
11     'phpazuretestshared1', 'NewBlob.txt', 'C:\Files\dataforazure.txt'
12 );

```

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.

orphan

ZendService\WindowsAzure\Storage\Table

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.

Operations on tables

This topic lists some samples of operations that can be executed on tables.

Creating a table

Using the following code, a table can be created on Windows Azure production table storage.

Creating a table

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Table(  
2     'table.core.windows.net', 'myaccount', 'myauthkey'  
3 );  
4 $result = $storageClient->createTable('testtable');  
5  
6 echo 'New table name is: ' . $result->Name;
```

Listing all tables

Using the following code, a list of all tables in Windows Azure production table storage can be queried.

Listing all tables

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Table(  
2     'table.core.windows.net', 'myaccount', 'myauthkey'  
3 );  
4 $result = $storageClient->listTables();  
5 foreach ($result as $table) {  
6     echo 'Table name is: ' . $table->Name . "\r\n";  
7 }
```

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

```
1 class SampleEntity extends ZendService\WindowsAzure\Storage\TableEntity  
2 {  
3     /**  
4      * @azure Name  
5      */  
6     public $Name;  
7  
8     /**  
9      * @azure Age Edm.Int64  
10     */  
11     public $Age;  
12  
13     /**  
14      * @azure Visible Edm.Boolean  
15     */  
16     public $Visible = false;  
17 }
```

Note that if no schema class is passed into table storage methods, ZendService\WindowsAzure\Storage\Table automatically works with ZendService\WindowsAzure\Storage\DynamicTableEntity.

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 provides 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

```

1 class SampleEntity extends ZendService\WindowsAzure\Storage\TableEntity
2 {
3     /**
4      * @azure Name
5      */
6     public $Name;
7
8     /**
9      * @azure Age Edm.Int64
10     */
11     public $Age;
12
13     /**
14      * @azure Visible Edm.Boolean
15      */
16     public $Visible = false;
17 }
```

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

```

1 /**
2  * @azure <property name in Windows Azure> <optional property type>
3  */
4 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

```
1 /**
2  * @azure Age Edm.Int64
3  */
4 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.

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`

```
1 $target = new ZendService\WindowsAzure\Storage\DynamicTableEntity(
2     'partition1', '000001'
3 );
4 $target->Name = 'Name'; // Will add property "Name" of type "Edm.String"
5 $target->Age = 25;      // Will add property "Age" of type "Edm.Int32"
```

Optionally, a property type can be enforced:

Forcing property types on ZendServiceWindowsAzureStorageDynamicTableEntity

```

1 $target = new ZendService\WindowsAzure\Storage\DynamicTableEntity(
2     'partition1', '000001'
3 );
4 $target->Name = 'Name'; // Will add property "Name" of type "Edm.String"
5 $target->Age = 25; // Will add property "Age" of type "Edm.Int32"
6
7 // Change type of property "Age" to "Edm.Int32":
8 $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.

Entities API examples

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

```

1 $entity = new SampleEntity ('partition1', 'row1');
2 $entity->FullName = "Maarten";
3 $entity->Age = 25;
4 $entity->Visible = true;
5
6 $storageClient = new ZendService\WindowsAzure\Storage\Table(
7     'table.core.windows.net', 'myaccount', 'myauthkey'
8 );
9 $result = $storageClient->insertEntity('testtable', $entity);
10
11 // Check the timestamp and etag of the newly inserted entity
12 echo 'Timestamp: ' . $result->getTimestamp() . "\n";
13 echo 'Etag: ' . $result->getEtag() . "\n";
```

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

```

1 $storageClient = new ZendService\WindowsAzure\Storage\Table(
2     'table.core.windows.net', 'myaccount', 'myauthkey'
3 );
4 $entity = $storageClient->retrieveEntityById(
5     'testtable', 'partition1', 'row1', 'SampleEntity'
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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Table(  
2     'table.core.windows.net', 'myaccount', 'myauthkey'  
3 );  
4 $entity = $storageClient->retrieveEntityById(  
5     'testtable', 'partition1', 'row1', 'SampleEntity'  
6 );  
7  
8 $entity->Name = 'New name';  
9 $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)

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Table(  
2     'table.core.windows.net', 'myaccount', 'myauthkey'  
3 );  
4 $entity = $storageClient->retrieveEntityById(  
5     'testtable', 'partition1', 'row1', 'SampleEntity'  
6 );  
7  
8 $entity->Name = 'New name';  
9  
10 // last parameter instructs the Etag check:  
11 $result = $storageClient->updateEntity('testtable', $entity, true);
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Table(  
2     'table.core.windows.net', 'myaccount', 'myauthkey'  
3 );  
4 $entity = $storageClient->retrieveEntityById(  
5     'testtable', 'partition1', 'row1', 'SampleEntity'  
6 );  
7 $result = $storageClient->deleteEntity('testtable', $entity);
```

Performing queries

Queries in `ZendService\WindowsAzure\Storage\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

```

1 $storageClient = new ZendService\WindowsAzure\Storage\Table(
2     'table.core.windows.net', 'myaccount', 'myauthkey'
3 );
4 $entities = $storageClient->storageClient->retrieveEntities(
5     'testtable',
6     'Name eq \'Maarten\' and PartitionKey eq \'partition1\'',
7     'SampleEntity'
8 );
9
10 foreach ($entities as $entity) {
11     echo 'Name: ' . $entity->Name . "\n";
12 }

```

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

```

1 $storageClient = new ZendService\WindowsAzure\Storage\Table(
2     'table.core.windows.net', 'myaccount', 'myauthkey'
3 );
4 $entities = $storageClient->storageClient->retrieveEntities(
5     'testtable',
6     $storageClient->select()
7         ->from($tableName)
8         ->where('Name eq ?', 'Maarten')
9         ->andWhere('PartitionKey eq ?', 'partition1'),
10     'SampleEntity'
11 );
12
13 foreach ($entities as $entity) {
14     echo 'Name: ' . $entity->Name . "\n";
15 }

```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Table(  
2     'table.core.windows.net', 'myaccount', 'myauthkey'  
3 );  
4  
5 // Start batch  
6 $batch = $storageClient->startBatch();  
7  
8 // Insert entities in batch  
9 $entities = generateEntities();  
10 foreach ($entities as $entity) {  
11     $storageClient->insertEntity($tableName, $entity);  
12 }  
13  
14 // Commit  
15 $batch->commit();
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Table(  
2     'table.core.windows.net', 'myaccount', 'myauthkey'  
3 );  
4  
5 $sessionHandler = new ZendService\WindowsAzure\SessionHandler(  
6     $storageClient, 'sessiontable'  
7 );  
8 $sessionHandler->register();
```

The above classname registers the `ZendService\WindowsAzure\SessionHandler` session handler and will store sessions in a table called “sessiontable”.

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

```

1  $storageClient = new ZendService\WindowsAzure\Storage\Table(
2      'table.core.windows.net', 'myaccount', 'myauthkey'
3  );
4
5  $sessionHandler = new ZendService\WindowsAzure\SessionHandler(
6      $storageClient, 'sessionstable'
7  );
8  $sessionHandler->register();
9
10 session_start();
11
12 if (!isset($_SESSION['firstVisit'])) {
13     $_SESSION['firstVisit'] = time();
14 }
15
16 // ...

```

Warning: The `ZendService\WindowsAzure\SessionHandler` session handler should be registered before a call to `session_start()` is made!

orphan

ZendService\WindowsAzure\Storage\Queue

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.

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.

Creating a queue

Using the following code, a queue can be created on development storage.

Creating a queue

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Queue();
2 $result = $storageClient->createQueue('testqueue');
3
4 echo 'Queue name is: ' . $result->Name;
```

Deleting a queue

Using the following code, a queue can be removed from development storage.

Deleting a queue

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Queue();
2 $storageClient->deleteQueue('testqueue');
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Queue();
2
3 // 3600 = time-to-live of the message, if omitted defaults to 7 days
4 $storageClient->putMessage('testqueue', 'This is a test message', 3600);
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Queue();
2
3 // retrieve 10 messages at once
4 $messages = $storageClient->getMessages('testqueue', 10);
5
6 foreach ($messages as $message) {
7     echo $message->MessageText . "\r\n";
8 }
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Queue();
2
3 // retrieve 10 messages at once
4 $messages = $storageClient->getMessages('testqueue', 10);
```

```
5
6 foreach ($messages as $message) {
7     echo $message . "\r\n";
8
9     // Mark the message as processed
10    $storageClient->deleteMessage('testqueue', $message);
11 }
```

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

```
1 $storageClient = new ZendService\WindowsAzure\Storage\Queue();
2
3 // retrieve 10 messages at once
4 $messages = $storageClient->peekMessages('testqueue', 10);
5
6 foreach ($messages as $message) {
7     echo $message->MessageText . "\r\n";
8 }
```

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 367

Copyright Information

The following copyrights are applicable to portions of Zend Framework.

Copyright © 2005-2014 Zend Technologies Inc. (<http://www.zend.com>)

Introduction to Zend Framework 2

- *Overview*
- *Installation*

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 370

In-depth tutorial for beginners

In this tutorial we will create a Blog-Application from scratch. We will go through all the details you need to learn to create your own ZF2 Application.

- *Introducing our first “Blog” Module*
- *Introducing Services and the ServiceManager*
- *Preparing for different Database-Backends*
- *Introducing Zend\Db\Sql and Zend\Stdlib\Hydrator*
- *Understanding the Router*
- *Making use of Forms and Fieldsets*
- *Editing and Deleting Data*
- *Reviewing the Blog-application*

Getting Started With Zend Studio 10 & Zend Server 6

The user guide is provided to take you through building a Zend Framework 2 application using Zend Studio and Zend Server.

- *Getting Started with Zend Framework 2*
- *A quick tour of the skeleton application*
- *The MyTaskList application*
- *Listing tasks*
- *Application Diagnostics*
- *Conclusion*

Zend Framework Tool (ZFTool)

- *Zend Framework Tool (ZFTool)*

Learning Zend Framework 2

- *Learning Dependency Injection*
- *Unit Testing a Zend Framework 2 application*
- *Advanced Configuration Tricks*
- *Using the EventManager*
- *Using ZendNavigation in your Album Module*
- *Using ZendPaginator in your Album Module*
- *Setting up a database adapter*

- *Migration from Zend Framework 1*
- *Namespacing Old Classes*
- *Running Zend Framework 2 and Zend Framework 1 in parallel*

Zend\Authentication

- *Introduction to Zend\Authentication*
- *Database Table Authentication*
- *Digest Authentication*
- *HTTP Authentication Adapter*
- *LDAP Authentication*
- *Authentication Validator*

Zend\Barcode

- *Introduction to Zend\Barcode*
- *Barcode creation using Zend\Barcode\Barcode class*
- *Zend\Barcode Objects*
- *Zend\Barcode Renderers*

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*

Zend\Captcha

- *Introduction to Zend\Captcha*
- *Captcha Operation*
- *CAPTCHA Adapters*

Zend\Code\Generator

- *Introduction*
- *Zend\Code\Generator Reference*
- *Zend\Code\Generator Examples*

Zend\Config

- *Introduction to Zend\Config*
- *Theory of Operation*
- *Zend\Config\Reader*
- *Zend\Config\Writer*
- *Zend\Config\Processor*
- *The Factory*

Zend\Console

- *Introduction to Zend\Console*
- *Console routes and routing*
- *Console-aware modules*
- *Console-aware action controllers*
- *Console adapters*
- *Console prompts*

Zend\Console\Getopt

- *Zend\Console\Getopt*
- *Declaring Getopt Rules*
- *Fetching Options and Arguments*
- *Configuring Zend\Console\Getopt*

Zend\Crypt

- *Introduction to Zend\Crypt*
- *Encrypt/decrypt using block ciphers*
- *Encrypt/decrypt a file*
- *Key derivation function*
- *Password*
- *Public key cryptography*

Zend\Db

- *Zend\Db\Adapter*
- *Zend\Db\ResultSet*
- *Zend\Db\Sql*
- *Zend\Db\Sql\Ddl*
- *Zend\Db\TableGateway*
- *Zend\Db\RowGateway*
- *Zend\Db\Metadata*

Zend\Debug

- *Dumping Variables*

Zend\Di

- *Introduction to Zend\Di*
- *Zend\Di Quickstart*
- *Zend\Di Definition*
- *Zend\Di InstanceManager*
- *Zend\Di Configuration*

- *Zend\Di Debugging & Complex Use Cases*

Zend\Dom

- *Introduction to Zend\Dom*
- *Zend\Dom\Query*

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*

Zend\EventManager

- *The EventManager*

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*

Zend\File

- *Zend\File\ClassFileLocator*

Zend\Filter

- *Introduction to Zend\Filter*
- *Standard Filter Classes*
- *Word Filters*
- *File Filter Classes*
- *Filter Chains*
- *Zend\Filter\Inflector*
- *Using the StaticFilter*
- *Writing Filters*

Zend\Form

- *Introduction*
- *Quick Start*
- *Form Collections*
- *File Uploading*
- *Advanced use of forms*
- *Form Elements*
- *Form View Helpers*

Zend\Http

- *Zend\Http*
- *The Request Class*
- *The Response Class*
- *The Headers Class*
- *HTTP Client*
- *HTTP Client - Connection Adapters*
- *HTTP Client - Advanced Usage*
- *HTTP Client - Static Usage*

Zend\I18n

- *Translating*
- *I18n View Helpers*
- *I18n Filters*
- *I18n Validators*

Zend\InputFilter

- *Introduction*
- *Input filter specifications*
- *File Upload Input*

Zend\Json

- *Introduction*
- *Basic Usage*
- *Advanced Usage*
- *XML to JSON conversion*
- *Zend\Json\Server - JSON-RPC server*

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*

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*

Zend\Log

- *ZendLog*
- *Writers*
- *Filters*
- *Formatters*

Zend\Mail

- *Introduction to ZendMail*
- *Zend\Mai\Message*
- *Zend\Mai\Transport*
- *Zend\Mai\Transport\SmtpOptions*
- *Zend\Mai\Transport\FileOptions*

Zend\Math

- *Introduction to ZendMath*

Zend\Memory

- *Overview*
- *Memory Manager*
- *Memory Objects*

Zend\Mime

- *ZendMime*
- *Zend\Mime\Message*
- *Zend\Mime\Part*

Zend\ModuleManager

- *Introduction to the Module System*
- *The Module Manager*
- *The Module Class*
- *The Module Autoloader*
- *Best Practices when Creating Modules*

Zend\Mvc

- *Introduction to the MVC Layer*
- *Quick Start*
- *Default Services*
- *Routing*
- *The MvcEvent*
- *The SendResponseEvent*
- *Available Controllers*
- *Controller Plugins*
- *Examples*

Zend\Navigation

- *Introduction to ZendNavigation*
- *Quick Start*
- *Pages*
- *Containers*
- *View Helpers*
- *View Helper - Breadcrumbs*
- *View Helper - Links*
- *View Helper - Menu*
- *View Helper - Sitemap*
- *View Helper - Navigation Proxy*

Zend\Paginator

- *Introduction to ZendPaginator*
- *Usage*

- *Configuration*
- *Advanced usage*

Zend\Permissions\Acl

- *Introduction to Zend\Permissions\Acl*
- *Refining Access Controls*
- *Advanced Usage*

Zend\Permissions\Rbac

- *Introduction to Zend\Permissions\Rbac*
- *Methods*
- *Examples*

Zend\ProgressBar

- *Progress Bars*
- *File Upload Handlers*

Zend\Serializer

- *Introduction to Zend\Serializer*
- *Zend\Serializer\Adapter*

Zend\Server

- *Introduction to Zend\Server*
- *Zend\Server\Reflection*

Zend\ServiceManager

- *Zend\ServiceManager*
- *Zend\ServiceManager Quick Start*
- *Delegator service factories*
- *Lazy Services*

Zend\Session

- *Session Config*
- *Session Container*
- *Session Manager*
- *Session Save Handlers*
- *Session Storage*
- *Session Validators*

Zend\Soap

- *Zend\Soap\Server*
- *Zend\Soap\Client*
- *WSDL Accessor*
- *AutoDiscovery*

Zend\Stdlib

- *Zend\Stdlib\Hydrator*
- *Zend\Stdlib\Hydrator\Filter*
- *Zend\Stdlib\Hydrator\Strategy*
- *AggregateHydrator*
- *CompositeNamingStrategy*
- *IdentityNamingStrategy*
- *MapNamingStrategy*
- *UnderscoreNamingStrategy*

Zend\Tag

- *Introduction to Zend\Tag*
- *Creating tag clouds with Zend\Tag\Cloud*

Zend\Test

- *Introduction to Zend\Test*
- *Unit testing with PHPUnit*

Zend\Text

- *Zend\Text\Figlet*
- *Zend\Text\Table*

Zend\Uri

- *Zend\Uri*

Zend\Validator

- *Introduction to Zend\Validator*
- *Standard Validation Classes*
- *Alnum Validator*
- *Alpha Validator*
- *Barcode Validator*
- *Between Validator*
- *Callback Validator*
- `modules/zend.validator.credit-card`
- *Date Validator*
- *Db\RecordExists and Db\NoRecordExists Validators*
- *Digits Validator*
- `modules/zend.validator.email-address`
- *File Validation Classes*
- `modules/zend.validator.greater-than`
- *Hex Validator*
- *Hostname Validator*
- *Iban Validator*
- *Identical Validator*
- `modules/zend.validator.in-array`
- *Ip Validator*
- *Isbn Validator*
- *IsInstanceOf Validator*
- *IsFloat*
- *IsInt*
- `modules/zend.validator.less-than`
- `modules/zend.validator.not-empty`

- [modules/zend.validator.post-code](#)
- [Regex Validator](#)
- [Sitemap Validators](#)
- [Step Validator](#)
- [modules/zend.validator.string-length](#)
- [Timezone Validator](#)
- [Uri Validator](#)
- [Validator Chains](#)
- [Writing Validators](#)
- [Validation Messages](#)

Zend\Version

- [Getting the Zend Framework Version](#)

Zend\View

- [Zend\View Quick Start](#)
- [The PhpRenderer](#)
- [PhpRenderer View Scripts](#)
- [The ViewEvent](#)
- [View Helpers](#)
- [View Helper - BasePath](#)
- [View Helper - Cycle](#)
- [View Helper - Doctype](#)
- [FlashMessenger Helper](#)
- [Gravatar Helper](#)
- [View Helper - HeadLink](#)
- [View Helper - HeadMeta](#)
- [View Helper - HeadScript](#)
- [View Helper - HeadStyle](#)
- [View Helper - HeadTitle](#)
- [View Helper - HtmlList](#)
- [View Helper - HTML Object](#)
- [View Helper - Identity](#)
- [View Helper - InlineScript](#)
- [View Helper - JSON](#)

- *View Helper - Partial*
- *View Helper - Placeholder*
- *View Helper - URL*
- *Advanced usage of helpers*

Zend\XmlRpc

- *Introduction to Zend\XmlRpc*
- *Zend\XmlRpc\Client*
- *Zend\XmlRpc\Server*

Services for Zend Framework 2 Reference

ZendService\Akismet

- *ZendService\Akismet*

ZendService\Amazon

- *ZendService\Amazon*
- *ZendService\Amazon\S3*
- *ZendService\Amazon\Sqs*
- *ZendService\Amazon\Ec2*
- *ZendService\Amazon\Ec2: CloudWatch Monitoring*
- *ZendService\Amazon\Ec2: Elastic Block Storage (EBS)*
- *ZendService\Amazon\Ec2: Elastic IP Addresses*
- *ZendService\Amazon\Ec2: Instances*
- *modules/zendservice.amazon.ec2.keypairs*
- *ZendService\Amazon\Ec2: Regions and Availability Zones*
- *ZendService\Amazon\Ec2: Reserved Instances*
- *ZendService\Amazon\Ec2: Security Groups*
- *ZendService\Amazon\Ec2: Windows Instances*

ZendService\AppleApns

- *ZendService\Apple\Apns*

ZendService\Audioscrobbler

- *ZendService\Audioscrobbler*

ZendService\Delicious

- *ZendService\Delicious*

ZendService\DeveloperGarden

- *ZendService\DeveloperGarden*

ZendService\Flickr

- *ZendService\Flickr*

ZendService\Google\Gcm

- *ZendService\Google\Gcm*

ZendService\LiveDocx

- *ZendService\LiveDocx*

ZendService\Rackspace

- *ZendService\Rackspace*
- *ZendService\Rackspace\Servers*
- *ZendService\Rackspace\Files*

ZendService\ReCaptcha

- *ZendService\ReCaptcha*

ZendService\SlideShare

- *ZendService\SlideShare*

ZendService\StrikeIron

- *ZendService\StrikeIron*
- *ZendService\StrikeIron: Bundled Services*
- *ZendService\StrikeIron: Advanced Uses*

ZendService\Technorati

- *ZendService\Technorati*

ZendService\Twitter

- *ZendService\Twitter*

ZendService\WindowsAzure

- *ZendService\WindowsAzure*

CHAPTER 377

Copyright

- *Copyright Information*

CHAPTER 378

Indices and tables

- *Programmer's Reference Guide of Zend Framework 2*
- [search](#)

F

`fromRoute()` (built-in function), [1126](#)